applied sciences

MDPI

Article

Technique for Searching Data in a Cryptographically Protected SQL Database

Vitalii Yesin [1,2], Mikolaj Karpinski [3,4,*], Maryna Yesina [1,2], Vladyslav Vilihura [1], Ruslan Kozak [4] and Ruslan Shevchuk [5,6,*]

1 Department of Security of Information Systems and Technologies, Faculty of Computer Science, V. Karazin National University of Kharkiv, 61022 Kharkiv, Ukraine; v.i.yesin@karazin.ua (V.Y.); m.v.yesina@karazin.ua (M.Y.); viligura93@gmail.com (V.V.)
2 Department of Information Technology Security, Institute of Computer Technologies, Automation and Metrology, Lviv Polytechnic National University, 79000 Lviv, Ukraine
3 Department of Computer Science, Faculty of Engineering Sciences, University of Applied Sciences in Nowy Sacz, 33-300 Nowy Sacz, Poland
4 Department of Cyber Security, Faculty of Computer Information Systems and Software Engineering, Ternopil Ivan Puluj National Technical University, 46001 Ternopil, Ukraine; ruslank@tntu.edu.ua
5 Department of Computer Science and Automatics, Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, 43-309 Bielsko-Biala, Poland
6 Department of Computer Science, Faculty of Computer Information Technologies, West Ukrainian National University, 46009 Ternopil, Ukraine
* Correspondence: mpkarpinski@gmail.com (M.K.); rshevchuk@ubb.edu.pl (R.S.)

Abstract: The growing popularity of data outsourcing to third-party cloud servers has a downside, related to the serious concerns of data owners about their security due to possible leakage. The desire to reduce the risk of loss of data confidentiality has become a motivating start to developing mechanisms that provide the ability to effectively use encryption to protect data. However, the use of traditional encryption methods faces a problem. Namely, traditional encryption, by making it impossible for insiders and outsiders to access data without knowing the keys, excludes the possibility of searching. This paper presents a solution that provides a strong level of confidentiality when searching, inserting, modifying, and deleting the required sensitive data in a remote database whose data are encrypted. The proposed SQL query processing technique allows the DBMS server to perform search functions over encrypted data in the same way as in an unencrypted database. This is achieved through the organization of automatic decryption by specially developed secure software of the corresponding data required for search, without the possibility of viewing these data itself. At that, we guarantee the integrity of the stored procedures used and special tables that store encrypted modules of special software and decryption keys, the relevance and completeness of the results returned to the application. The results of the analysis of the feasibility and effectiveness of the proposed solution show that the proper privacy of the stored data can be achieved at a reasonable overhead.

Keywords: database; security; database management system (DBMS); confidentiality; encryption; searchable encryption

Citation: Yesin, V.; Karpinski, M.; Yesina, M.; Vilihura, V.; Kozak, R.; Shevchuk, R. Technique for Searching Data in a Cryptographically Protected SQL Database. *Appl. Sci.* **2023**, *13*, 11525. https://doi.org/10.3390/app132011525

Academic Editor: Luis Javier Garcia Villalba

Received: 27 September 2023
Revised: 13 October 2023
Accepted: 19 October 2023
Published: 20 October 2023

1. Introduction

Today, storing and processing data on third-party remote cloud servers is widely used, showing explosive growth [1]. However, as the scale, value, and centralization of data increases, the reverse side of this process is revealed—the problems of ensuring the security and privacy of data are aggravated, which causes serious concern for owners and users of data. There is an identified risk that data stored in databases may be compromised [2], and this, in accordance with various international laws and standards such as: General Data Protection Regulation (GDPR [3], Payment Card Industry Data Security Standard

(PCI DSS) [4], the Health Insurance Portability and Accountability Act (HIPAA) [5], and some others, cannot be allowed. The owner of the data must be sure that the data stored on the third party remote servers of the service provider are protected from theft by outsiders. Moreover, these data must be protected even from the service provider itself (a valid user, known as an insider), if the respective provider cannot be trusted.

As you know, one of the fundamental solutions to this problem is the use of relevant cryptographic methods and primitives. Encryption is the standard approach to providing data confidentiality that is outsourced to so-called honest-but-curious cloud servers. Encryption makes it impossible for both insiders and outsiders to access data without knowing the keys. However, encryption also has a downside. The direct use of traditional data encryption/decryption approaches in most cases makes it difficult to perform search operations in encrypted data [6–8]. A simple solution to this problem is to download the entire dataset of the corresponding storage, then decrypt it locally and search for the required data. This approach creates serious performance issues that negate the benefits of outsourcing, making it unacceptable for most applications. The other method allows the server to decrypt the data, execute the query on the server side, and send only the results back to the user. However, in this case, the level of security is reduced, since data protected by encryption can potentially become available to the service provider (privileged user). Therefore, it is desirable to support the fullest possible server-side search functionality with the least possible loss of data confidentiality. In particular, a secure search system should aim to ensure that the service provider does not learn anything about the data stored in the secure database or about the queries, and the requester of the relevant data (querier) learns nothing, except for the query results [2].

The problem of searching data In encrypted databases has aroused great interest both in the scientific community and in industry [9]. To solve the problem of providing a search in cryptographically protected databases, relevant studies were carried out related to the development of new cryptographic primitives, new data structures for searchable encryption, and the development of views on security [6,10]. The solutions available today for searching encrypted data combine non-trivial ideas from cryptography, from the main provisions of the theory of algorithms and data structures, information search, and databases [2,6,11]. However, despite the wide variety of options offered, there is no dominant solution for all use cases. The goal of a security plan, according to Andress [12], is to find the balance between protection, usability, and cost. Similar views are held by Fuller et al. [2], who believe that designing a protected search system is a balance between security, functionality, performance, and usability. Therefore, it is important for data owners and users to understand how a fairly wide range of secure database systems are offered for their various applications and what compromises are acceptable for their respective use case. All this has stimulated research in the field of secure data management and increased its relevance.

The objective of our paper is to present a solution that allows the user and the server to use the standard structured query language (SQL) for secure data search in a remote database (DB) encrypted using a reliable and efficient cryptosystem, as well as to perform a secure insert, modify, and delete the required sensitive data at a reasonable overhead. Security in this case is associated with information that, during the operation of the searchable encryption (SE) scheme, must be protected and cannot be disclosed by an attacker with access to the database server. The proposed solution provides a practical and reliable level of confidentiality (the solution allows only access pattern leakage) without significant system complexity, increasing the amount of memory required, with an acceptable query execution time. In general, the security of our solution is based on the strength of the underlying cryptographic primitives, namely symmetric-key encryption. At the same time, our solution does not hide the database schema, the general structure (schema) of tables, or the number of their rows. Each database table can be fully encrypted (every column), partially encrypted (some columns are encrypted, some not), or fully unencrypted.

The main contribution of the authors is the creation of a SQL query processing technique that allows the DBMS (database management system) server to perform search functions in encrypted data (using basic data manipulation language operators) the same way as in an unencrypted database. This is achieved through the organization of prompt automatic decryption by specially developed secure software of the relevant data required for search, without the possibility of viewing this open data. The latter also applies to privileged users (insiders), primarily those who use the documentation tools of the DBMS server. Our technique involves separate processing of the SQL query on the side of the DBMS server and the application server (database proxy). On the application server, the initial request is first converted to the required form, and then transferred to the database server for execution. After the request is executed on the DBMS server, the requested encrypted data are transferred to the database proxy server for decryption and transfer to the user (client). The DBMS server cannot decrypt stored data (encrypted with a strong algorithm) that is not requested by the application. We offer a basis for implementing our solution on the server side of the Oracle DBMS (versions 12c and later) using unmodified DBMS software using our own developed persistent stored modules (PSM). We also provide examples of transformed queries and evaluate the practicality of our solution by testing it on a database that stores sensitive data.

The rest of this paper is organized as follows: Section 2 presents related works from the literature; Section 3 sets out the main security aspects relevant to the current situation. Section 4 presents the proposed technique for searching data in an encrypted database and its features. Section 5 presents the comparative evaluation results of the search performance for the required data in encrypted and unencrypted database tables. Section 6 concludes this work.

2. Related Works

Security, as is known, is associated with information that, during the operation of searchable encryption schemes, is revealed or leaked to an attacker who has access to the database server. Bösch et al. [6] believe that information leakage is possible in such schemes, which can be divided into three groups:

(a) index information (refers to the information about the keywords contained in the index);
(b) search pattern (information that can be obtained by knowing whether two search results refer to the same keyword);
(c) an access pattern (refers to information that is implied by the query (search) results, namely which documents contain the requested keyword for each of the queries [13] or which document identifiers match the query [14]).

Bösch et al. [6] note that in many schemes, there is leakage of at least the search pattern and the access pattern. At that, identifying the search pattern may not be a problem in some scenarios, whereas for others it is unacceptable. For example, in a medical database, disclosing a search pattern through statistical analysis (which allows an attacker to get full information about the plaintext keywords) can lead to the leakage of a large amount of information. This information can be used to match it with other (anonymous) public databases.

Fuller et al. [2] distinguish two types of entities that can pose a security threat to a database: a valid user known as an insider who performs one or more roles and an outsider. The latter can monitor and potentially modify network interactions between valid users, separating attackers into those that persist for the lifetime of the database and those that obtain a snapshot at a single point in time. At that, attackers are divided into those that persist for the lifetime of the database and those that obtain a snapshot at one a single point in time [15]. In addition, Fuller et al. [2] differentiate attackers into those who are: semi-honest (or honest-but-curious), i.e., those who follow the prescribed protocols, but may try to get additional information from data that they observe; and malicious, that is, those that actively perform actions aimed at obtaining additional information or influencing the operation of the system. They also note that much of the active research in protected

search technology considers semi-honest security against a persistent insider adversary. At that, special attention is paid to such types of objects within a protected search system that are vulnerable to leaks, such as: (a) data items and any indexing data structures; (b) queries; (c) records returned in response to queries or other relationships between data items and queries; (d) access control rules and the results of their application.

The cryptographic community has developed several common primitives:

- fully homomorphic encryption [16–19],
- functional encryption [20,21] with its subclasses and earlier representatives:
  - predicate encryption [22,23],
  - identity-based encryption [24],
  - attribute-based encryption [25]

and some others that completely or partially solve the problem of searching in a secure database. Protected search techniques are often based on these primitives, but rarely rely solely on one of them. Instead, they tend to use specialized protocols, often with some leakage in order to improve performance [2].

One possible approach to reduce the damage caused by a server compromise is to encrypt sensitive data and run all computation (application logic) on the clients. However, as noted by Popa et al. [26], some important applications are not suitable for this approach. For example, database-backed websites that process queries to generate data for the user, and applications that compute large amounts of data. Another possible approach is the use of such theoretical solutions as fully homomorphic encryption (FHE) [16–19]. Its use allows servers to compute arbitrary functions over encrypted data while only clients see the decrypted data. However, one of the problems of schemes with fully homomorphic encryption is performance, since current schemes require large computational resources and large storage overheads [6,26]. For some applications, so-called somewhat homomorphic encryption schemes may be used. These schemes are more efficient than FHE, but only allow a certain number of additions and multiplications [16,18]. The main problem when using somewhat or fully homomorphic encryption is that the resulting search schemes require a linear search time in the length of the dataset and this is too slow for practical use in modern applications.

As noted earlier, the problem of searching over encrypted data is of great interest from both theoretical and practical points of view. This is explained by the importance of ensuring the security and privacy of data stored and processed on third-party remote cloud servers of the service provider. However, as noted by some experts in this field [9,27], research on this topic is more focused on the scenario of a user who outsources an encrypted set of documents (such as e-mails or medical records) and would like to continue keyword search in this encrypted dataset. However, in practice, many companies, organizations, and institutions store data in databases that use the relational data model. Users are accustomed to using widely accepted SQL, which allows them to store, query, and update their data in a convenient way. Databases that support SQL (this applies in general to both NewSQL and some NoSQL databases that also allow you to work in the SQL query paradigm) provide fast search and retrieval of records, provided that the database can read out the data contents. However, encryption makes it difficult to search encrypted databases. Therefore, the direct application of solutions to search for the required information in the encrypted data of traditional databases is not an easy task.

In order to solve certain issues, Hacigümüş et al. [28] have developed techniques by which the bulk of the work of executing SQL queries can be performed by the service provider without the need to decrypt the stored data. The paper explores an algebraic structure for query splitting to minimize client-side computations. Using a so-called "coarse index" allows you to partially execute the SQL query on the provider side. The result of this query is sent to the client. The final correct result of the query is found by decrypting the data and executing a compensation query on the client side.

Popa et al. [26] proposed a system called CryptDB that supports SQL queries over encrypted data. This solution is based on various types of encryption, such as random

(RND), deterministic (DET), and order-preserving encryption (OPE), applied to a SQL table column. To request data from an encrypted database, CryptDB converts an unencrypted SQL query into its encrypted equivalent and decrypts the appropriate encryption layers. CryptDB achieves its goals using three ideas: running queries over encrypted data using a new encryption strategy with SQL support, dynamically adjusting the encryption level using encryption onions to minimize the information disclosed to the untrusted DBMS server, and chaining encryption keys to user passwords in a way that only authorized users can access to encrypted data. At that, although CryptDB protects data confidentiality, it does not guarantee the integrity, actuality, or completeness of the results returned to the application. However, the main disadvantage of CryptDB, as noted by Azraoui et al. [9], is that whenever one layer is removed, the encryption scheme becomes weak. In light of this, the main problem is to provide a practical solution for searching over encrypted databases that does not suffer from the leakage occurring in CryptDB and that provides transparent processing of complex queries over encrypted SQL databases. In their paper [9], the authors attempt to solve this problem by proposing a practical construct for searching data in an encrypted SQL databases that limits information leakage. Their solution is based on the searchable encryption technique developed by Curtmola et al. [29] and applied to unstructured documents. This mechanism creates an inverted search index of keywords in the database to enable keyword search queries over encrypted data. The practicality of this solution is achieved through the use of the cuckoo hashing technique, which makes the search in the index efficient. The proposed solution supports Boolean and range queries.

Pilyankevich et al. [27] propose a system (called Acra) which allows, among other things, to provide a search for encrypted data in SQL databases. The proposed Acra Searchable Encryption (Acra SE) solution is based on a blind indexing approach that develops the original idea of the CipherSweet project [30]. The main component of the Acra SE scheme is the so-called Acra Server, which works as a reverse proxy (transparent encryption/decryption proxy server). It sits between the application and the database. The application does not know that the data are encrypted before it gets into the database, the database does not know that someone encrypted the data. It is worth noting that the encryption and secure search functions of Acra Server can be configured for each column. This means that every table in the database can be fully encrypted (every column), partially encrypted (some columns are encrypted, some not), or fully unencrypted. All Acra's searchable encryption security properties are very similar to the security properties of CipherSweet, which poses the risk of partially known plaintext attacks. In this connection, Pilyankevich et al. [27] provide practical recommendations to ensure security. However, despite certain solutions aimed at ensuring the security of storing and searching for sensitive data, Acra, like CipherSweet, which was taken as a prototype of a searchable encryption scheme, supports the minimum functionality of queries, namely, only for equality.

Various DBMSs are characterized by the so-called technology of "transparent data encryption" (TDI) [31], which allows you to selectively encrypt sensitive data stored in database files, as well as in files related to data recovery, such as redo logs, archive logs, backup tapes. The essence of transparent encryption is that a combination of two keys is used: a key for each database table, which is unique, a master key that is stored outside the database in the so-called "wallet". Data stored on disk are encrypted; however, they are automatically decrypted for the legitimate user to process queries. That is, when the user selects encrypted columns, the DBMS quietly extracts the key from the "wallet", decrypts the columns and shows them to the user. As a result, the server must have access to the decryption keys, and an attacker who has compromised the DBMS software can gain access to all data. Therefore, the main goal of TDE is to protect sensitive data located in the corresponding files of the operating system. TDE is not a full blown encryption system and it should not be used in this capacity.

In addition, attention should be paid to the fact that the ability to perform search operations over encrypted databases leads to the complexity of systems and an increase in the amount of memory required and query execution time. At that, some searchable

encryption schemes when performing certain queries do not provide sufficient data confidentiality. That is, with long-term observation, an attacker can obtain a significant part of the information about sensitive data. All this testifies to the openness of the secure search problem and the need for further research in this direction to ensure secure work with remote databases and data storages.

3. Main Security Aspects

In the proposed solution, as in most active research in the field of secure search technology, we consider a curious database administrator (DBA) or other external attacker with full access to the data stored on the DBMS server as the main threat to database security while recognizing that such attackers can be either semi-honest or malicious.

In our solution, we use a searchable encryption scheme that allows the owner of the secret key (more precisely, keys) to read and write data, creating searchable encrypted data, and trapdoors (providing the so-called architecture (S/S)—single writer/single reader [6]). In doing so, we extend this scheme to the so-called multiwriter/multireader (M/M) architecture layer by distributing a secret key to allow multiple users to perform write and search over an encrypted database. Moreover, we are taking certain measures to simplify the process of user revocation if necessary (not to cause big overhead).

The architecture of the proposed solution is shown in Figure 1. This is a typical of three-tier client-server architecture. This architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a specific level of abstraction [32]. This allows developers to create flexible and reusable applications. In addition, the three-tier architecture allows applications in any of the three tiers to be independently upgraded or replaced in response to changing requirements and technologies, thereby simplifying the implementation and maintaining such a system up to date, taking into account the development of security standards.

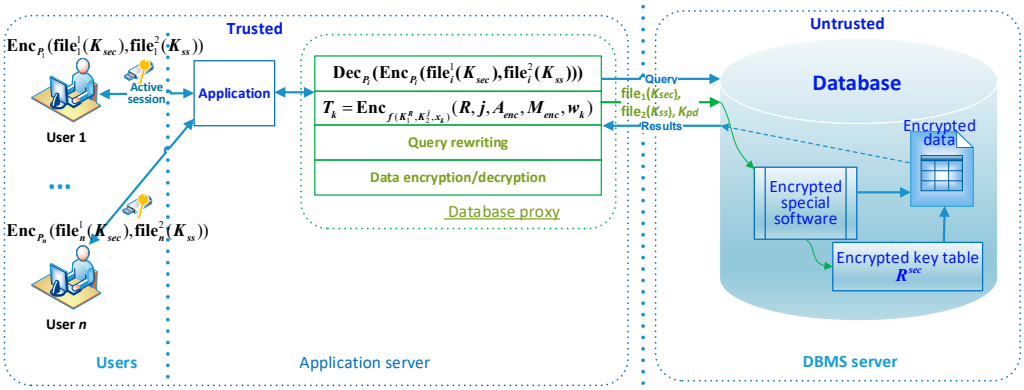

Figure 1. Architecture of the proposed solution.

In the solution under consideration, users and the application server are trusted, and the DBMS server is untrusted. Each user $i = 1, \ldots, n$ has some security token, in which container files (stegocontainers)—$\text{file}_i^1(K_{sec})$, $\text{file}_i^2(K_{ss})$ are stored in a compressed and encrypted form ($\text{file}_{i\_zip}^{enc} = \text{Enc}_{P_i}(\text{file}_i^1(K_{sec}), \text{file}_i^2(K_{ss}))$) using the secret key P_i. These container files contain decryption keys: K_{sec} for the special database table R^{sec} and K_{ss} for special software modules. R^{sec} and modules of special software are stored on the DBMS server. The container files are different for each legitimate user; moreover, where and how the corresponding keys K_{sec} and K_{ss} are contained in them, the users themselves do not know. These stegocontainers usually have different names. The database proxy leads them to names known to special software stored in encrypted form on the DBMS server. This is intentional in order to simplify the process of user revocation. Namely, do not distribute once again a new secret key (in this context, a file with stegocontainers that store the keys K_{sec} and K_{ss}) among the remaining users.

The transmitted compressed encrypted file ($file^{enc}_{i\_zip} = Enc_{P_i}(file^1_i(K_{sec}), file^2_i(K_{ss}))$) from the corresponding legitimate user $i = 1, \ldots, n$ arrives at the application server (database proxy). The database proxy (it is essentially a reverse proxy between the application and an untrusted DBMS server), knowing the secret key P_i, decrypts and decompresses the file $file^{enc}_{i\_zip}$ ($Dec_{P_i}(file^{enc}_{i\_zip}) = Dec_{P_i}(Enc_{P_i}(file^1_i(K_{sec}), file^2_i(K_{ss})))$) when processing the first user request related to the search for the required encrypted data in the active session. The decrypted container files $file^1_i(K_{sec})$, $file^2_i(K_{ss})$ are subsequently transferred over a secure channel to the DBMS server.

Table R^{sec} stores in encrypted form the keys for decrypting the encrypted sensitive data of certain attributes of the corresponding database tables. Persistent stored modules (PSMs) of special software are also stored on the DBMS server in encrypted form. They provide secure covert (without leaving a trace in the existing means of documenting completed queries) automatic extraction and decryption (without showing the plaintext) required to find encrypted data using keys stored in R^{sec}. At the same time, using the method based on the use of the potential of the modern blockchain model described in [33,34], the integrity of the key table and persistent stored system and user modules, as well as modules of special software developed within the framework of the proposed approach, is controlled. This increases the security of stored data and special software (increases protection against unauthorized modification, including through malware) with lower overhead costs (the amount of data stored for this and computing resources). Therefore, for example, in order to control the integrity and authenticity of a specific stored module (as is known, some unintentional and deliberate changes to certain software can potentially lead to data breaches) in the usual way, it would be necessary to perform hashing and digital signature procedures with storing the corresponding data for each of them. The use of the hash tree structure allows ensuring the integrity control not only of the specific PSM being checked, but also of all other stored database programs since this one data fragment is included in the general structure and changing at least one bit in it will entail a complete change in the value of the Merkle root. At the same time, it is appropriate to note that the developed software used in the proposed solution is considered secure (at least from the point of view of the basic principles of building secure software) [35]). As for the potential presence of possible vulnerabilities in modules of developed special software, which can be directly used by an attacker to implement security threats, this is a separate research topic, which is not discussed in this paper.

To encrypt/decrypt data of fields of various types of a tuple row of a certain table R of a database, a scheme is used based on the use of keys K^R_1, K^j_2, K^γ_3 similar to that described in [36,37], where K^R_1 is a unique 128-bit random value (secret key) generated by a cryptographically strong pseudo-random number generator (PRNG) for each table R, constant for all values that will be encrypted in this table; K^j_2 is a unique 128-bit random value (secret key) generated by a cryptographically strong PRNG for each attribute j of table R; and K^γ_3 is the value of the integer identifier of the primary key of the γ-th row of the table R. Secret (symmetric) keys K^R_1, K^j_2 are encrypted by one of the cryptographically strong algorithms and stored in the special database table R^{sec}. The values of these keys are never shown. They are not known to either the database administrator nor to any other user. Indirect access (through special software) to the secret keys is available to an authorized user with the appropriate privileges, who will present the corresponding keys (K_{sec},K_{ss}) in the active session. At the same time, the values of the keys (K_{sec}, K_{ss}) are also not shown anywhere in the clear.

As mentioned above, the special software itself is stored on the DBMS server in encrypted form. Decryption of this software and its activation is carried out automatically when executing the corresponding query related to the search for the required encrypted data in an active legitimate session. Such decryption can only be performed by a legitimate user for whom the database proxy server has the key K_{pd} to decrypt some module of the program (from the composition of special software), which extracts the key K_{ss} from the

stegocontainer file$_i^2(K_{ss})$, which is necessary to decrypt the remaining encrypted modules of the special software package.

The legitimate user in the proposed solution is granted timely and uninterrupted access to the database. Availability is supported by proven traditional control mechanisms and strategies that provide authorized access and acceptable levels of performance to quickly handle interrupts, to provide for redundancy, to maintain reliable backups, and to prevent data loss or destruction.

After the corresponding query is executed, the decrypted modules of special software are automatically deleted from the database server. At the same time, the text of these modules is not shown in clear text; it will also not be possible to trace it in the query history (through the corresponding logs).

Next, we will consider in more detail the functionality and features of the proposed solution, the implementation of which allows you to search for the required data in an encrypted database and manipulate them.

4. The Proposed Technique for Searching the Required Data

Preparing to execute a search query

In order to be able to generate and execute the corresponding search queries for the required data stored in encrypted form in the database on a third-party remote DBMS server, as well as queries for manipulating such data, you must first perform some operations. The main such operations are:

A. Operations related to the creation of security tokens for legitimate users:

- generation by a cryptographically strong PRNG of the secret (symmetric) keys K_{sec}—for encrypting/decrypting the table R^{sec} and K_{ss}—for encrypting/decrypting modules of the special software;
- formation of stegocontainers file$_i^1(K_{sec})$, file$_i^2(K_{ss})$ with their subsequent compression, encryption (file$_{i_zip}^{enc}$= Enc$_{P_i}$(file$_i^1(K_{sec})$, file$_i^2(K_{ss})$)) and entering this information on security tokens;
- issuance to users of the created security tokens with file$_{i_zip}^{enc}$.

B. Operations related to the generation of secret keys, subsequently used in an appropriate way by the proxy server and the database server:

- generation by a cryptographically strong PRNG of the secret keys K_1^R, K_2^j for each corresponding table R_α ($\alpha = 1, \ldots, A$) of the remote database and an attribute containing sensitive data that should be encrypted;
- generation by a cryptographically strong PRNG of key K_{pd} to encrypt/decrypt the module of the program for extracting key K_{ss} from file$_i^2(K_{ss})$.

C. Operations related to preparing for the secure operation of a remote database:

- creating a table R^{sec} in the database and writing to it the corresponding data encrypted with one of the cryptographically strong algorithms (for example, AES-256; K_{sec} is used as a key): secret keys K_1^R, K_2^j along with the names of the corresponding tables and attributes for which they were created;
- encryption using the keys K_{ss} and K_{pd} and recording in the database of special software modules.

Next, we will consider the actual algorithm for generating a search query implemented on a trusted application server (database proxy server). The main steps of the proposed algorithm for generating an SQL search query over encrypted databases are given below.

1. The i-th legitimate user during an active session receives:
   (a) compressed encrypted file file$_{i_zip}^{enc}$= Enc$_{P_i}$(file$_i^1(K_{sec})$, file$_i^2(K_{ss})$) with stegocontainers (at the beginning of the session);
   (b) the original SQL query with plaintext (Q).

2. Decrypting the file with stegocontainers $\text{file}_{i\_zip}^{enc}$ ($\text{Dec}_{P_i}(\text{file}_{i\_zip}^{enc})$), preparing them for transfer, and actually transferring the stegocontainers to the DBMS server. In the active session, when processing the first user query related to the search for the required encrypted data, the compressed file $\text{file}_{i\_zip}^{enc}$ is decrypted and the corresponding stegocontainers $\text{file}_i^1(K_{sec})$, $\text{file}_i^2(K_{ss})$ are extracted from it. The latter are renamed in order to bring their names in line with the names known to the special software and are transmitted to the DBMS server via a secure channel.
3. Parsing the original SQL query Q. Extraction from the user's original SQL query of the keywords w_k, presented in clear text, in accordance with which it is required to search for the required data in the database on a third-party remote DBMS server.
4. Generating a random number x_k. For this, some PRNG is used: $x_k = random\_PRNG$.
5. Formation of trapdoor T_k, on the basis of which the search will be carried out over the encrypted database. To do this, the corresponding search keyword w_k is encrypted using one of the cryptographically strong algorithms. At that, we use the same scheme that is used to encrypt the data of row γ for the corresponding column j of some table R of the database. Namely, an encryption scheme based on the use of some encryption algorithm (A_{enc}), encryption mode (M_{enc}) and secret key (K_T) selected from the list, as a function of $K_T = f(K_1^R, K_2^j, x_k)$. That is $T_k = \text{Enc}_{f(K_1^R, K_2^j, x_k)}(R, j, A_{enc}, M_{enc}, w_k)$. The proposed solution has no fundamental restrictions on the encryption algorithm that will be used in it. It can be any, either existing or newly developed; the only requirement is the presence of its implementation at the time of using the solution. For example, for definiteness, let these be the DES, Triple DES, AES algorithms with various modifications: $A_{enc} = \{'\text{DES}', '\text{3DES}', '\text{3DES\_2KEY}', '\text{AES128}', '\text{AES192}', '\text{AES256}'\}$). There are also no fundamental restrictions on encryption modes. Example modes: $M_{enc} = \{'\text{CBC}', '\text{ECB}', '\text{CFB}', '\text{OFB}'\}$.

It should be noted that the encrypted trapdoor values for the same keyword w_k may differ (including due to the arbitrary choice of the number x_k). That is, trapdoors are non-deterministic, which makes it difficult to leak the search pattern.

6. Stages 4 and 5 are repeated for all w_k ($w_k \in W$). However, instead of generating a random number x_k each time, an increment of its initially generated value is allowed: $x_{k+1} = x_k + 1$, $k = 1, \ldots, |W|$, where $|W|$ is the cardinality of the set of keywords w_k.
7. Formation of the final query (Q_M) to the database that does not disclose sensitive data.
8. Launching some procedure stored on the DBMS server (P_{DS}), which determines the session parameters based on the data extracted from the transferred stegocontainers $\text{file}_i^1(K_{sec})$, $\text{file}_i^2(K_{ss})$, and key K_{pd}. These parameters are valid for a legitimate user only in his active session. They are used in the corresponding modules of special software that support the functionality of the proposed solution.
9. Transferring the modified user query to the DBMS server for execution.

The general scheme of the search query formation algorithm (Algorithm 1) is presented below.

The essence of this query is to determine what actions were performed using bank cards '4454102135347018' and '5167135104128196' for a certain time interval (for simplicity. This period is not explicitly indicated; for example, it is assumed that this is the all time report), where at_1 *is the* TABLE_1 (R) attribute of the remote database, the values of which are stored in encrypted form (in this case, this is the card number); *at_2* is an attribute of *TABLE_1* whose values are stored in plaintext (for example, an opcode); *id* is a unique row identifier (primary key—*PK*) of the *TABLE_1* table.

Then, in accordance with the above algorithm, the following actions will be performed.

After receiving the compressed encrypted file $\text{file}_{i\_zip}^{enc}$, decrypting it ($\text{Dec}_{P_i}(\text{file}_{i\_zip}^{enc})$) with extracting the corresponding stegocontainers, as well as receiving the original SQL query with plaintext, the latter is parsed on the DB proxy server.

Algorithm 1: Query Formation Algorithm (QFA)

Input: K_1^R, $K_2^j Rj$, A_{enc}, M_{enc}, P_i
Output: modified user query Q_M and its transfer for execution
1 : Read file$_{i_zip}^{enc}$= Enc$_{P_i}$(file$_i^1(K_{sec})$, file$_i^2(K_{ss})$)
2 : Decrypt file containing stegocontainers Dec$_{P_i}$(file$_{i_zip}^{enc}$)
3 : Extract and rename stegocontainers : file$_i^1(K_{sec})$, file$_i^2(K_{ss})$
4: Read original SQL (query with plaintext)—Q
5: Parsing the original query Q: extract $w_k \in W$
6 : Random number generation : $x_1 = random\_PRNG$
7 : $T_1 = \text{Enc}_{f(K_1^R, K_2^j, x_1)}(R, j, A_{enc}, M_{enc}, w_1)$
8 : **for** k $= 2$ **to** $|W|$
9 : $x_k = \begin{cases} random\_PRNG \\ x_{k-1} + 1 \end{cases}$ /* implementation dependent */
10 : $T_k = \text{Enc}_{f(K_1^R, K_2^j, x_k)}(R, j, A_{enc}, M_{enc}, w_k)$
11: **end for**
12 : Formation of the final query to the database Q_M.
13: Running a procedure that defines session parameters.
14: Transferring the modified user query to the DBMS server for execution.

Example 1. Let the following query come from a legitimate user:
select id, at_1, at_2 from TABLE_1
where at_1 in ('4454102135347018', '5167135104128196')
order by id;.

Keywords w_k are extracted first. In this case, such keywords are bank card numbers w_1 = '4454102135347018', w_2 = '5167135104128196'. After that, using the existing PRNG, a random number is generated. For example, suppose the PRNG generated a random number c = 160634721428732436748471615956417046369.

Then, trapdoors T_k are formed, on the basis of which a search will be carried out over the encrypted database. In the case under consideration, for w_1 = '4454102135347018' and x_1 = 160634721428732436748471615956417046369, the value of T_1 = Enc$_{f(K_1^R, K_2^j, x_1)}$ $(R, j, A_{enc}, M_{enc}, w_1)$ can be equal to '0512F7D59B92BF08EFAE3E0E488D1B67CC1AE5A57991C 2BE00B1F3 5C2AA9D310'.

As noted above, encrypted trapdoor values for the same w_1 keyword may differ. For example, when x'_1 = 24339107466813018440859597836919234881, the value of T_1' is '1085E6EBA12C50E2352BEA9D021A7C8931299614906DF199BD34D1E87D436673'. Both one and the other value of trapdoor are equivalent when searching. The value T_2 = '161349D2EA5D997C80F5E60FDDB7F6E17E206A483C19103E9A81680D1A92E71E' for w_2 = '5167135104128196' and x_2 = '335991485542368735771035274697529587778' is determined in a similar way.

After that, based on the initial query Q, the final query Q_M is formed. Namely, based on the preferences of the algorithms used and encryption modes (for definiteness, let these be 'AES128' $\in A_{enc}$ and 'CBC' $\in M_{enc}$), the visibility of the query representation, as well as the specific implementation of the database on the Oracle DBMS platform, the query is converted to the following form:

```
select id, at_1, at_2 from TABLE_1
where get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id)) in
(
(select get_decr_val('0512F7D59B92BF08EFAE3E0E488D1B67CC1AE5A57991C2BE00B1
F35C2AA9D310', 'TABLE_1', 'at_1', 'AES128', 'CBC', 160634721428732436748471
615956417046369) from dual),
(select get_decr_val('161349D2EA5D997C80F5E60FDDB7F6E17E206A483C19103E9A816
80D1A92E71E', 'TABLE_1', 'at_1', 'AES128', 'CBC', 335991485542368735771035274
4697529587778) from dual)
```

```
)
order by id;.
```

Distinguishing features of the converted query are shown in bold type, where *get_decr_val* is some function ($F_{decr}(\alpha) = \mathrm{Decr}_{f(K_1^R, K_2^j, K_\alpha)}(R, j, A_{enc}, M_{enc}, \alpha)$) of special software, where α is either the encrypted value of $E_{K_3^\gamma}$ stored in the γ-th row (*id*) of the j-th column (*at_1*) of some table R in the database (in this example it is *TABLE_1*), or the corresponding value of T_k. This function is stored on the DBMS server in encrypted form. It implements a secure covert (without leaving traces in the available means of documenting completed queries) decryption of the data required for search (without displaying these data in the clear). K_α is either a random number x_k or a unique row identifier (*id*) of some table R in the database. As can be seen from the above query Q_M, sensitive data are not presented anywhere in the clear.

After the query Q_M is formed, a command is issued to execute some procedure P_{DS}, which determines the session parameters necessary for the legitimate user to work further in the active session. After that, the modified query Q_M itself is transferred to the DBMS server for execution.

Features of the query implementation on the DBMS server

As is known [26], in the fight against the threat caused by the illegal actions of a curious DBA or other external attacker with full access to data, there is a problem that lies in the contradiction between minimizing the amount of confidential information disclosed to the DBMS server and the ability to efficiently execute various queries. At that, existing approaches to working with encrypted data are either too slow or do not provide adequate confidentiality. On the other hand, it is also known that encrypting the data with some efficient cryptosystem would prevent many SQL queries from being executed by the DBMS server. In this case, the only practical solution would be to give the DBMS server access to the decryption key. However, if certain efforts are not made to prevent access to this key (by taking possession of the value of this key) by illegitimate users, and first of all, in this case, this applies to privileged users (such as a DBA or an external attacker with DBA rights), then this would allow the attacker to gain access to all data. Therefore, taking into account all of the above, a compromise solution was adopted in the proposed approach. Namely, to use an effective cryptosystem to encrypt sensitive data, but to provide access to these data only to a legitimate user and only one that will provide the database server (indirectly through the database proxy) during an active session with an encrypted file with steganocontainers $\mathrm{file}_{i\_zip}^{enc}$ containing the corresponding secret keys K_{sec}, K_{ss} of the user. The encrypted file $\mathrm{file}_{i\_zip}^{enc}$ is decrypted and decompressed by the DB proxy server, and the corresponding container files $\mathrm{file}_i^1(K_{sec})$, $\mathrm{file}_i^2(K_{ss})$ are transmitted over a secure channel to the DBMS server. On the DBMS server, the corresponding keys are extracted automatically by special software, the main modules of which are also encrypted. These modules are decrypted only if the K_{pd} and K_{ss} keys are available, which are associated with a legitimate user. The activity of the K_{sec}, K_{ss} keys is limited by the duration of the session. Their values are assigned to the corresponding attribute of the so-called application context (application context—a set of pairs: "attribute name—value"), more precisely, the database session-based application context, which is initialized locally [38]. As is known [39], such values are valid within the active session, and they are not visible to another session.

Thus, in our solution, the encryption keys are associated with the user's keys/passwords, which allows decryption of the corresponding data elements only by legitimate users who present the corresponding keys in an active session. Therefore, a curious DBA and an attacker with DBA rights, not knowing K_{sec}, K_{ss}, K_{pd} and not being able to disclose their values, and, therefore, not being able to extract the corresponding keys (K_1^R, K_2^j) from the table R^{sec}, will not be able to decrypt the main modules of special software and encrypted data, which is what we were striving for in this case.

In general, our solution allows the DBMS server to execute SQL queries over encrypted data in much the same way as if it were executing the same queries over open data. At the same time, existing applications do not need to be modified. The query itself is modified on

the proxy server. The database server within the active session of a legitimate user executes a modified query, automatically extracting the required encrypted data. For automatic extraction of encrypted data, encrypted stored modules of special software are used. The latter are automatically decrypted by some open modules stored on the database server using file$_i^2(K_{ss})$ and K_{pd} received from the proxy server.

We will consider the features of query processing on the DBMS server using the example of the implementation of the proposed solution for the Oracle DBMS. In our solution, we use the potential of the RLS (Row Level Security) technology [40], supplementing it with the capabilities of the application context [39,41,42]. We also use the capabilities of fine-grained audit (FGA) technology [39,41,42], and first of all, the ability to mimic a SELECT trigger (by automatically executing a stored procedure when a user issues SELECT).

The main stages of the proposed solution are:

1. Execution of the P_{DS} procedure stored in clear text on the DBMS server.

With the help of this procedure, using the key K_{pd} received from the DB proxy server, one of the modules of special software is decrypted to extract the key K_{ss} from file$_i^2(K_{ss})$. After decrypting the corresponding module, the latter is compiled and launched for execution. As a result of the operation of the decrypted module, the key K_{ss} is extracted from file$_i^2(K_{ss})$, the value of which is assigned to one of the attributes of the current user session—the application context attribute—*atrc1* (the so-called local initialization of the application context, for user session). This value, as noted above, is valid within this session and is not available from other sessions. After determining the appropriate attribute of the application context, the decrypted module of the special software for extracting the key K_{ss} from file$_i^2(K_{ss})$ is deleted.

Moreover, using the P_{DS} procedure, which uses the set value of the *atrc1* application context attribute during its execution, the next module of special software is decrypted, followed by its compilation and execution to extract the K_{sec} key from file$_i^1(K_{sec})$. The resulting value of the K_{sec} key is assigned to another attribute (*atrc2*) of the current user session (local initialization of the application context). This value will later (when executing the corresponding queries) be used to enable the legitimate user to automatically extract (using decrypted special software) the appropriate keys from the encrypted table R^{sec} to decrypt sensitive data stored in the corresponding attributes of the remote database tables. This actual application context value is also only valid within this session and is not available from other sessions.

2. To execute the corresponding request Q_M in our solution, it is proposed to use the capabilities of technologies: RLS (with the implementation of the application context) and FGA (namely, the ability to mimic a SELECT trigger). For this purpose, within the framework of the above technologies, an additional stored procedure and function were implemented. These persistent stored modules run automatically under the appropriate conditions specified in the appropriate security policies. It is these modules that are involved in decrypting the remaining stored modules of special software, compiling, running for execution, as well as automatically deleting the latter after executing the corresponding data search operator in a cryptographically protected database.

Activated stored modules of special software provide secure hidden (without leaving traces in the available documentation tools of the DBMS server) automatic extraction and decryption (without showing in the clear) of encrypted data using keys stored in R^{sec}. That is, the automatically obtained intermediate results of decryption $F_{decr}(E_{K_3^\gamma}) = \mathrm{Decr}_{f(K_1^R, K_2^j, K_3^\gamma)}(R, j, A_{enc}, M_{enc}, E_{K_3^\gamma})$, necessary to search for the required data among the encrypted ones ($E_{K_3^\gamma}$ is the encrypted value stored in the γ-th line of the j-th column of some table R of the database), are not disclosed to users, including privileged ones.

3. The DBMS server returns the result of the query Q_M to the database proxy server in encrypted form.

4. The database proxy server decrypts the received encrypted data and returns to the application the result of the original query (Q) of the user.

The general scheme of the query execution algorithm (Algorithm 2) is presented below.

Algorithm 2: Query Execution Algorithm (QEA)

Input: Q_M, $\text{file}_i^1(K_{sec})$, $\text{file}_i^2(K_{ss})$, K_{pd}
Output$Q$.
1 : Executing a stored procedure P_{DS}.
1.1 : Decryption $\left(\text{using } K_{pd}\right)$ of one of the special software modules.
1.2: Compiling and running the decrypted module of special software.
1.3 : Extracting the key K_{ss} from the stegocontainer $\text{file}_i^2(K_{ss})$.
1.4 : The application context attribute (atrc) is set to a value of K_{ss}.
1.5: Using the value of the *atrc1* application context attribute, the next module of special software is decrypted
1.6 : Compiling and running some decrypted modules to extract K_{sec}.
1.7 : Extracting the key K_{sec} from the stegocontainer $\text{file}_i^1(K_{sec})$.
1.8 : The atrc application context attribute is set to K_{sec}.
2 : Executing a search query (Q_M) over encrypted data.
2.1: Decryption of the remaining stored modules of special software, their compilation and execution.
2.2: The pre-decrypted software provides secure, hidden and automatic extraction and decryption (without displaying it in the clear) of the encrypted data required for searches.
2.3 : Executing a search query Q_M.
2.4: Removing decrypted stored modules of special software.
3 : The DBMS server returns the result of the query Q_M to the database proxy server in encrypted form.
4: The database proxy decrypts the received encrypted data and returns to the application the result of the original user query Q.

Thus, in order for the initial request to be executed only by a legitimate user, the latter in an active session must present the corresponding keys stored on the security tokens he has. The corresponding data are decrypted using the legitimate user's key chain, including the K_{pd} key stored on the database proxy server.

Examples of data search queries (SELECT statement).

Pattern matching. Search for activities that were carried out using bank cards whose numbers start with '516'.

A search query for appropriate activities with such bank cards may look like this:

```
select id, at_1, at_2 from TABLE_1
where get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id))
Like
(select get_decr_val('82C5218A798679BB9C5FC82B135D66CD', 'TABLE_1', 'at_1',
'AES128', 'CBC', 27207686983138459463931363559587691027 7)
from dual)
order by id;
```

where the value '82C5218A798679BB9C5FC82B135D66CD' in the *get_decr_val* function is the encrypted value $w_l =' 516\%'$.

Range search. Search for activities that were carried out using bank cards whose numbers range from '378282246310005' to '4454102135347018'.

Possible query appearance:

```
select id, at_1, at_2 from TABLE_1
where get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id))
BETWEEN
(select get_decr_val('785855FCE8CAC88C9876FB14834B1E4F', 'TABLE_1', 'at_1',
'AES128', 'CBC', 41743711899582577735039044513724643184)
from dual)
```

```
AND
(select get_decr_val('CE11FA4F985C39A2F1F207FA2D687CEC85B17BE048FA7AD17
68CAE5105563520', 'TABLE_1', 'at_1', 'AES128', 'CBC', 12345999999) from dual)
order by id;
```

Search by inequality. Search for actions that were carried out using bank cards whose numbers are greater than '378282246310005'.

Possible query appearance:

```
select id, at_1, at_2 from TABLE_1
where get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id)) >
(select get_decr_val('785855FCE8CAC88C9876FB14834B1E4F', 'TABLE_1',
'at_1', 'AES128', 'CBC', 417437118995825777350390445137246431 84)
from dual)
order by id;
```

Above, examples of the implementation of search queries for the required data in the proposed solution using the SELECT statement were presented. However, as is known, the data to be protected must first be entered into the database in a safe way. In the future, actions aimed at modifying and deleting them must also be safe from the point of view of data confidentiality. Therefore, when writing new sensitive data to the database in the proposed solution, the original user query is also transformed by the database proxy. That is, the sensitive data specified in the original query in clear text are encrypted on the proxy server with one of the cryptographically strong algorithms. Encryption of data is a key component in implementing the principle of multi-level protection. Below, in pseudocode (using the extended Backus–Naur form—EBNF [43]), the corresponding SQL statement is presented, which is transmitted to the DBMS server:

```
INSERT INTO R [(col_1, col_2, col_3,...)]
VALUES (dataValuePlaintext_1, dataValuePlaintext_2,
dataValueCrypted_3,...);
```

where dataValuePlaintext_1, dataValuePlaintext_2 are open data written to the database; dataValueCrypted_3—encrypted sensitive data written ($E_{K_3^\gamma} = \mathrm{Enc}_{f(K_1^R, K_2^j, K_3^\gamma)}(R, j, A_{enc}, M_{enc}, F_{decr}(T_l))$, $T_l = \mathrm{Enc}_{f(K_1^R, K_2^j, x_l)}(R, j, A_{enc}, M_{enc}, w_l)$) to the database. Thus, open data are written to the database unchanged, and sensitive data of the original query is encrypted by one of the cryptographically strong algorithms.

With a specific implementation of the database, for example, on the Oracle DBMS platform, this query may look like this:

```
INSERT INTO TABLE_1 (at_1, at_2, at_3)
VALUES (TABLE_1_at_1_SEQ.NEXTVAL, 'XXX', get_encr_val(get_decr_val('67984BE5
2836AAE16C58336213FB13FF', 'TABLE_1', 'at_3', 'AES128', 'CBC', 7302095866069
897001916527992404524 9079),
'TABLE_1', 'at_3', 'AES128', 'CBC', TABLE_1_at_1_SEQ.NEXTVAL));
```

At the same time, the new confidential value (in this example, it is the bank card number '373802460082124'), which is written to the database, is not shown to anyone in clear text. Based on it, a trapdoor T_l ('67984BE52836AAE16C58336213FB13FF') is preliminarily formed on an arbitrary key x_l (730209586606989700191652799240452490 79) using the appropriate algorithm (in this case, AES128), using a special procedure (from the software of the proposed solution).

Queries to update and delete sensitive encrypted data in the proposed solution are implemented as follows.

When you update sensitive encrypted data, the database proxy will convert the user's query to the form shown below in pseudocode:

```
UPDATE R
SET at_1 = E_{K_3^γ}
WHERE F_decr(E_{K_3^γ}) = F_decr(T_k));
```

where $E_{K_3^\gamma} = \mathrm{Enc}_{f(K_1^R, K_2^j, K_3^\gamma)}(R, j, A_{enc}, M_{enc}, F_{decr}(T_l))$ is encrypted sensitive data that are written to the database, $T_l = \mathrm{Enc}_{f(K_1^R, K_2^j, x_l)}(R, j, A_{enc}, M_{enc}, w_l)$; $F_{decr}(T_k) = \mathrm{Decr}_{f(K_1^R, K_2^j, x_k)}(R, j, A_{enc}, M_{enc}, T_k)$. Moreover, the values of $F_{decr}(E_{K_3^\gamma})$ and $F_{decr}(T_k)$ are not disclosed to users, including privileged ones. $F_{decr}(E_{K_3^\gamma})$ is the open value corresponding to the encrypted value stored in the γ-th line, j-th column of table R. $F_{decr}(T_k)$ is w_k. The trapdoor value T_k used for comparison in this query usually does not exist in the actual table (thus the search pattern is not disclosed).

When implementing a database on a specific platform, this query may look like this:

```
UPDATE Table_1
SET at_1 = get_encr_val(get_decr_val('1B1541B10DE424CA84F544CE8E70365B',
'TABLE_1', 'at_1', 'AES128', 'CBC', 2365092804011759428330446784055990986 73),
'TABLE_1', 'at_1', 'AES128', 'CBC', id)
WHERE get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id) =
get_decr_val('9EAD4C452EC7A53AD1EA204B2FB0AC75A1ABDD8988BDE22D0E1C58DACC
23360F', 'TABLE_1', 'at_1', 'AES128', 'CBC', 290128337798191690105369894 4
63767116351);
```

It should be noted that in this implementation:

- $R \rightarrow$ 'TABLE_1';
- $F_{decr}(E_{K_3^\gamma}) \rightarrow$ get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id);
- $F_{decr}(T_k) \rightarrow$ get_decr_val('9EAD4C452EC7A53AD1EA204B2FB0AC75A1AB DD8988BD E22D0E1C58DACC23360F', 'TABLE_1', 'at_1', 'AES128', 'CBC', 2901283377981916901053 69894463767116351);
- $T_k = \mathrm{Enc}_{f(K_1^R, K_2^j, x_k)}(R, j, A_{enc}, M_{enc}, w_k)$ = '9EAD4C452EC7A53AD1EA204B2FB0AC75A1 ABDD8988BDE22D0E1C58DACC23360F' is the trapdoor value obtained by pre-encrypting the keyword w_k on an arbitrary key x_k;
- $E_{K_3^\gamma} = \mathrm{Enc}_{f(K_1^R, K_2^j, K_3^\gamma)}(R, j, A_{enc}, M_{enc}, F_{decr}(T_l)) \rightarrow$ get_encr_val((SELECT get_decr_val ('1B1541B10DE424CA84F544CE8E70365B', 'TABLE_1', 'at_1', 'AES128', 'CBC', 23650928 040117594283304467840 5599098673) from dual), 'TABLE_1', 'at_1', 'AES128', 'CBC', id), *get_encr_val* is some function of the special software;
- $T_l = \mathrm{Enc}_{f(K_1^R, K_2^j, x_l)}(R, j, A_{enc}, M_{enc}, w_l)$ = '1B1541B10DE424CA84F544CE8E70365B' is the trapdoor value obtained by pre-encrypting the keyword w_l on an arbitrary key x_l;
- $w_k = w_l$;
- T_k and T_l used for searching are not stored in the database.

If it is necessary to delete the required sensitive encrypted data, the database proxy server transforms the user's query to the form shown below in pseudocode:

```
DELETE FROM R
WHERE F_decr(E_K3^γ) = F_decr(T_k));.
```

When implemented, this query may look like this:

```
DELETE FROM Table_1
WHERE get_decr_val(at_1, 'TABLE_1', 'at_1', 'AES128', 'CBC', id) =
get_decr_val('9EAD4C452EC7A53AD1EA204B2FB0AC75A1ABDD8988BDE22D0E1C58DACC233
60F', 'TABLE_1', 'at_1', 'AES128', 'CBC', 290128337798191690105369894463767
116351);
```

5. Performance Evaluation

We tested our prototype on a single machine that simulates both a proxy and a database server. Machine hardware resources: CPU—Intel(R) Core(TM) i5-8300H CPU 2.30 GHz 2.30 GHz; Cores—4; Logical processors—8; RAM—8 GB; SSD—128 GB; HDD—1000 GB. The database is implemented on the Oracle 12.2 c DBMS platform installed on the Windows 10 (x64) operating system.

Figures 2–6 present (in various forms) the results of a comparative analysis of the average times (in seconds—T) of the search for the required unencrypted (plaintext) and

encrypted data with their full output for display depending on the number of obtained values (lines—N) when executing the corresponding SELECT query. Namely, queries that satisfy the main types of search conditions: comparison (equality, inequality), range, set membership (IN), pattern matching (Like). Two almost identical tables were used to obtain estimates of the corresponding time values. One table stored unencrypted data, and the other stored the same data in encrypted form. At the same time, the first table, unlike the second one (for encrypted data), was not associated with security policies (RLS and FGA). Figure 7 shows separately the average data search and decryption times (with their full output for display) for the case of encrypted data stored in the corresponding database table, depending on the number of obtained relevant data rows. The data search was performed in tables with more than 200,000 rows of records.

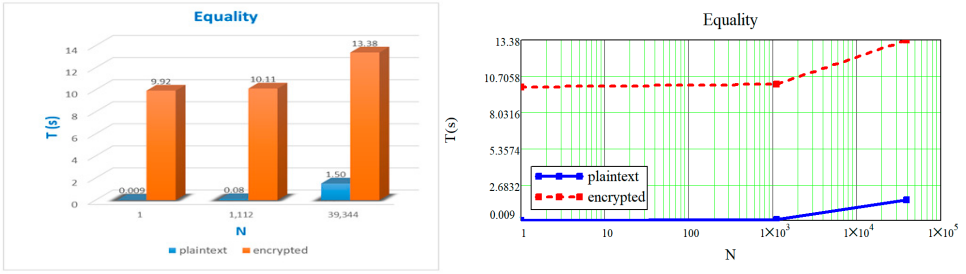

Figure 2. Average values of the search and data output times under the condition of equality.

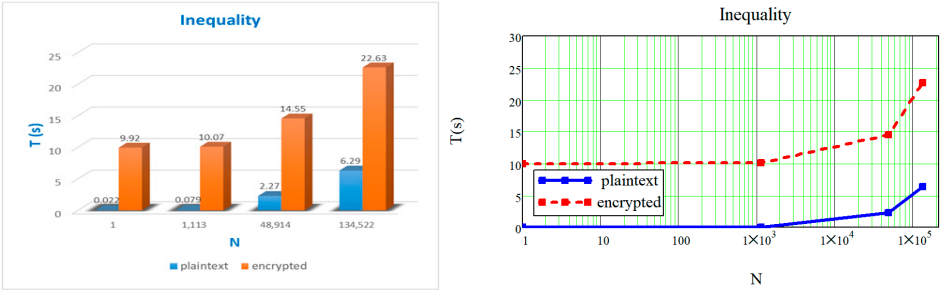

Figure 3. Average values of the search and data output times under the condition of inequality.

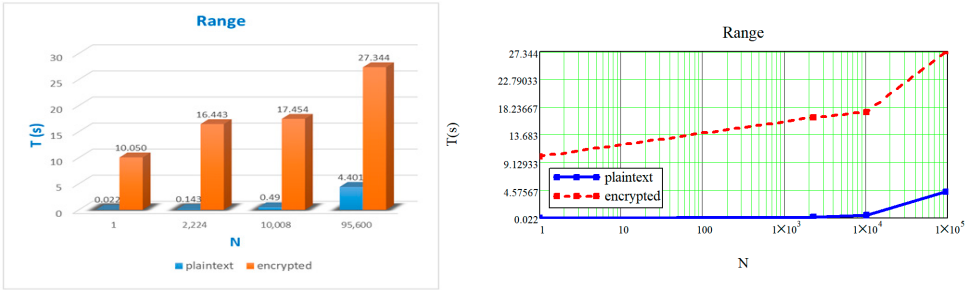

Figure 4. Average values of the search and data output times under the condition of the specified range.

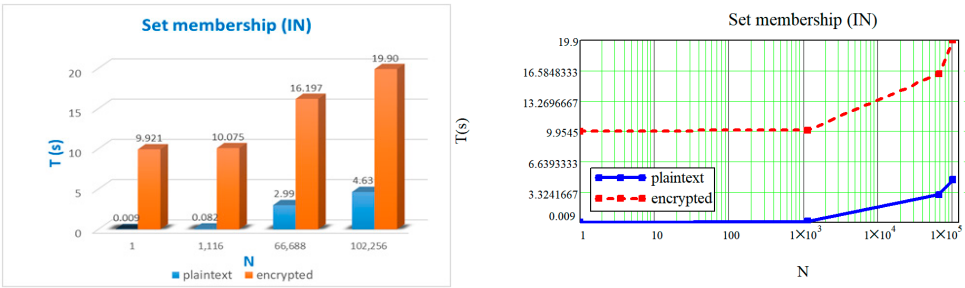

Figure 5. Average values of the search and data output times under the condition of the membership in the set.

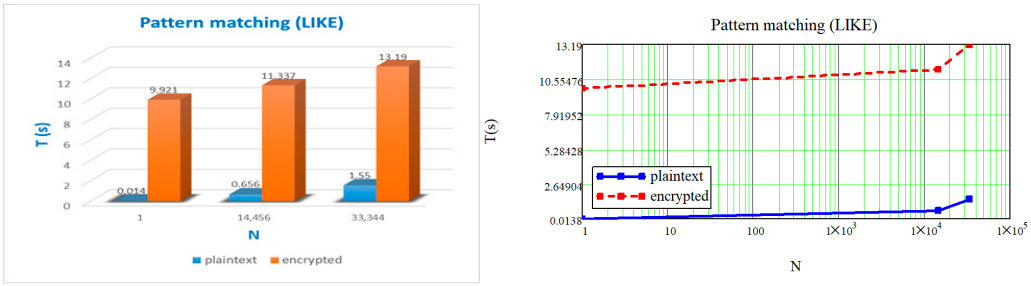

Figure 6. Average values of the search and data output times under the condition of pattern matching.

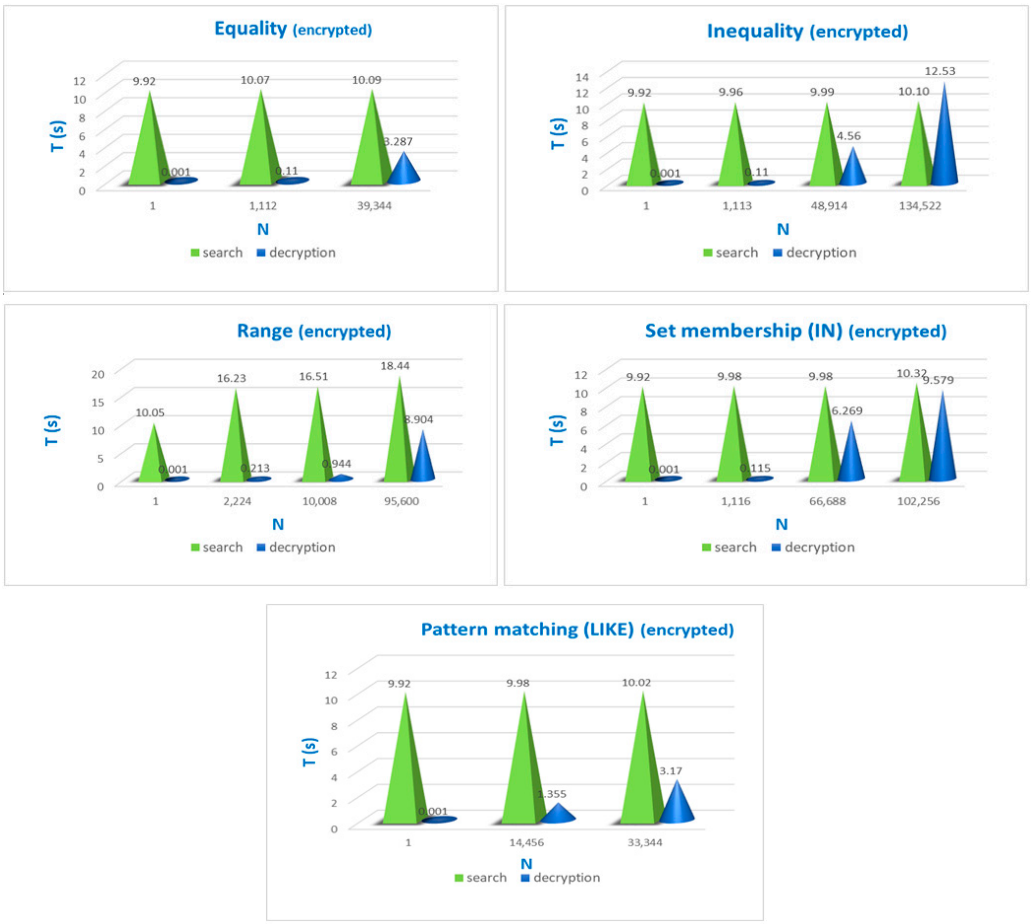

Figure 7. Average search times (for various conditions) and data decryption (with their full output for display) for encrypted data.

The presented results of the analysis indicate a reasonable overhead in the proposed solution. The increase in the corresponding search and decryption times, at first glance, may seem significant relative to the search time for data in unencrypted databases, as well as the search and decryption time in some existing searchable encryption schemes. However, in our opinion, this is an adequate price to pay for the security of the data stored by the service provider (in the context of not allowing anything but the access pattern to be leaked), query expressiveness (in the context of the variety of supported search queries), and the usability of standard SQL (same as in the unencrypted database). Since It is known [6] that securIty is never free, there is always a trade-off between security on the one hand and efficiency and expressiveness of queries on the other. Therefore, we once again focus on what was said at the beginning of the paper, namely, data owners and users must understand how suitable one or another option for protecting database systems is for their various applications.

Are they willing to accept longer time delays in response to their database queries as a corresponding price to pay for security, and what compromises are acceptable to them?

6. Conclusions

This paper presents a solution that provides a strong level of confidentiality when searching, inserting, modifying, and deleting the required sensitive data in a remote database whose data are encrypted. We have proposed a SQL query processing technique that allows the DBMS server to perform search functions over encrypted data in the same way as in an unencrypted database. This is achieved through the organization of automatic decryption by specially developed secure software of the appropriate data required for search, without the possibility of viewing these data itself. Decryption of the special software itself and its activation is also carried out automatically when accessing encrypted data in the corresponding query in an active legitimate session. Moreover, only a legitimate user who presented the corresponding key at the beginning of the session can perform such decryption. After the corresponding query is made, the decrypted special software is deleted automatically. The DBMS server cannot decrypt stored data encrypted with a strong cryptographic algorithm if these data were not requested by the application. At the same time, we guarantee the integrity of the used open stored procedures and special tables that store encrypted modules of special software and decryption keys, the relevance and completeness of the results returned to the application.

Our technique involves separate processing of an SQL query on the side of the DBMS server and the application server (database proxy). At that, legitimate users are granted timely and uninterrupted access to the database. We offer a basis for implementing our solution on the Oracle DBMS server side, using both unmodified DBMS software and our own developed persistent stored modules.

The results of the analysis of the feasibility and effectiveness of our approach, based on the evaluation of the performance of the proposed solution, show that proper security and privacy, despite the existing threats of data leakage from service providers, can be achieved with reasonable overhead.

In the future, we plan to conduct an in-depth performance evaluation of the proposed solution, including a comparison with existing implementations, to show its practicality in various real-life situations. In addition, it is planned to study the proposed solution in more depth from the point of view of its security, namely, vulnerability to inference leakage due to access and search patterns, to which almost any implementation of an SE scheme is susceptible to a greater or lesser degree, and the security of the special software used itself.

Author Contributions: Conceptualization, V.Y.; methodology, V.Y. and V.V.; software, V.Y., V.V. and R.K.; formal analysis, M.K. and R.K.; investigation, V.Y., M.Y., V.V. and R.S.; writing—original draft preparation, V.Y., M.Y. and R.S.; writing—review and editing, V.Y., M.K. and M.Y.; funding acquisition, M.K. and R.S. All authors have read and agreed to the published version of the manuscript.

Funding: The research work reported in this paper was, in part, supported by the National Centre for Research and Development, Poland, under grant no. POIR.04.01.04-00-0048/20.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not available.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Abadi, D.; Ailamaki, A.; Andersen, D.; Bailis, P.; Balazinska, M.; Bernstein, P.; Boncz, P.; Chaudhuri, S.; Cheung, A.; Doan, A.; et al. The Seattle Report on Database Research. *ACM Sigmod Rec.* **2020**, *48*, 44–53. [CrossRef]
2. Fuller, B.; Varia, M.; Yerukhimovich, A.; Shen, E.; Hamlin, A.; Gadepally, V.; Shay, R.; Mitchell, J.D.; Cunningham, R.K. SoK: Cryptographically protected database search. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 172–191. [CrossRef]

3. General Data Protection Regulation GDPR. Available online: https://gdpr-info.eu/ (accessed on 2 August 2023).
4. Payment Card Industry (PCI) Data Security Standard. Requirements and Testing Procedures Version 4.0. 2022. Available online: https://www.pcisecuritystandards.org/documents/PCI-DSS-v4_0.pdf (accessed on 2 August 2023).
5. Atchinson, B.K.; Fox, D.M. From the field: The politics of the health insurance portability and accountability act. *Health Aff.* **1997**, *16*, 146–150. [CrossRef] [PubMed]
6. Bösch, C.; Hartel, P.; Jonker, W.; Peter, A. A survey of provably secure searchable encryption. *ACM Comput. Surv. (CSUR)* **2014**, *47*, 1–51. [CrossRef]
7. Yesin, V.; Vilihura, V. Research on the main methods and schemes of encryption with search capability. *Radiotekhnika* **2022**, *2*, 138–155. [CrossRef]
8. Yesin, V.; Vilihura, V. Researching basic searchable encryption schemes in databases that support SQL. *Radiotekhnika* **2022**, *3*, 53–74. [CrossRef]
9. Azraoui, M.; Önen, M.; Molva, R. Framework for Searchable Encryption with SQL Databases. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018), Madeira, Portugal, 19–21 March 2018; pp. 57–67.
10. Ramasamy, R.; Vivek, S.S.; George, P.; Kshatriya, B.S.R. Dynamic verifiable encrypted keyword search using bitmap index and homomorphic MAC. In Proceedings of the 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), New York, NY, USA, 26–28 June 2017; pp. 357–362. [CrossRef]
11. Kamara, S. Encrypted search. *XRDS* **2015**, *21*, 30–34. [CrossRef]
12. Andress, J. *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*, 2nd ed.; Syngress: Waltham, MA, USA, 2014.
13. Liu, C.; Zhu, L.; Wang, M.; Tan, Y.A. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.* **2014**, *265*, 176–188. [CrossRef]
14. Oya, S.; Kerschbaum, F. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 127–142.
15. Grubbs, P.; McPherson, R.; Naveed, M.; Ristenpart, T.; Shmatikov, V. Breaking web applications built on top of encrypted data. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1353–1364. [CrossRef]
16. Gentry, C. Computing arbitrary functions of encrypted data. *Commun. ACM* **2010**, *53*, 97–105. [CrossRef]
17. van Dijk, M.; Gentry, C.; Halevi, S.; Vaikuntanatha, V. *Fully Homomorphic Encryption over the Integers*; Advances in Cryptology—EUROCRYPT 2010. EUROCRYPT 2010. Lecture Notes in Computer Science; Gilbert, H., Ed.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6110, pp. 24–43. [CrossRef]
18. Brakerski, Z.; Vaikuntanathan, V. *Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages*; Advances in Cryptology—CRYPTO 2011. CRYPTO 2011. Lecture Notes in Computer Science; Rogaway, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6841, pp. 505–524. [CrossRef]
19. Brakerski, Z.; Gentry, C.; Vaikuntanathan, V. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory* **2014**, *6*, 1–36. [CrossRef]
20. Garg, S.; Gentry, C.; Halevi, S.; Raykova, M.; Sahai, A.; Waters, B. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM J. Comput.* **2016**, *45*, 882–929. [CrossRef]
21. Boneh, D.; Sahai, A.; Waters, B. *Functional Encryption: Definitions and Challenges*; Theory of Cryptography. TCC 2011. Lecture Notes in Computer Science; Ishai, Y., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6597, pp. 253–273. [CrossRef]
22. Boneh, D.; Waters, B. *Conjunctive, Subset, and Range Queries on Encrypted Data*; Theory of Cryptography. TCC 2007. Lecture Notes in Computer Science; Vadhan, S.P., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4392, pp. 535–554. [CrossRef]
23. Katz, J.; Sahai, A.; Waters, B. *Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products*; Advances in Cryptology—EUROCRYPT 2008. EUROCRYPT 2008. Lecture Notes in Computer Science; Smart, N., Ed.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 4965, pp. 146–162. [CrossRef]
24. Boneh, D.; Franklin, M. *Identity-Based Encryption from the Weil Pairing*; Advances in Cryptology—CRYPTO 2001. CRYPTO 2001. Lecture Notes in Computer Science; Kilian, J., Ed.; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2139, pp. 213–229. [CrossRef]
25. Sahai, A.; Waters, B. *Fuzzy Identity-Based Encryption*; Advances in Cryptology—EUROCRYPT 2005. EUROCRYPT 2005. Lecture Notes in Computer Science; Cramer, R., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3494, pp. 457–473. [CrossRef]
26. Popa, R.A.; Redfield, C.M.; Zeldovich, N.; Balakrishnan, H. CryptDB: Protecting confidentiality with encrypted query processing. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, 23–26 October 2011; pp. 85–100. [CrossRef]
27. Pilyankevich, E.; Kornieiev, D.; Storozhuk, A. Proxy-Mediated Searchable Encryption in SQL Databases Using Blind Indexes. *Cryptol. Eprint Arch.* **2019**, 806.
28. Hacigümüş, H.; Iyer, B.; Li, C.; Mehrotra, S. Executing SQL over encrypted data in the database-service-provider model. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 4–6 June 2002; pp. 216–227. [CrossRef]

29. Curtmola, R.; Garay, J.; Kamara, S.; Ostrovsky, R. Searchable symmetric encryption: Improved definitions and efficient constructions. In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06), Association for Computing Machinery, Alexandria, VA, USA, 30 October–3 November 2006; pp. 79–88. [CrossRef]
30. CipherSweet. Available online: https://ciphersweet.paragonie.com/ (accessed on 2 August 2023).
31. McCarty, R.J. Methods and Systems for Transparent Data Encryption and Decryption. US Patent 7426,745 B2, 16 September 2008.
32. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Soft-Ware Architecture—Volume 1: A System of Patterns*; John Wiley & Sons Ltd.: Chichester, UK, 1996.
33. Yesin, V.; Karpinski, M.; Yesina, M.; Vilihura, V.; Warwas, K. Ensuring Data Integrity in Databases with the Universal Basis of Relations. *Appl. Sci.* **2021**, *11*, 8781. [CrossRef]
34. Yesin, V.I.; Yesina, M.V.; Vilihura, V.V. Monitoring the integrity and authenticity of stored database objects. *Telecommun. Radio Eng.* **2020**, *79*, 1029–1054. [CrossRef]
35. Viega, J.; McGraw, G.R. *Building Secure Software: How to Avoid Security Problems the Right Way*; Addison-Wesley: Reading, MA, USA, 2008.
36. Yesin, V.I.; Vilihura, V.V. Some approach to data masking as means to counter the inference threat. *Radiotekhnika* **2019**, *198*, 113–130. [CrossRef]
37. Yesin, V.; Karpinski, M.; Yesina, M.; Vilihura, V.; Warwas, K. Hiding the Source Code of Stored Database Programs. *Information* **2020**, *11*, 576. [CrossRef]
38. Using Application Contexts to Retrieve User Information. Available online: https://docs.oracle.com/en/database/oracle/oracle-database/21/dbseg/using-application-contexts-to-retrieve-user-information.html#GUID-51C9D5FA-6787-4F05-82EF-A5968BEDC5A0 (accessed on 2 August 2023).
39. Feuerstein, S.; Pribyl, B. *Oracle PL/SQL Programming*, 6th ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2014.
40. Cotner, C.; Miller, R.L. Row-Level Security in a Relational Database Management System. US Patent 9,870,483 B2, 16 January 2018.
41. Kyte, T. *Expert Oracle*; Apress: New York, NY, USA, 2005.
42. Nanda, A.; Feuerstein, S. *Oracle PL/SQL for DBAs*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2005.
43. *ISO/IEC 14977:1996*; Information Technology—Syntactic Metalanguage—Extended BNF. Available online: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:14977:ed-1:v1:en (accessed on 2 August 2023).