Advanced Research on Information Systems Security

ISSN: 2795-4609 | ISSN: 2795-4560
Print & Online

Database Decomposition to Satisfy the Least Privilege Principle in Healthcare

Fabrizio Baiardi [a*], Vincenzo Sammartino [b]

[a b]*Università di Pisa – Dipartimento di Informatica, Largo B. Pontecorvo 3, 56127 Pisa*
[a]*Email: f.baiardi@unipi.it*
[b]*Email: v.sammartino@studenti.unipi.it*

Abstract

The Multilevel Database Decomposition Framework is a cybersecurity strategy to enhance system robustness and minimize the impact of data breaches with a focus on healthcare systems. With respect to more conventional normalization methods, the framework prioritizes robustness against cyber threats over mere data redundancy reduction. The key strategy of the framework is the decomposition of a database into smaller databases to restrict user access and mitigate the impact of successful intrusions by satisfying the least privilege principle in a more complete way. For this purpose, each database the decomposition produces is uniquely associated with a set of users and the decomposition ensures that each user can access all and only the data his/her operations need. This limits the potential impact of threat agents impersonating users to the information a compromised user can access. To prevent the propagation of an intrusion across the databases it produces, the framework can apply alternative allocation strategies by distributing the databases to distinct virtual or physical entities according to the security requirement of the original application. This flexibility in allocation management ultimately reinforces defenses against evolving cyber threats and it is the main advantage of the deposition. As a counterpart of better robustness, some tables will be replicated across the databases the decomposition returns and updates of these tables should be properly replicated to prevent inconsistencies among copies of the same table in distinct databases. The paper includes a performance analysis to evaluate the overheads associated with the alternative allocations. This offers insights into the framework implementation and adaptability to distinct security needs and to evaluate the framework effectiveness for healthcare data systems.

Keywords: database decomposition; decomposition; least privilege principles; impact minimization.

Citation: V. Sammartino and F. Baiardi, “Database Decomposition to satisfy the Least Privilege Principle in Healthcare”, ARIS2-Journal, vol. 4, no. 1, pp. 47–69, Apr. 2024.

DOI: https://doi.org/10.56394/aris2.v4i1.43

----

* Corresponding author. Email address: f.baiardi@unipi.it

1 Introduction

In response to increasing threats to data privacy and security, the Multilevel Database Decomposition Framework (MDDF) is an innovative approach that fits the needs of the healthcare sector. The framework is designed to implement relational databases with high robustness as it fully satisfies the principles of least privilege and related concepts to effectively mitigate contemporary threats and safeguard sensitive healthcare information.

The MDDF key notion is the decomposition of a relational database to minimize access rights [11], [12] for each user so that the user can access all and only the data the user operations need. In the event of a successful intrusion, this strongly reduces the intrusion blast radius i.e. the data that may be leaked because a threat agent impersonating a legitimate user can only access a lower amount of information [13]. This fortifies overall data security.

To prevent the spread of intrusions across decomposed databases, the MDDF can apply alternative allocation strategies. While the simplest allocation maps all databases onto the same machine, the framework supports enhanced confinement and robustness by distributing databases across distinct containers or physical/virtual machines. The ability to choose the allocation according to the robustness of interest is a fundamental advantage of decomposition with respect to a solution that simply manages the user access rights on the resulting databases.

Given the nature of decomposition, the resulting databases may share tables or subsets of tables. Consequently, these tables are replicated across databases, and updates are replicated to maintain consistency among multiple copies of a table and to offer the same data view to distinct users. The synchronization of update operations, integral to the framework, introduces complexity and overhead, with the latter escalating proportionally to the robustness of the separation [14].

The paper unfolds as follows: Section 2 delves into related works, offering foundational insights into the principles of least privilege and other key concepts that underpin the proposed framework. Section 3 provides a comprehensive overview of the framework application, emphasizing the decomposition of databases to optimally adhere to the least privilege principle. Section 4 presents a detailed example of the application, while Section 5 reports performance figures of the overhead of synchronizing multiple copies of a table as a function of the adopted separation among databases.

2 State of the Art

The principle of least privilege (PoLP) suggests minimizing the access rights of each legal user of a system so that the user never owns some access rights he/she does not need. This has several implications for the management of access rights and prevents the adoption of those strategies based upon a hierarchical level of privileges. Systems satisfying the PoLP can minimize the impact of an intrusion where a threat agent succeeds in impersonating a legal user. Stateof-the-art strategies to improve database security adopt not only the PoLP but also advanced measures such as RBAC, encryption, tokenization, auditing, monitoring, dynamic data masking, and pseudonymization. This is the way to face the challenges posed by evolving cyber threats and the adoption of cloud-based solutions. This security scenario requires a holistic and adaptive approach to fortify databases against an array of security risks. We can resume the most popular security mechanisms that can be integrated with the proposed strategy to address the challenging cyber risk scenarios [7], [5] as follows.

Role-Based Access Control (RBAC) [8] complements PoLP by assigning access permissions based on predefined roles. This streamlines user privilege management, ensuring individuals have access only to resources essential for their specific organizational roles.

Encryption and **Tokenization** are crucial components in securing sensitive data within databases [6]. Encryption transforms data into unreadable formats, while tokenization replaces sensitive data with non-sensitive placeholders, reducing the risk of unauthorized access and data exposure.

Database Auditing and **Monitoring** are proactive measures for identifying and mitigating security threats. Real-time monitoring enhances situational awareness, enabling swift responses to suspicious activities and potential vulnerabilities.

Dynamic Data Masking (DDM) [2] conceals or obfuscates sensitive information dynamically, based on predefined security policies. This ensures that only authorized individuals can access and view sensitive data, adding an extra layer of protection.

Challenges in modern database security include the constantly evolving **cyber threat landscape** with challenges such as ransomware, advanced persistent threats (APTs), and insider threats [16]. Adaptive and comprehensive security measures are essential to counter these evolving challenges.

The widespread adoption of **cloud-based database** solutions introduces new security concerns [9]. Protecting data stored in the cloud requires specialized measures to guard against unauthorized access, data breaches, and compliance with data residency regulations.

Integration with DevOps Practices is critical, balancing continuous development and deployment with robust security protocols. Innovative solutions are required to prevent vulnerabilities introduced during rapid changes.

Pseudonymization, another vital facet of contemporary database security, involves the substitution of personally identifiable information (PII) with artificial identifiers or pseudonyms [15]. This technique adds another layer of privacy protection by making it challenging to directly associate sensitive data with specific individuals. Pseudonymization contributes to compliance with data protection regulations such as GDPR [3], as it allows organizations to leverage data for legitimate purposes without compromising individual privacy. The implementation of pseudonymization within database systems enhances data security by both reducing the impact of unauthorized access to personal information and limiting exposure in the event of a breach.

3 Multilevel Database Decomposition Framework

This chapter presents the Multilevel Database Decomposition Framework, MDDF, and it fully describes the decomposition process, its architecture, and the results of this framework in real world scenarios.

3.1 *General Approach*

MDDF aims to fully satisfy the **principle of least privilege** (PoLP) by decomposing a relational database shared among a set of users to minimize user access to the data involved in the operations each user can invoke. To this purpose, starting from a system where each user can access any information in the database, the framework produces a system with distinct databases and where each user is granted access to the smallest amount of data to implement the operations of interest. This is achieved by distributing the tables in the original database to several databases, each consisting of only a subset of those in the original one. Each resulting database includes all the tables to implement the operations of a class of users and if the operations do not access an attribute of a table, the attribute is dropped. Each user belongs to one class, and it is assigned access rights on the tables in one of the databases the decomposition returns. This strategy satisfies the PoLP as it assigns all and only the access rights the user needs. MDDF can be integrated with any of the various mechanisms discussed in the previous section as any of these mechanisms can be applied to each of the databases. The last step of the framework maps the databases it builds to distinct containers, virtual or physical machines to confine a successful intrusion to one of the databases.

This section will discuss the decomposition step by step to outline the underlying principles. To this end, it is essential to explain how to decompose a database and then move from the decomposition of a database to that of its tables. Then, the section highlights the synchronization problems generated by a decomposition where the same table is shared, i.e. it belongs to distinct databases. This requires that some operations on a table in one of the databases produced by the decomposition fire an automatic update of databases sharing the same table.

Definition and Purpose: Normal forms apply decomposition to simplify the understanding and management of the database schema. This helps to reduce redundancy, to improve maintainability, and to enhance the overall database performance.

The main differences between the proposed approach and normalization can be outlined as follows.

- **Normalization**: It is a technique for decomposing databases by organizing data into multiple related tables to minimize redundancy, remove insertion, update and deletion anomalies and improve data integrity. There are several normal forms, each with specific rules and guidelines for structured database design.

- **MDDF Decomposition**: The decomposition of a database, e.g. a set of tables, returns subsets of these tables. Each table is either an original table or a subset of the attributes, the columns, of an original table. The subsets of tables are not partitions because they can share some tables. Each resulting subset is stored separately, and it can be accessed and managed independently.

The two notions are not in conflict because even in the MDDF context, normalization in 3NF plays a key role in optimizing the overall performance as it reduces the amount of information shared among the databases that MDDF produces.

The goal of the MDDF decomposition is to satisfy the PoLP to minimize the data each user can access. It can result in synchronization issues when the databases it produces share information to be updated simultaneously to correctly implement a user operation. This synchronization can be implemented through triggers, stored procedures or other mechanisms that automatically spread changes among databases, as discussed in detail in Section 4.

The MDDF Decomposition: The MDDF decomposition returns a set of databases so that each user can access just the database that includes all and only the information in the original database its operations need. The decomposition is implemented through a sequence of steps that, given a database and a set of users can be described as follows:

1. **Traditional Normalization in 3NF**: First, the database of interest is transformed to satisfy the third normal form. Normalization eliminates data redundancy and minimizes synchronization costs.

2. **Identification of User Groups**: All users interacting with the database are identified together with their operations and the data each operation accesses. Users are then partitioned into groups where two users belong to the same group if they execute the same operations and hence access the same data. These user groups are subsets that are the equivalence classes of the relation R where two users satisfy R if they need to access the same data.

3. **Creation of Database Subsets**: For each group of users identified in the previous steps, a new database is created. The database consists of a subset of the original one. Each subset includes all and only the tables with the data one subset of users can access. If the users in the group do not need to access an attribute of a table, we remove the attribute from the table in the subset. A table or a proper subset of the table attributes may be shared among distinct subsets i.e. appears in distinct subsets and users in distinct subsets can read or update these attributes.
4. **Association of Groups and Subsets**: A bi-univocal mapping is established between subsets of users and the databases the decomposition returns. The mapping matches each subset of users with a distinct database subset and the other way around in a bijective association.
5. **Configuring Access Mechanisms**: Access mechanisms must be configured to enable each user to interact only with the database associated with its user group. Hence, the group the user belongs to determines the user access rights.

After applying these steps and according to the required level of robustness, the confinement, or separation level, must be defined for the new databases, i.e. for each subset of the original database. The required robustness determines the mapping of the databases in a range of solutions that go from mapping the databases to a set of containers on the same machine to mapping the databases to distinct physical machines. The mapping determines the amount of work a threat agent has to do in an intrusion that spreads across distinct databases. Table 1 lists alternative mappings and the confinement robustness each enables. It is worth noticing that the ability to choose an allocation strongly increases the confinement level to the one enabled by tuning the access rights of users to the databases.

3.2 *Detailed Description of the Steps*

We analyze each of the previous steps in more detail:

1. **Traditional Normalization in 3NF**

   Normalization is a process that structures data in a database to eliminate redundancy and dependency. The third Normal Form (3NF) is a widely used standard that reduces data redundancy. By adopting this normalization, we can ensure that the database is well structured, and that the data is stored consistently.

2. **Identification of User Groups**

   Identifying users is a preliminary step that identifies all users who interact with the database and analyzes the needs of the operations of each user. In this way, we create groups of users tailored to their needs. This allows access to sensitive information to be limited only to users who are entitled to operate on such information according to the *need-to-know principle*.[1].

3. **Creating databases/subsets**

   The creation of multiple databases, each a subset of the original one is the central step of the framework. It creates a distinct database for each user group, according to the operation the users in the group execute and the data these operations access. We consider the worst case, e.g. any table and any attribute a user may require ensuring that users have access to all and only the information they need. In general, some tables will be accessed by users in distinct groups, and

[1] The need-to-know principle states that an individual must only have access to data if he or she has a genuine need to know that information.

they will be shared among distinct databases. The attributes of these shared tables implement data exchange among users in distinct groups.

4. **Associating user groups and database subsets**

   The biunivocal association between user groups and the databases the decomposition returns is the fundamental input for the next step that defines user access rights.

5. **Configuring the Access Mechanism**

   This step offers a first level of security by granting each user the access rights to access all and only the information in the database associated with the user group. This prevents users from accessing information in the original database that is not necessary for their work. The detailed implementation of this step depends upon both the operating system and database management system that have been adopted. By enabling users access only to the information, they need, we minimize the amount of data that may be lost because of data breaches and unauthorized access.

6. **Choosing the confinement robustness**

   The definition of the database allocation onto physical machines, virtual ones and containers determines the confinement level according to the robustness of the overall system that is required to protect the various information. The choice of the allocation usually depends on the level of criticality of some information.

By following these steps, we can achieve the robustness of the overall system to satisfy data security needs. The only issue to be still evaluated is the overhead of synchronizing the shared tables the MDDF introduces. This overhead is the cost to improve the overall database security.

3.3 *Confinement*

MDDF offers the ability to choose among distinct confinement levels. This is critical to define an allocation that matches the required confinement level as well as other factors such as resource usage, management complexity and security issues.

Mapping	**Confinement Robustness**
Distinct physical machines	High
Distinct virtual machines	Medium-High
Distinct containers	Medium
Simple database decomposition	Low

Table 1: Robustness levels of Alternative Database Mappings.

Table 1 shows the confinement level, i.e. robustness, resulting from alternative database allocations. An allocation of the databases to distinct physical machines offers the best confinement, while a simple database decomposition provides the lowest confinement level. Each alternative solution has its strengths and weaknesses. Let's explore the advantages and disadvantages of allocating to distinct physical machines, distinct virtual machines, distinct containers, and simple database decomposition [17, 18, 19].

An allocation onto distinct physical machines offers a high level of safety and confinement as physical separation enhances security by preventing unauthorized access or inadvertent resource sharing. However, this approach comes with drawbacks such as high resource usage because using separate machines requires a significant hardware investment. Additionally, managing multiple physical machines can be complex and time-consuming.

An allocation onto distinct virtual machines offers a good level of confinement even if as not as robust as physical machines. The advantage lies in the customizable allocation of resources to each VM as needed. Furthermore, VMs can migrate across multiple network nodes. However, there is a resource overhead associated with virtualization, potentially impacting efficiency compared to direct physical machine usage. Moreover, managing multiple virtual machines requires expertise in virtual resource management.

Using distinct containers on the same physical or VM, on the other hand, results in a lightweight solution that enables rapid deployment as they share the host kernel, and this minimizes the overhead for VMs. As a counterpart, the downside is the potential security risk due to the shared underlying operating system. If the host system is compromised, it may pose a threat to the security of all the containers.

If the original database is very simple, a solution using distinct containers is easier to manage and requires fewer resources with reduced complexity. However, the trade-off is a very low confinement as the data remains within just one system. This results in poor confinement that may lead to potential data integrity issues, especially in case of accidental or malicious disk corruption or attacks.

To confirm the difference between the MDDF and normalization, it is worth noticing that all the solutions result in better security than a simple transformation to satisfy the third normal form. The choice among these alternatives must be based on the required level of security and integrity for the whole system. Each option results in a distinct balance between confinement, resource efficiency, and management complexity, and the decision must align with the specific requirements of the system in question.

3.4 *Contexts of use*

Table 2 presents four distinct allocations each with a corresponding usage context:

Solution	**Contexts of use**
Distinct physical machines	• High-security applications where confinement is critical. • Systems with severe compliance requirements (e.g. financial institutions, healthcare, etc.)
Distinct virtual machines	• Medium or large-scale applications. • Applications requiring confinement and flexibility in scalability. • Cloud-based applications where infrastructure management is almost independent of physical resources.
Distinct containers	• Applications based on micro-service architectures. • Rapid application development and deployment. • Applications that require moderate confinement and fast start-up times.

Simple database decomposition	• Small to medium-sized applications. • Applications with simple data models. • Applications where data confinement is not critical.

Table 2: Practical use cases for different database decomposition solutions.

It is worth stressing that the previous allocations are not mutually exclusive if distinct confinement levels are required for distinct tables. As an example, some of the databases the MDDF returns may be allocated to distinct VMs and others to distinct containers on the same VM.

4 Example

In the context of a single hospital managing vital information, our scenario initially consists of a centralized database storing all the data to satisfy the needs of medical staff and patients. The database includes tables with patient details and prescription records, emphasizing simplicity and efficiency in healthcare data management. The overall system ensures medical professionals have access to proper patient information, while patients can conveniently retrieve their own medical records and prescription details. Now we discuss some possible groups of users and the corresponding operations.

4.1 *Users and Database*

The Healthcare Database (HealthDB) is designed to meet the different needs of its users, including Patients, Medical Doctors, Nurses, Administrative Staff, and Statisticians.

Users:

- **Patients (Patients)**: This group requires read access to their medical records, prescription details, exam results, and associated costs. Access is facilitated through a pseudonymous ID for privacy.
- **Medical Doctors (Doctors)**: Medical professionals need read and write access to patient information, medical history, and the ability to prescribe medications and order exams.
- **Nurses (Nurses)**: Nurses require read and write access to patient information, medical history, and the ability to record exam results and administer prescribed medications.
- **Administrative Staff (Admins)**: This group focuses on write access to the CostsTable for billing purposes. They may also have read access to other relevant information.
- **Statisticians (Statisticians)**: Statisticians have read-only access to aggregated data in the Patients table for statistical analysis. They cannot view sensitive information like names and addresses.

Database:

- **Healthcare Database (HealthDB)**: This centralized database contains essential healthcare information:
  - **Patient Information Table (PatientTable)**: Includes pseudonymous IDs (PatientID), comprehensive medical history (MedicalHistory), and IDs of assigned doctors (AssignedDoctorID) and nurses (AssignedNurseID). A mapping table (MappingTable) facilitates the link between pseudonymous IDs and real patient identities, automatically generated on the first hospital visit.

- **Sensitive Data Table (SensitiveDataTable)**: Contains sensitive patient information such as real names (RealName), addresses (Address), and details of who pays the bill (PayerDetails). Linked to PatientTable via pseudonymous IDs.
- **Prescription Records Table (PrescriptionTable)**: Stores data related to prescriptions, including unique prescription IDs (PrescriptionID), details of prescribed medications (MedicationDetails), dosage (Dosage), prescribing doctor IDs (PrescribingDoctorID), and associated costs (AssociatedCost).
- **Exam Records Table (ExamTable)**: Stores information about tests and exams, including unique IDs (examID) and test and exam results (Results).
- **Costs Table (CostsTable)**: Manages the costs of medicines and exams, with transaction IDs (TransactionID), item descriptions (Item), and associated costs (Cost). Accessed by administrative staff for billing purposes.
- **Mapping Table (MappingTable)**: Simplifies the mapping between pseudonymous patient IDs (PatientID) and their real identities (RealIdentityID).
- **Statistics Table (StatisticsTable)**: Contains aggregated and de-identified data from the Patients table, accessible to statisticians for analysis. Includes aggregated data IDs (AggregatedDataID) and de-identified aggregated data (AggregatedData).

The following table summarizes the key fields and data types for each table in the healthcare database:

Table	**Column Name**	**Data Type**
PatientTable	PatientID	Pseudonymous ID
	MedicalHistory	Text
	AssignedDoctorID	Numeric
	AssignedNurseID	Numeric
SensitiveDataTable	PatientID	Pseudonymous ID
	RealName	Text
	Address	Text
	PayerDetails	Text
PrescriptionTable	PrescriptionID	Numeric
	MedicationDetails	Text
	Dosage	Numeric
	PrescribingDoctorID	Numeric
	AssociatedCost	Numeric
ExamTable	examID	Numeric
	Results	Text

CostsTable	TransactionID Item Cost	Numeric Text Numeric
MappingTable	PatientID RealIdentityID	Pseudonymous ID Numeric
StatisticsTable	AggregatedDataID AggregatedData	Numeric Text

Table 3: Tables and fields in the centralized database

In this scenario, the patients' names are replaced with pseudonymous IDs, ensuring a high level of privacy, and the mapping table facilitates the association between these pseudonymous IDs and technical systems. The administrative staff manages costs, while statisticians analyze aggregated data without compromising individual patient identities.

4.2 *Implementation*

The implementation of MDDF adopts a systematic approach to satisfy the unique needs of user groups of the healthcare system. The key steps involve identifying users and their specific needs, computing the access permissions to satisfy these needs, and ensuring secure data management within the Healthcare Database (HealthDB). Let's delve into a short, high-level overview of these crucial implementation steps. These are the steps to implement MDDF:

1. **Identify Users and Their Needs**:

   (a) **Patients**

   - **Needs**:
     - Read access to their own medical records (MedicalHistory in PatientTable).
     - Read access to their prescription details (MedicationDetails, Dosage in PrescriptionTable).
     - Read access to their exam records (Results in ExamTable).
   - **Required Access**: Read access to PatientTable, PrescriptionTable, ExamTable in the Healthcare Database (HealthDB).

   (b) **Doctors**

   - **Needs**:
     - Read and write access to patient information for assigned patients (PatientTable).
     - Write access to prescribe medications (PrescriptionTable).
     - Write access to order exams (ExamTable).
   - **Required Access**: Read and write access to PatientTable, DoctorsTable, PrescriptionTable, Examtable in the Healthcare Database (HealthDB).

   (c) **Nurses**

   - **Needs**:
     - Read and write access to patient information for assigned patients (PatientTable).

  - Write access to record exam results (Examtable).
  - Write access to administer prescribed medications (PrescriptionTable).

• **Required Access**: Read and write access to PatientTable, NursesTable, Examtable, PrescriptionTable in the Healthcare Database (HealthDB).

(d) **Administrative Staff (Admins)**

- **Needs**:
  - Write access to the CostsTable for billing purposes (CostsTable).
- **Required Access**: Write access to CostsTable in the Healthcare Database (HealthDB).

(e) **Statisticians**

- **Needs**:
  - Read-only access to aggregated data in the PatientsTable for statistical analysis (StatisticsTable).
- **Required Access**: Read-only access to StatisticsTable in the Healthcare Database (HealthDB).

4.3 *Create Database Subsets*

A critical step to apply MDDF is the one that computes the database subsets to satisfy the specific needs of distinct user groups. This ensures that each subset contains only the relevant tables and fields the designated user roles require. By creating the distinct database subsets of HealthDB, such as MedDB for medical staff, PatientDB for patients, AdminDB for administrative staff, and StatDB for statisticians, we adhere to the principle of least privilege. The subsets the decomposition returns are listed below:

Subset MedDB: This subset includes only the tables that in the main database are relevant to Medical Staff. Key tables comprise:

- **PatientTable**: PatientID, MedicalHistory, AssignedDoctorID, AssignedNurseID
- **DoctorsTable**: DoctorID, Name, Specialty
- **NursesTable**: NurseID, Name, AssignedPatients
- **examsTable**: examID, Results
- **PrescriptionsTable**: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID
- **MedicineCostsTable**: TransactionID, Item, Cost

Subset PatientDB: This subset includes only the tables in the main database relevant to patients. Key tables include:

- **PatientTable**: PatientID, MedicalHistory, AssignedDoctorID, AssignedNurseID
- **SensitiveDataTable**: PatientID, RealName, Address, PayerDetails
- **examsTable**: examID, Results
- **PrescriptionsTable**: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID

Subset AdminDB: This subset includes only the tables from the main database relevant to Administrative Staff. Key tables comprise:

- **CostsTable**: TransactionID, Item, Cost

Subset StatDB: This subset includes only the main database tables relevant to Statisticians. Key tables comprise:

- **StatisticsTable**: AggregatedDataID, AggregatedData

These subsets ensure that each user group can access all and only to the information the corresponding roles require according to the principle of least privilege to improve data security and privacy.

The tables and fields of the two main subsets are shown in detail below:

Subset	**Fields**
MedDB	- **PatientTable**: PatientID, MedicalHistory, AssignedDoctorID, AssignedNurseID - **DoctorsTable**: DoctorID, Name, Specialty - **NursesTable**: NurseID, Name, AssignedPatients - **examsTable**: examID, Results - **PrescriptionsTable**: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID - **MedicineCostsTable**: TransactionID, Item, Cost
PatientDB	- **PatientTable**: PatientID, MedicalHistory, AssignedDoctorID, AssignedNurseID - **SensitiveDataTable**: PatientID, RealName, Address, PayerDetails - **examsTable**: examID, Results - **PrescriptionsTable**: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID

Table 4: Fields in Medical and Patient subsets

2. **Associating User Groups and Database Subsets**: The biunivocal association between a user group and one of the databases the decomposition returns is the fundamental input for the next step that defines user access rights.

   - **Define User-Subset Mapping**: The association between each user group and a database subset should map, Medical Staff into the MedDB subset, Patients into the PatientDB subset while Doctors should be mapped to the MedDB subset (with access to fields like DoctorID, Name, Specialty), Nurses should be mapped into the MedDB subset (with access to fields like NurseID, Name, AssignedPatients), and Administrative Staff should be mapped into the AdminDB subset (with access to fields like TransactionID, Item, Cost). Statisticians are to be mapped into the StatDB subset (with access to fields like AggregatedDataID, AggregatedData).
   - **Use Mapping for Access Control**: Starting from the mappings previously defined, we can configure access control mechanisms. The mapping implies the assignment of specific access rights and permissions to each user in a user group on the corresponding database subset.

- **Regularly Update Mapping**: User groups should be reviewed with a fixed frequency because the group of a user can be updated to consider changes in roles, responsibilities, and database structure. This ensures that access control remains aligned with organizational requirements.

The described process ensures that the mapping between users and database subsets is clearly defined, access control mechanisms are implemented according to this mapping, and updates occur to adapt to organizational changes.

3. **Configure Access Mechanisms**: After creating the database subsets, it is essential to define who has access to each subset and what level of access is allowed:

   (a) **Patients** should have read access to the **PatientDB** subset, allowing them to view and access their pseudonymous medical records, prescription details (fields: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID), and exam records (fields: examID, Results).

   (b) **Doctors** should have read and write access to the **MedDB** subset, like Medical Staff, with the additional ability to prescribe medications (fields: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID) and order exams (fields: examID, Results).

   (c) **Nurses** should have read and write access to the **MedDB** subset as Medical Staff, but with the additional ability to record exam results (fields: examID, Results) and administer prescribed medications (fields: PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID).

   (d) **Administrative Staff (Admins)** should have write access to the **AdminDB** subset, to enable them to manage costs associated with medicines and exams (fields: TransactionID, Item, Cost).

   (e) **Statisticians** should have read-only access to the **StatDB** subset, to enable them to analyze aggregated and de-identified data in the **PatientsTable** for statistical purposes (fields: AggregatedDataID, AggregatedData).

After defining the relationships between the user groups and the various database subsets, it is essential to adopt some procedures to guarantee security and data management. Below, we list some further details of each procedure:

- **Authentication and Authorization**: It ensures that every user is authenticated, allowing only authorized users to access data in their respective subsets. This may include the use of usernames and passwords, two-factor authentication, or other secure authentication methods. Additionally, it is crucial to assign specific roles and permissions based on user responsibilities. For example, Medical Staff and Doctors must have the role “Medical Professional” with full permissions for the **MedDB** subset, while Patients must have appropriate read access roles for the **PatientDB** subset. Nurses and Administrative Staff must also be assigned roles with relevant permissions.

- **Auditing and Monitoring**: It implements an audit system to track who accesses the data and what operations are invoked. This system will help to detect suspicious activities or unauthorized access. As an example, it is critical to record who accesses tables in the **MedDB** and **AdminDB** subsets and to track the updates to these data.

- **Key Management**: Encryption and key management are important mechanisms to protect sensitive data, as an example, prescription data in the **PatientDB** subset must be encrypted, and encryption keys must be securely managed to prevent unauthorized access.
- **Backup and Recovery**: They ensure data integrity despite accidental loss or damage by implementing regular backup procedures for each subset. Backups must be encrypted and stored in secure locations, to support data recovery in case of emergencies.
- **User Training**: Lastly, user training on secure data access and system usage is important. Users, including Medical Staff, Patients, Doctors, Nurses, Administrative Staff, and Statisticians, must be aware of security policies and best practices to ensure that sensitive data is adequately protected.

By implementing this process, an organization can ensure appropriate data management and security, according to the specific needs and responsibilities of each type of user.

4. **Synchronize Common Tables in Subsets**: If some tables are shared, i.e. they appear in distinct database subsets, as in this example, multiple copies of the table exist, one for each subset. Now it is essential to synchronize the updates of these shared tables to ensure data consistency and integrity despite replication. When multiple users or divisions can update a shared table, it is necessary to:

- **Identify Common Tables**: A first step must identify the tables containing shared data among subsets. For instance, in our healthcare organization, tables such as **PatientTable** (fields: PatientID, MedicalHistory, AssignedDoctorID, AssignedNurseID) and **examsTable** (fields: examID, Results) are shared between the **MedDB** and **PatientDB** subsets, and synchronization is crucial to avoid data inconsistencies.
- **Implement Synchronization Rules**: It defines rules to implement data synchronization among subsets. These rules cover scenarios such as updates, insertions, and deletions. For example, when medical staff updates patient data in the **MedDB** subset, these changes must be reflected in the **PatientDB** subset so that patients have access to the updated information. Similar synchronization rules must be established for shared tables among other subsets.
- **Plan Synchronization**: It plans when data synchronization occurs. For instance, update to critical data should be immediate to ensure immediate alignment. Less critical data can tolerate a weaker consistency. As an example, data update for administrative staff can occur once a day, usually at night. A weekly update may be appropriate for statisticians. The synchronization strategy should consider the needs of all user groups, including Medical Staff, Patients, Doctors, Nurses, and Administrative Staff.

The adoption of these synchronization practices tunes the synchronization overhead to the urgency and relevance of data for each user group and ensures that shared data remains consistent across different subsets, supporting the integrity of the overall healthcare database.

4.4 *Optimizing Confinement Robustness*

The optimization of confinement robustness involves the definition of the degree of physical and logical separation among the database subsets according to data sensitivity and security needs. The steps to follow are:

- **Data Classification**: It classifies data based on its sensitivity level. For example, personal information (RealName, Address, PayerDetails) and medical history (MedicalHistory) should be classified as highly sensitive, while public or non-sensitive data are classified as less critical.
- **Assigning Security Levels**: It assigns a security level to each database subset based on data classification. For instance, the financial database (CostsTable) requires the highest security level, while the other database may require a lower one.
- **Physical Separation**: To enhance confinement robustness, it evaluates hosting the **PatientDB** subset on a separate physical machine and utilizing virtualization for other subsets.
  - **Separate Physical Machine (PatientDB)**: The **PatientDB** subset should be allocated to a dedicated physical server for maximum confinement. This is required when absolute physical separation is a critical security requirement.
  - **Virtual Machines (VMs)**: We show how to allocate the other subsets onto three virtual machines (VMs) to ensure logical separation within a single shared physical server. This includes:
    - **VM Prescription**: A virtual machine to manage prescription data (PrescriptionID, MedicationDetails, Dosage, PrescribingDoctorID, AssociatedCost).
    - **VM Exam Costs**: A virtual machine dedicated to exam cost data (TransactionID, Item, Cost).
    - **Containerization (Mapping and Statistics)**: Within the VM “Exam Costs” machine we allocate to distinct containers the **MappingTable** and **StatisticsTable**. Containers offer lightweight and efficient confinement, striking a balance between complete confinement and resource efficiency.

  This solution ensures that sensitive data is appropriately separated and secured, with varying degrees of physical and logical confinement according to the security needs of each database subset.

This example shows how the decomposition of a database into subsets according to the user requirements can impact both performance and security.

5 Comparison of alternative synchronization solutions

This section evaluates alternative solutions to synchronize tables shared among distinct database subsets. It compares the features and capabilities of two APIs (Application Programming Interfaces) based on, respectively, triggers and events. A synchronization implemented through an API based on events offers several advantages:

- Support for different types of databases and platforms, simplifying the management of replication between heterogeneous databases.
- Advanced features such as conflict resolution, data filtering and bi-directional synchronization.
- Easy configuration and maintenance thanks to the web interface and documentation available.

However, due to the overhead it introduces, the solution may not be the optimal one for scenarios with severe performance requirements.

The advantages of synchronization based on triggers are:

- Better control on the management of replicated information that, in turn, can support some problem features to optimize performance.
- Reduced overhead and resource usage, as the database directly execute triggers.

The main problems the adoption of this solution poses are:

- Writing and maintaining the trigger code require specific skills and time.
- Implementing advanced features such as conflict resolution and data filtering that are available in event-based solutions.
- Support of multiple database types because distinct triggers must be written for each database.

Figure 1 shows an example of synchronization between two databases sharing some tables. In the example, distinct users access a sales portal and a production one that shares some information on produced goods.

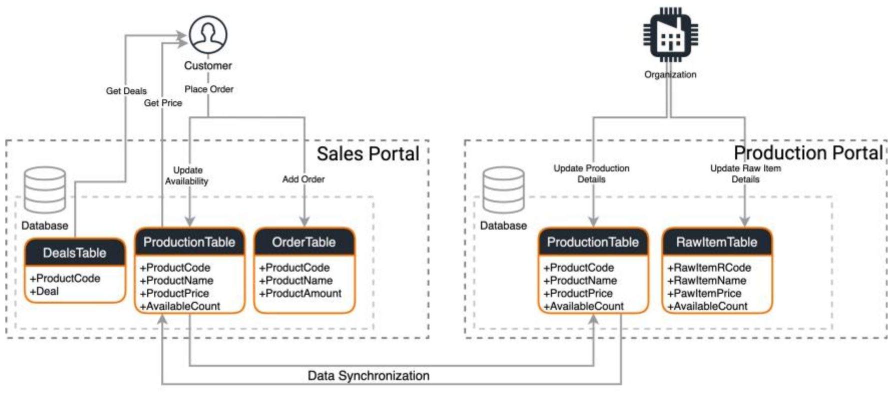

Figure 1: Synchronization of two separate databases with a common table

5.1 *Trigger - API*

A trigger is a database event that fires the execution of a code fragment as a response to predefined conditions to preserve the integrity of the database, ensuring that some actions are executed when specific changes occur. In some cases, the trigger may involve several databases. This is complex when the databases are mapped onto distinct physical machines or containers because the execution of the trigger requires to use an API and the APIs enable communication among software applications and this makes it possible to execute a trigger on multiple databases. One approach to trigger execution via APIs is to implement an API-driven, software architecture that uses APIs as the main communication mechanism among the various components. Now the trigger is implemented as an API endpoint that can be invoked by other architecture components. When the API endpoint is invoked, the trigger is executed automatically, and it fires actions on multiple databases. In conclusion, the execution of a trigger on multiple databases can be complex when the databases are located on different physical machines or containers, but the adoption of an API can make the trigger transparent for the user.

5.2 *SymmetricDS*

An alternative solution adopts SymmetricDS an open-source tool to support data replication and synchronization among databases. It is used by several companies and organizations to replicate distributed environments, such as bank branches, remote offices, or geographically separated data centers (e.g. CDN). One of its main advantages is the ability to synchronize data in real time, with a latency of a few milliseconds. SymmetricDS adopts a client-server architecture, where the central server maintains the configuration synchronization and clients connect to the server to update data. The SymmetricDS server can support several databases, including Oracle, MySQL, PostgreSQL, and SQL Server. SymmetricDS clients are available for Java, .NET, and Android. Multiple servers may interact with the same clients, and this avoids centralization and catastrophic drop points. An example of a schema for synchronizing three databases is shown in Fig.2.

SymmetricDS supports alternative synchronization strategies, including unidirectional, unidirectional with overwriting, bidirectional and cascading. The strategy to adopt depends on the needs of both the user needs and the system features. For example, unidirectional synchronization is useful for replicating data from a central database to several remote databases, while bi-directional synchronization is useful for keeping data up to date on both databases. SymmetricDS offers advanced features, such as event-based synchronization, event-driven data replication, real-time data replication, conflict management and data encryption. Event-based synchronization makes it possible to replicate data only when a specific event occurs, such as the insertion of a new record or the update of an existing one. Real-time data replication allows data to be replicated in real-time, with a latency of milliseconds. Conflict management can handle situations where data are updated simultaneously on two databases. Data encryption can protect sensitive data during synchronization.

5.3 *Performance comparison between the two solutions*

This section compares the performance of the two synchronization solutions: Trigger-API and SymmetricDS. A series of experiments have been implemented to evaluate the effectiveness of

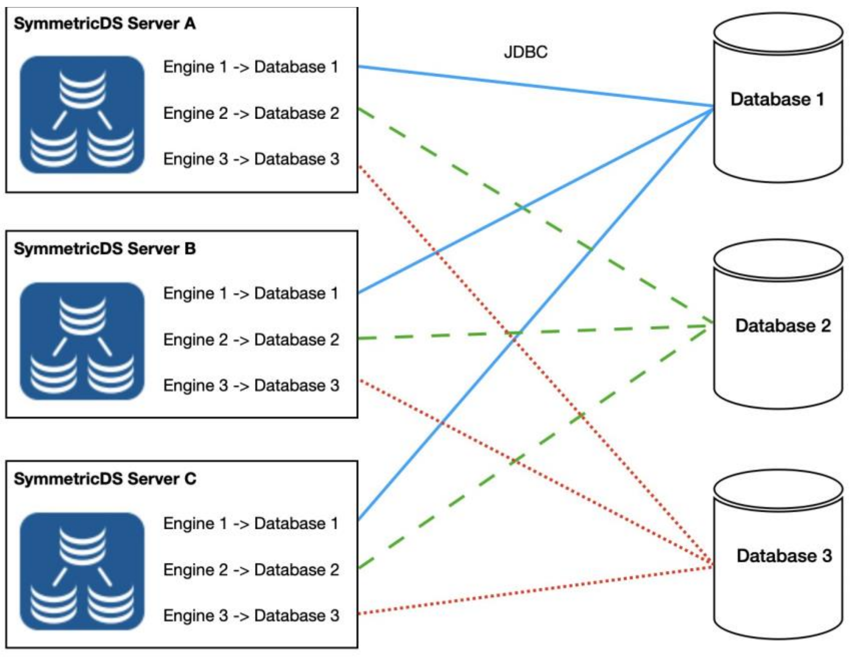

Figure 2: SymmetricDS synchronization scheme

the two solutions under various conditions. First, we introduce the specifications of the server used in the experiments:

- **Operating System**: Ubuntu 22.04.3 LTS
- **RAM Memory**: 16 GB
- **Processor**: Intel Xeon Gold 5120 CPU @2.20GHz (8 core)

- **Storage Unit**: 128 GB SSD

5.3.1 Latency Experiment

The latency experiment measures the time each solution takes to propagate changes from one database to one or more remote ones. The results are shown in the following table:

Configuration	Trigger-API Latency (ms)	SymmetricDS Latency (ms)
Local Configuration	59	95
Remote Configuration	181	258
Distributed Configuration	800	2160

The latency experiment tests three alternative configurations:

1. **Local configuration**: The database and the synchronization system share the same machine. Latency times are relatively low, with Trigger-API showing a latency of 59 ms and SymmetricDS of 95 ms. This is an ideal case with the lowest latency.
2. **Remote configuration**: The database and the synchronization system run on two distinct physical machines in a local network. Here we see an increase in latency, with Trigger-API taking 181 ms and SymmetricDS 258 ms to propagate changes. This scenario represents a common situation where the database is on a distinct server in a local network.
3. **Distributed configuration**: This configuration represents a more complex environment where the database is on a geographically dispersed network. To simulate this geographical dispersion a VPN was configured to connect the databases on physically separate servers. Here latency largely increases, with Trigger-API requiring 800 ms and SymmetricDS requiring 2160 ms for synchronization.

In general, the results show that Trigger-API has a lower latency than SymmetricDS in all three configurations. This difference is large because Trigger-API was tuned for this environment to optimize the synchronization process. Anyway, other parameters are important such as the complexity of the implementation, the scalability, and the system specifications. At the price of a slightly higher latency, SymmetricDS may be the preferred choice in scenarios that require advanced synchronization capabilities, and where the synchronization and configuration complexity are not a significant obstacle. In the following, we will examine further aspects of the two solutions and show further experiments to evaluate performance in a wider range of scenarios to enable a more informed choice of the best solution to satisfy project requirements.

5.3.2 Scalability Experiment

In the scalability experiment, several trials were conducted to test the ability of the two solutions to handle increasing workloads. The following tables show the CPU utilization for both solutions under different workload levels.

Workload CPU	CPU Utilization Trigger-API	CPU Utilization SymmetricDS
Light	5%	25%
Medium	30%	60%
Heavy	70%	90%

The load levels are defined as follows:

- **Light Workload**: 100 parallel iterations are executed using the ThreadPoolExecutor[2], each of which requires a relatively low level of resources.
- **Medium Workload**: 500 parallel iterations, increasing the workload compared to light one, but keeping it manageable in terms of resources.
- **Heavy Workload**: 1000 parallel iterations are performed, representing a very high workload that requires considerable use of resources, both in terms of CPU and memory.

A more detailed discussion of the result follows:

- **Light Workload**: In this configuration, the workload was relatively low: Trigger-API generated a CPU utilization of 5%, SymmetricDS generated a utilization of 25%. This indicates that both solutions can handle light loads efficiently, but Trigger-API requires a much lower amount of s resources.
- **Medium Workload**: With an increase in workload to a medium level, Trigger-API utilized the CPU at 30%, while SymmetricDS reached 60%. Both solutions successfully handled the average workload, but Trigger-API still required fewer resources than SymmetricDS.
- **Heavy Workload**: With a heavy workload, Trigger-API reached 70% CPU utilization, while SymmetricDS required 90%. In this scenario of heavy load, both solutions start to reach their limits, but Trigger-API still has a lower utilization than SymmetricDS.

While CPU utilization may vary depending on the hardware and the execution environment, the overall results suggest that Trigger-API usually requires fewer system resources than SymmetricDS under several workloads.

5.3.3 Conflict Resolution Experiment

This experiment creates data conflicts to measure the time each solution takes to resolve a conflict. The results are expressed in terms of the time each solution takes to resolve conflicts at different levels. The results are shown in the following table:

Conflicts Resolved	Trigger-API Time (ms)	SymmetricDS Time (ms)
100%	76	180
90%	55	70
80%	40	60

- **Conflicts Solved**: The percentage of conflicts that were resolved in the experiment. The percentage increases with the number of conflicts to be handled.
- **Trigger-API time (ms)**: The time in milliseconds taken by Trigger-API to resolve conflicts at each percentage level.

[2] *ThreadPoolExecutor* is a class in Python's *concurrent.futures* module that allows you to manage a thread pool for executing functions in parallel. It is useful for improving the efficiency and scalability of operations which can be execute d concurrently, in this context it was used to simulate many parallel iterations.

- **SymmetricDS time (ms)**: The time in milliseconds taken by SymmetricDS to resolve conflicts corresponding to the specified percentage.

It is important to note that the data in the table confirms that Trigger-API is faster in conflict resolution in all conflict categories.

5.3.4 Two-way Synchronization Experiment

The bidirectional experiment synchronizes data between two databases in both directions: from the main database to the remote database and vice versa. The following table shows the results:

Direction Synchronization	Trigger-API Time (ms)	SymmetricDS Time (ms)
Main to Remote	161	215
Remote to Main	166	256

In both directions, the performances of Trigger-API are slightly better than those of SymmetricDS. Synchronization from the main database to the remote database takes 161 ms with Trigger-API and 166 ms with SymmetricDS, while synchronization from the remote database to the main database takes 215 ms with Trigger-API and 256 ms with SymmetricDS. These times represent average performance, and they depend upon the number of databases to be synchronized, the amount of data and the network congestion. Overall, this experiment shows that both solutions can manage bidirectional data synchronization in a few milliseconds, but the Trigger-API solution offers a slightly better performance.

5.3.5 Variable Load Experiment

The variable load experiment evaluates the responsiveness and scalability of solutions in response to workload fluctuations. The results are shown in the following table (the unit of measurement used is tps, transactions per second[3]):

Scenario Workload	Performance Trigger-API	Performance SymmetricDS
Peak Activity	63 tps	51 tps
Calm Period	37 tps	49 tps

The table considers two different workload scenarios that may occur in a synchronization environment:

- In "**Scenario 1 - Peak Activity**", a high workload with several transactions to be synchronized. In this scenario, Trigger-API manages 63 transactions per second, while SymmetricDS only manages 51 transactions per second. Thus Trigger-API is more efficient during peak activity.

- In "**Scenario 2 - Calm Period**", the workload is reduced, with less transactions to be synchronized. Here, SymmetricDS outperforms Trigger-API with 49 transactions per second, while Trigger-API handles 37 transactions per second. SymmetricDS shows a greater responsiveness during quiet periods.

These figures provide an objective evaluation of the performance of the two solutions under different workloads. They underline that Trigger-API is more efficient during activity peaks, while SymmetricDS is more responsive during quiet periods.

[3] The term "transactions per second" indicates the number of synchronization operations completed in one second.

6 Conclusions

In conclusion, the Multilevel Database Decomposition Framework offers several benefits, including:

- **Preserve privacy:** In healthcare, where sensitive patient information is handled, database decomposition improves security by restricting access to specific subsets. In this way, users will only have access to the data they need, protecting the privacy of individuals. This is crucial for complying with privacy regulations such as HIPAA [10] and GDPR [3]. The framework supports the implementation of precise access controls, ensuring that only authorized medical staff can access specific patient information.
- **Reduced Impact of Security Breaches:** The healthcare sector is a prime target for cyber-attacks. Database decomposition limits the impact of security breaches, as compromising one subset doesn't expose the entirety of patient records.
- **Reduction of complexity:** By decomposing the database into smaller subsets, users can use simpler data structures that are easier to understand.
- **Improved performance:** Access to smaller, more tailored tables and relationships improves database performance.
- **Efficient Data Retrieval for Medical Research:** Researchers can access specific subsets relevant to their studies, streamlining data retrieval for medical research purposes. This can contribute to advancements in healthcare through data-driven insights.
- **Scalability for Growing Healthcare Systems:** As healthcare systems expand, the multilevel database decomposition framework allows for scalability. New subsets can be added to accommodate the growing volume of patient data.

However, the proposed framework also presents some challenges:

- **Management complexity:** Managing multiple subsets could increase the complexity in database administration and require greater resources for maintenance and updates.
- **Risk of data inconsistency:** Database decomposition requires the adoption of proper synchronization mechanisms.

References

[1] A. Brahma and S. Panigrahi. “Application of soft computing techniques in database intrusion detection”. In: *Intelligent Technologies: Concepts, Applications, and Future Directions*. Springer, 2022, pp. 201–221.

[2] A. Cuzzocrea and H. Shahriar. “Data masking techniques for NoSQL database security: A systematic review”. In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, pp. 4467–4473.

[3] European Union. *Data protection in the EU*. 2023. url: https://ec.europa.eu/info/law/law-topic/data-protection_en.

[4] E. Fern´andez-Medina and M. Piattini. “Designing secure databases”. In: *Information and Software Technology* 47 (2005), pp. 463–477.

[5] M. Humayun et al. “Security threat and vulnerability assessment and measurement in secure software development”. In: *Computers, Materials and Continua* 71 (2022), pp. 5039–5059.

[6] S. Ibrahim et al. “A novel data encryption algorithm to ensure database security”. In: *Acta Infologica* 7.1 (2023), pp. 1–16.

[7] N.A. Al-Sayid and D. Aldlaeen. “Database security threats: A survey study”. In: *2013 5th International Conference on Computer Science and Information Technology*. 2013, pp. 60–64.

[8] I. Singh et al. “Database intrusion detection using role and user behavior based risk assessment”. In: *Journal of Information Security and Applications* 55 (2020), p. 102654.

[9] P. Yang, N. Xiong, and J. Ren. “Data security and privacy protection for cloud storage: A survey”. In: *IEEE Access* 8 (2020), pp. 131723–131740.

[10] D. L. Anthony, A. Appari, M. E. Johnson, Institutionalizing HIPAA compliance: Organizations and competing logics in U.S. health care, Journal of Health and Social Behavior 55 (2014) 108–124. URL: https://doi.org/10.1177/0022146513520431. doi:10.1177/0022146513520431, PMID: 24578400.

[11] J. H. Saltzer, M. D. Schroeder, The protection of information in computer systems, Proc. IEEE 63 (1975) 1278–1308.

[12] R. Smith, A contemporary look at Saltzer and Schroeder’s 1975 design principles, IEEE Security & Privacy 10 (2012) 20–25.

[13] M. Campobasso, L. Allodi, Impersonation-as-a-service: Characterizing the emerging criminal infrastructure for user impersonation at scale, in: Proc. of the IEEE Int. Symposium on Secure Software Engineering, volume 1, IEEE, 2006, p. 1.

[14] S. A. Moiz, P. Sailaja, G. Venkataswamy, S. N. Pal, Database replication: A survey of open source and commercial tools, International Journal of Computer Applications 13 (2011) 1–8.

[15] M. Binjubeir, A. A. Ahmed, M. A. B. Ismail, A. S. Sadiq, M. Khurram Khan, Comprehensive survey on big data privacy protection, IEEE Access 8 (2020) 20067–20079.

[16] I. Linkov, F. Baiardi, M. V. Florin, S. Greer, J. H. Lambert, M. Pollock, B. D. Trump, Applying resilience to hybrid threats, IEEE Security & Privacy 17 (2019) 78–83.

[17] J. Wu, et al., An access control model for preventing virtual machine escape attack, Future Internet 9 (2017) 20.

[18] A. Y. Wong, et al., On the security of containers: Threat modeling, attack analysis, and mitigation strategies, Computers & Security 128 (2023) 103140.

[19] S. Shringarputale, P. McDaniel, K. Butler, T. La Porta, Co-residency attacks on containers are real, in: Proc. of the 2020 ACM SIGSAC Conf. on Cloud Computing Security Workshop, ACM, 2020, pp. 53–66.