Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Encrypted Query Processing in DuckDB

Author: Sam Ansmink (2578816)

1st supervisor: Prof.dr. Peter Boncz (CWI, VU)
2nd reader: dr. ir. Marc Makkes (VU)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 30, 2021

"Remember Kids, The Only Difference Between Screwing Around and Science Is Writing It Down"

from Mythbusters Episode 186*, by Adam Savage*

Abstract

With the global adoption of cloud computing, database outsourcing has become a common practice. Growing privacy concerns have led to a demand for better privacy of data stored in databases running on untrusted hardware. To supply this demand for privacy, many different encrypted database management systems (EDBMS) have been proposed in recent literature. In this Thesis, we focus on systems designed for analytical workloads. Database systems designed for analytical workloads have been well researched over the past decades, with many new designs and optimization techniques having been developed, such as vectorized query execution and columnar compression. In EDBMS literature however, these techniques have not yet been applied.

Therefore, we present an EDBMS design based on the modern analytical database, DuckDB, that deploys both vectorized execution and compression to achieve minimal performance overhead while providing strong security guarantees. The design is based on Intel SGX, an instruction set extension supported by many Intel CPUs to perform secure computation. The DuckDB-based EDBMS design is evaluated using the industry-standard TPC-H benchmark suite. In the experimental evaluation we show that well constructed encryption scheme leads to an overhead of 22%. Adding compressed execution is shown to further reduce this overhead by up to 2.12×. The evaluation further demonstrates that both presented designs for SGX integration can be viable with reasonable performance overheads for some TPC-H queries, but for other queries suffer severely from the limitations of the current generation of Intel SGX.

Acknowledgements

Firstly, I want to thank my supervisor, Peter Boncz, for his support and guidance throughout the project. The discussions and feedback on my research and the multiple drafts have been very insightful and helpful. Peter's knowledge on databases is clearly among the best in the field and it has been an amazing learning experience to work with him. I also want to thank Peter for the frequent calls we had discussing the research during the COVID-19 global pandemic that forced everyone to work from home. These calls were not only were helpful for my research, but also made working on the project in isolation a bit less isolated.

I also would like to thank Marc Makkes and Mark Raasveldt for their insights, advice, and help during my time at CWI. The help I got from Mark Raasveldt when coding and debugging with DuckDB has saved me many hours.

Finally, I want to thank all my friends and family, especially Marianne, Ronald, Koen, Jorinde, and Hannah, for their support and encouragement. This work would not have been possible without them.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

1.1.1 Relevance

In recent years, cloud computing has become indispensable. For many companies outsourcing computing infrastructure is favourable over purchasing on-site hardware. Instead of owning and managing server hardware, software, platforms and even complete infrastructures are offered as services running on hardware owned by cloud providers. Database management systems(DBMS) have been no exception and the Database-as-a-service (DBaaS) market is projected to reach US$399.5 billion by 2027 (14).

While for many companies the security and privacy of DBaaS providers is sufficient, for some companies such as insurance providers, medical facilities or governments, the threat of malicious/negligent cloud providers is problematic. Furthermore, privacy laws are becoming more strict, limiting the way businesses can process private data on outsourced hardware. Securing outsourced data by itself is not a hard problem and can be easily solved by using encryption. Most, if not all, cloud providers offer a variety of solutions based on encryption. The difficulties arise when trying to support searching or even rich SQL queries over encrypted data. In scientific literature, this problem of supporting SQL over encrypted data was recognized nearly 20 years ago (15), and has recently seen a lot of new research. The industry itself has also been involved, with Microsoft and IBM publishing on the subject (16)(7)(6)(8). Even some commercial products are available today such as Microsoft's Always Encrypted (16), which uses software-based enclaves to allow SQL queries over encrypted columns. However, despite the comprehensive research being done, to this day no commercial product is currently offered that supports the

full SQL standard over encrypted data while fully protecting against the threats faced in outsourced databases.

1.1.2 Different types of EDBMS

Over the last decade, many different approaches to implementing an Encrypted Database Management System (EDBMS) have been developed. Currently, there is no silver bullet solution and consequently every EDBMS to date has had to make trade-offs. Generally, EDBMS design is a trade-off between performance, privacy, and functionality. To give an example: to achieve maximum privacy, an EDBMS should hide the queries from the untrusted environment, but hiding queries through special encryption schemes or using special hardware practically always comes at the price of reduced performance. Depending on the requirements, EDBMSs choose different priorities in the performance-privacy-functionality trade-off. In some EDBMS, the balance in this trade-off can even be configured by the database administrator. For example, in CryptDB (3), different encryption schemes can be chosen per database column that will either offer more functionality or more confidentiality. Besides the difference in goals, EDBMSs also differ widely in what type of underlying primitives are used to achieve security. The underlying primitives used has a large impact on the characteristics of the resulting system. In chapter 3, we will make a clear distinction between the different types of systems and go into detail in the most relevant systems found in the literature to date.

1.1.3 High-efficiency OLAP

While much research has been done on EDBMSs based on traditional OLTP database systems such as Postgres (4)(3) or Microsoft SQL server (6), relatively few research has been done focusing on creating EDBMSs optimized for OLAP workloads. The research into OLAP-oriented EDBMSs that does exist, does not use a DBMS engine with a modern query processing model such as vectorization (17) or JIT-compilation (18). Existing OLAP research is often based on the same traditional row-store systems that are used for OLTP workloads (19)(4), or on distributed query processing systems such as Apache Spark (20)(21). While this research provides valuable insight in EDBMS design, in OLAP applications where performance is critical, these systems will always perform significantly worse than an EDBMS using a performance-optimized query processing engine such as MonetDB (22), HyPer (23) or DuckDB (24). This performance improvement can be crucial for many real-world use cases. Generally, companies have large amounts of data to be

analyzed and increased efficiency translates directly into either lower cloud provider cost, or a more detailed analysis at the same cost.

1.2 Research Questions

This thesis will focus on designing the systems architecture of a new high-efficiency OLAP EDBMS. The basis for this EDBMS design will be DuckDB (24), an analytical embedded database system developed at CWI using modern techniques such as a vectorized execution engine and a columnar storage layout. To the best of our knowledge, no previous research into implementing encryption this type of database has been done before. With regards to the performance-privacy-functionality trade-off, this research will focus strongly on performance. It will research the level of functionality and privacy we can achieve when very little sacrifices in performance are to be made. So for example, query privacy or a specific SQL functionality may be dropped from the design if it results in high performance overheads.

Based on this goal we define the following research questions:

1. What are the use cases for encryption enabled DuckDB and what are the corresponding trust and threat models?

2. How to implement encryption in DuckDB at a negligible performance overhead?

   (a) What is the optimal granularity to encrypt the data? (e.g. per vector or per value)

   (b) What encryption scheme is most suitable?

   (c) What functionality can we support when only a negligible performance overhead is allowed?

3. How to integrate a trusted hardware solution into our encrypted DuckDB implementation

   (a) Which solution is most suitable?

   (b) How to integrate a trusted hardware solution into encryption enabled DuckDB to improve privacy at a negligible performance overhead?

1.3 Structure

This thesis is structured as follows. In Chapter 2, the background information essential to the understanding of this thesis is provided. In Chapter 3 an overview of the EDBMS research field is given along with an in-depth analysis of the most relevant related literature. In Chapter 4, the requirements specific to a DuckDB-based EDBMS are determined by analyzing several use-cases and their corresponding security models. In Chapter 5, a baseline implementation of encrypted DuckDB is presented and evaluated. The baseline implementation is then used to quantify the overhead of decryption and to demonstrate the effectiveness of compression to mitigate decryption overhead. In Chapter 6, two designs are given for integration of Intel SGX into the baseline implementation. These designs are then compared and evaluated. Finally, in Chapter 7 the conclusions and answers to the research questions from Section 1.2 are given, along with several suggestions for future work.

2

Background

In this section we will discuss the background information and concepts required for understanding both the related work section and the rest of this thesis. Firstly, database fundamentals are discussed. Secondly, the basic concepts of different types of encryption are discussed. Finally, we discuss how security models are commonly defined in secure systems literature.

2.1 Databases

A database (DB) is defined as an organized collection of data. In the context of our research, this data is stored digitally in a computer system. Databases can come in a wide variety of sizes, ranging from kilobytes up to multiple petabytes or more.

Relational databases One of the most common types of databases are relational databases (RDB). RDBs follow the relational model introduced by Codd (25). In the relational model, data is grouped in *tuples*, also known as *records*. Tuples represents an item in the DB and the information about that item. Tuples are organised as a set of named properties called *attributes*. Groups of tuples that share the same set of attributes are called *relations*. Usually relations are described as tables organized into rows and columns. The rows represent the tuples and the columns represent the matching attributes between the tuples. To reduce redundancy and improve data integrity, data in relational databases is stored in *normalized form*. Globally, database normalization is achieved by organizing the layout or *schema* of a database across different tables with specific restrictions applied. To link data in different tables together, *primary keys* and *foreign keys* are used. Primary keys

are used to uniquely identify a row, while foreign keys indicate that a row is linked to a row in another column.

Database management systems To make sure a database remains an *organized* collection of data, a Database Management System (DBMS) is used when interacting with a database. A DBMS is defined by Connoly and Begg (26) as a "software system that enables users to define, create, maintain and control access to the database". DBMSs provide a wide range of functionality such as processing queries over the data, handling inserts and updates, or managing and enforcing user access policies. Furthermore, DBMSs generally guarantee certain properties over the data in the database to provide users of the database management system with valid, predictable view of the data even in the presence of errors, power failure, or other problems that may arise. DBMSs that focus on relational database are often called relational database management systems (RDBMS). This is the type of DBMS that will be focused on in this research.

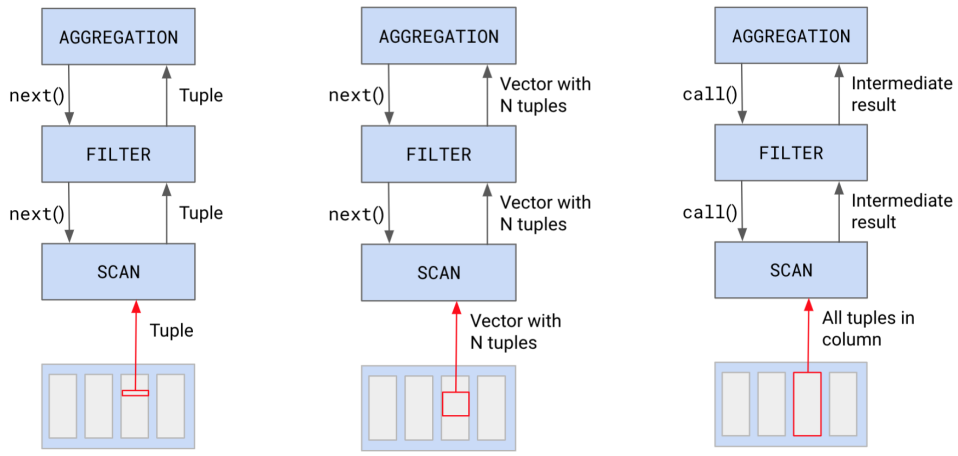
SQL Most modern RDBMSs allow querying and data manipulation through the Structured Query Language (SQL). SQL was one of the first commercial query languages to (largely) follow the relational model by Codd (25) and has become by far the most popular query language for RDBMSs. SQL allows many different types of queries, such as simply inserting, updating and deleting data, but also formulating complex analytical queries to extract data from the database.

Workload types While many RDBMSs adhere largely to the SQL specification and will therefore be able to run the same queries, RDBMSs are often optimized for specific types of queries. The main types of workloads an RDBMS tends to optimize for are *OnLine Transactional Processing* (OLTP) and *OnLine Analytical Processing* (OLAP). OLTP workloads are characterized by high numbers of transactional statements such as row-inserts, -updates, or -deletes. OLAP workloads are characterized by queries that require scanning large parts of the data, doing large grouping and aggregation, and performing large joins. Typically this can be summarized as OLTP being write-heavy, with OLAP being read-heavy. Additionally, a new category has risen combining the two: *Hybrid OLTP and OLAP* (HTAP). DuckDB can be categorized as an HTAP RDBMS with a focus on OLAP performance. For our research we will adapt this goal of focusing on OLAP performance while maintaining the possibility of adding OLTP functionality.

Storage layouts Data stored in an RDB can be stored in different layouts. Which layout is used to store the database is generally determined by the choice of RDBMS and not exposed to a user interacting with the database over SQL. Still, the choice of storage layout can have a significant impact on the performance characteristics of a database system as a whole. Traditionally, the most popular storage layout is called *row-store* layout. In the row-store layout, data for each table is stored as a contiguous array of rows with each row containing all attributes of that row. The row-store layout is generally used in DBMS optimized for OLTP workloads. An alternative storage layout that has quickly gained popularity is the *column-store* layout. This layout was popularized by systems such as MonetDB (22) and C-store (27). The column-store layout, often supplemented with column-wise compression (28)(29), has become the leading format for OLAP workloads in all commercial data analytics platforms (30).

Execution models To execute queries, DBMSs generally follow the same basic pattern. Firstly, the SQL query is parsed into a parse tree. Then, a query planner creates a *logical query plan* which describes which logical operators should be used to perform the query. Now an optimizer will analyze the logical query plan and convert it into a faster but logically equivalent query plan. Then, the query execution engine takes the logical query plan and converts it into a *physical query plan* which which describes that actual operations that will be executed. Finally, the physical query plan is executed by the execution engine to run the actual the query on the hardware. For the execution step of the query processing process, different categories approaches exist. We categorize these approaches into 4 main categories.

Volcano iterator This model was introduced in 1994 (31) and is the execution model used by traditional row-based DBMS such as MySQL and Postgres. In the Volcano iterator model, each operator implements a `next()` function that returns the next tuple in its (intermediate) result set. The query is executed by calling the `next()` function on the root operator until all tuples in the result set have been exhausted. Note that this root `next()` function recursively calls the `next()` functions of its child operators. Also note that the amount of `next()` calls is equal to the sum of all input, output, and intermediate result tuples. With this amount of `next()` calls, and the fact that each call will typically require interpretation of some query expression, the Volcano iterator model can cause significant execution overhead. An example query plan using the Volcano iterator model is shown in Figure 2.1(a).



(a) Tuple-at-a-time (b) Vector-at-a-time (c) Column-at-a-time

Figure 2.1: DBMS execution models

Column-at-a-time The second model we distinguish is the column-at-a-time model. This model is used by databases such as MonetDB (22). This model reduces the interpretation overhead associated with the volcano iterator model and allows more opportunities for CPU optimized code. In this model each column-wise operator is executed entirely, instead of calling `next()` for each tuple in the result set. By executing the operations in one go, typically in one loop, the execution overhead that the Volcano model suffers from vanishes. Additionally, using large loops in the operators opens the door to many compiler optimizations such as loop unrolling and auto-vectorization. While the column-at-a-time execution model produces CPU-efficient query execution code, the operators require full materialization of intermediate results, which can result in high memory consumption. An example query plan using the column-at-a-time model is shown in Figure 2.1(c).

Vector-at-a-time The third model is the vector-at-a-time, or vectorized execution model introduced by Boncz et. al. (17) in the VectorWise system. This is the model used by DuckDB (24), the DBMS that is used in our research. This model combines the concepts from the column-at-a-time and the volcano iterator model. In the vectorized model operators are similar to the volcano model where `next()` is called recursively, however instead of returning a single tuple from the result set, multiple tuples are returned. This approach profits from the same low interpretation overhead and CPU optimization opportunities as the column-at-a-time model, but does not suffer from the large intermediates that can

result in a memory bandwidth bottleneck. An example query plan using the vectorized model is shown in Figure 2.1(b).

Compiled execution The final model we discuss is the compiled execution model. In this model an alternative approach is taken to mitigate the large interpretation overhead caused by the tuple-at-a-time vulcano model: JIT compiling (parts of) the query into a routine that gets executed. This approach is pioneered by HyPer (23). While JIT compiled query execution can produce efficient code, it complicates the design and dependencies of the DBMS as it must include compiler infrastructure. Additionally, JIT compilation introduces significant latency for each query.

Compression Compression is a common techniques in many types of computer systems. In databases it was traditionally used to reduce the volume of data stored on disk. Common compression schemes used for this purpose are commonly general-purpose schemes like LZ77. In columnar databases however, more efficient light-weight compression schemes have been developed. These schemes exploit the fact that data from the same column tends to be strongly correlated and is therefore easier to compress. These light-weight compression techniques can be an order of magnitude faster than general purpose compression schemes. These schemes are in fact so efficient, that modern in-memory execution techniques use the compression schemes to compress data stored in main memory (32). Together with *compressed execution*, where data remains compressed throughout the query execution process, this can significantly reduce the memory bandwidth and increase performance.

We briefly go over the most common light-weight compression schemes that are used in columnar databases. Run Length encoding (RLE) is a compression techniques that is efficient for data (columns) that has many adjacent duplicate values. It works by replacing repeating values in the data by a single instance of the value, combined with the run length. For example, the RLE encoding of the string "AAAAABBCCCCC" would be "5A2B5C". Patched Frame-of-Reference (PFOR) (29) is a compression technique that is suitable for data that has good data distribution locality. Data is encoded as a difference to a base value. This base value is then encoded once for every piece of the data (for example, for each page). If a value happens to be below the base value, it can be stored as an exception. PFOR-DELTA (29) is similar to PFOR, but here the values are stored as the difference to to previous value, again with a base value for each piece of data. This scheme is highly efficient for ordered data. Finally, PDICT (29) is most suitable for data of which the

distribution is dominated by a small set of frequent values. In PDICT data is stored as a separate dictionary and a list of pointers into the dictionary.

2.2 Encryption

In cryptography, encryption is the process of converting information or data into a code to prevent unauthorized access. In its most general definition, an encryption function takes a key and a plaintext to produce a ciphertext. The plaintext is the data in its unencrypted form, for example an array of characters. The ciphertext is an encoded version of the same data that should be unintelligible to any unauthorized actor. To reverse the encryption operation, one passes the ciphertext and the key into the corresponding decryption function to obtain the plaintext. Encryption schemes can be either *symmetric* where the same key is used for encryption and decryption:

$$\begin{aligned} Encrypt_{symmetric}(key, plaintext) &= ciphertext \\ Decrypt_{symmetric}(key, ciphertext) &= plaintext \end{aligned} \tag{2.1}$$

or *asymmetric*, which uses two separate keys. These keys are often referred to as the private and public key:

$$\begin{aligned} Encrypt_{asymmetric}(key_{public}, plaintext) &= ciphertext \\ Decrypt_{asymmetric}(key_{private}, ciphertext) &= plaintext \end{aligned} \tag{2.2}$$

While there exist many different symmetric and asymmetric schemes, each with totally different characteristics, in general it holds that symmetric schemes tend to be significantly faster than asymmetric encryption schemes. For this reason, in encrypted databases symmetric encryption schemes are often preferable over symmetric schemes for the bulk of the work if possible. A common pattern in systems using encryption is to use asymmetric encryption to exchange the key for a symmetric encryption which is then used to encrypt the data that needs to be transmitted.

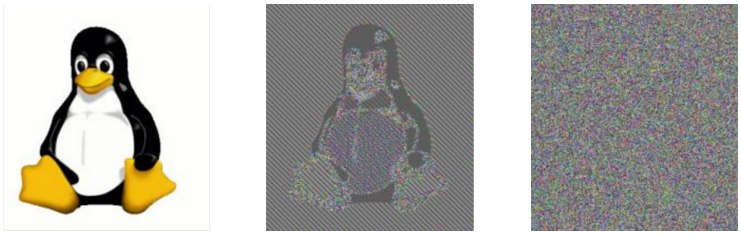
Randomized encryption It is important to understand that encryption as defined before is not sufficient to provide confidentiality. The problem is that the encryption process is deterministic, meaning that two identical plaintexts encrypted with the same key will always produce the same ciphertext. This determinism opens the door to inference attacks where an attacker can derive information from the ciphertexts by frequency and equality analysis. To solve this, encryption algorithms will often take a random value

called an *initialization vector* (IV) or *nonce*. The IV will need to be stored along with the ciphertext to be able to decrypt later.

$$\begin{aligned} Encrypt_{probabilistic}(key, plaintext) &= \{ciphertext, IV\} \\ Decrypt_{probabilistic}(key, ciphertext, IV) &= plaintext \end{aligned} \tag{2.3}$$

Authenticated encryption While applying a strong randomized encryption algorithm to encode information will provide data confidentiality, it does not provide authentication of data. In the context of encryption, authentication of data is the guarantee that the message was not modified after encryption. In an unauthenticated cryptosystem, an attacker could modify intercepted ciphertexts to manipulate the system. This ability allows a type of attack called chosen-ciphertext-attack, through which an attacker can reveal (parts of) the encrypted data or even retrieve the key. Authenticated encryption schemes solve this problem by adding an integrity check called the authentication tag or message authentication code (MAC). Authenticated encryption can be achieved by combining a non-authenticated encryption scheme with a secure MAC function, or by using an encryption scheme with a built-in authentication step such as AES-GCM.

Encryption modes Symmetric encryption schemes exist in two main types: stream ciphers and block ciphers. Stream ciphers are conceptually very simple and use a pseudo-random function(PRF) based on the key to generate a keystream that is xor-ed with the plaintext to obtain the ciphertext. The decryption function is is identical to the encryption function as xor-ing the ciphertext with the keystream will return the plaintext. Block ciphers are the most ubiquitous in practice today and use a different approach where the plaintext is divided into blocks of *blocksize* bits for encryption. Each block of the plaintext is encrypted separately with the key into similarly sized blocks of ciphertext. Note that by default each ciphertext only depends on the key and the corresponding plaintext block. This means that when identical blocks arise in the plaintext, identical ciphertexts are produced. This principle is visualized in Figure 2.2, where the non-diffusing mode AES-ECB is used to encrypt an image of tux, the Linux mascot. As can be clearly seen, some information on the original contents is still clearly visible. Another issue that block ciphers have is that plaintexts may not always be a multiple of *blocksize*. This can become a security problem when insecure padding schemes are used. To be able to use a block cipher securely on plaintexts with a different size than precisely *blocksize*, block ciphers always require a block cipher operation mode. The operation mode algorithm will handle the masking of patterns and padding problems. Additionally, these block cipher mode algorithms allow



(a) Plaintext (b) AES-ECB (c) AES-CBC

Figure 2.2: Naive usage of block ciphers does not lead to confidentiality (1)

passing of an initialization vectors for randomization or allow for authenticated encryption through calculation of a checksum over the encrypted data.

Property preserving encryption For most use-cases, the goal of encryption is to achieve semantic security. A system is semantically secure if only negligible information on the plaintext can be derived from the ciphertext. In the context of encrypted query processing, however it can be useful to relax the security requirements and reveal some information about the data. The information that is leaked can then be used to allow operations over the encrypted data. The type of encryption where some information on the plaintext is deliberately leaked is called property preserving encryption (PPE). The simplest form of PPE is to use deterministic encryption (DET). As mentioned before in section 2.2, DET reveals equality between encrypted values which allows equality comparison at the cost of potential vulnerabilities to inference attacks. A relatively new category of PPE schemes is Order-Revealing Encryption (ORE). With ORE, there exists a function with which the relative order between two can be determined. The simplest form of ORE is called Order Preserving Encryption (OPE). With OPE, Ciphertexts can be directly compared on their order as if they are unencrypted. While very simple to implement, OPE does raise serious security concerns when used in an EDBMS context (33). Recently there has been work exploring more complex types of ORE that offer better security at the cost of a more complex comparison function (34)(35).

Homomorphic encryption Homomorphic encryption (HE) is a type of encryption that allows computation over ciphertext without the need for decryption. A distinction can be made between two main categories of HE: Fully Homomorphic Encryption (FHE) and Partially Homomorphic Encryption (PHE).

FHE In FHE schemes, arbitrary computation is possible on ciphertexts. In formula 2.4 a simple formula describing this property is shown with f being an arbitrary computation and f_* being a corresponding function that performs the same computation as f but operating over the encrypted argument. For FHE there exists an $f_*(x)$ for every $f(x)$.

$$Decrypt_{fhe}(f_*(Encrypt_{fhe}(plaintext))) = f(plaintext) \tag{2.4}$$

FHE was first theorized in 1978 (36) and has for over 30 years remained one of the unreachable holy grails of cryptography. In 2009 however, Gentry et. al. demonstrated a FHE encryption scheme (37). Today, improved FHE schemes have been published and several open source libraries exist implementing FHE schemes (38)(39). However, for most purposes, including EDBMS, FHE remains impractically slow with overheads of many orders of magnitude for both encryption, decryption and computation over ciphertext.

PHE A more lightweight alternative to FHE is Partially Homomorphic Encryption (PHE). PHE schemes support, as the name implies, only a limited type of computation over encrypted data. With this limitation to the functionality comes a massive improvement in performance. A well known PHE is the Paillier cryptosystem. Paillier is an asymmetrical scheme with the following homomorphic property:

$$Enc_{Paillier}(plaintext_1) * Enc_{Paillier}(plaintext_2) = Enc_{Paillier}(plaintext_1 + plaintext_2) \tag{2.5}$$

Due to this property the sum of two encrypted values can be obtained by multiplying the ciphertexts, followed by a decryption of the result. This property is very useful for a variety of applications, for example in an EDBMS it can be used for addition operators or aggregations. For other operations such as multiplications, other PHE schemes exists.

2.3 Defining the limits of secure systems

To understand the limitations of a secure system, the designers of the system generally define the limitations through a threat and a trust model. These threat and trust models describe the types of attacker that the system is secure against and which parts of the system can be compromised in an attack without failing the promised security guarantees.

Trust models In the trust model, the parts that can be compromised by an attacker are defined as *untrusted*, while the parts that will lead to secure system failure are defined as *trusted*. The set of all trusted components of a system is called the Trusted Computing Base (TCB). Let us consider the classic use case of an outsourced database. The trusted components would be all client-side hardware and software: the machines used by the users to query the database can not be compromised as their compromise would make the system fail. The untrusted component would consist of all server-side hardware and software. Note that a secure system generally requires at least 1 trusted component since a system without a trusted component would not be able to communicate trusted information with its users.

Threat models In secure systems research and engineering, it is essential to clearly define the capabilities of a system attacker. In the field of EDBMS there are three main attacker types that are distinguished:

Passive snapshot attacker The weakest type of attacker is the passive snapshot attacker. Passive, also known as Honest-but-curious (HbC) in this context, means that the attacker will not actively attempt to compromise the system, for example by manipulation of network packets or modifying data stored on disks. Snapshot means that this type of attacker can only view the state of the untrusted parts of the secure system once. An example passive snapshot attacker is an adversary who can obtain a single memory dump of the secure application.

Passive persistent attacker A passive persistent attacker is similar to the passive snapshot attacker with one exception: instead of a single view of the secure system state, has a persistent view of the secure system state. The persistence generally means that the adversary has much more attack capabilities as the temporal dimension of the information leaked can reveal much useful information.

Active attacker The active attacker, also known as a malicious attacker, has all the capabilities of a passive persistent attacker with the addition that the active attacker can actively try to compromise the system. This capability makes the attacker significantly harder to protect against.

3

Related work

In this chapter, we will provide an analysis of the most relevant existing research in the field of EDBMSs. First, an overview of the research field is given based on an analysis of the secondary literature. Then, the research field of encrypted database systems is divided in two main categories: Cryptography-based and Trusted Hardware-based. A detailed categorization and reasoning behind this categorization is discussed in section 3.1. Then, for each category the most relevant work is discussed in sections 3.2 and 3.3. Finally, in section 3.4 the industry adoption is analyzed and an overview of open issues preventing widespread EDBMS adoption is given.

3.1 Research field overview

The problem of creating a DBMS capable of operating over encrypted data is considered to be one of the holy grails in the field of database security. It is still very much an open issue as currently, no production-ready systems exist that offer similar functionality and performance as traditional unencrypted DBMSs. The problem of encrypted relational databases was first described in 2002 by Hacigümüs et al. (15). In this highly influential paper the authors recognized the privacy problems caused by the emergence of cloud computing and the Database-as-a-Service model(DBaaS). They were one of the first to explicitly consider a database outsourced to an untrusted server, a model that is fundamental to the field of EDBMSs today. Their proposed approach to achieve privacy in the DBaaS model with an untrusted server was to split the query execution between the trusted client and the untrusted server. Around 2011, influential works such as CryptDB (3), TrustedDB (40) marked a beginning of a fast growing research field. With the rapid growth of the DBaaS model and cloud computing over the past two decades, the field of EDBMS research grew

along with it. To this day many different solutions have been proposed based on a variety of both software and hardware primitives.

3.1.1 Secondary literature

Firstly, an overview of the most relevant secondary literature is given. Due to the fast growing nature of the field of EDBMS research, only recent surveys were included. The goal of this is to get a better understanding of the types of solutions that have been proposed to this date, and how these solutions are generally categorized.

Kohler et al. (2015) Kohler et al. published a work with the goal of creating a taxonomy of confidentiality preserving DBaaS approaches (41). In their work, the authors analyze a variety of software-based techniques such as Private Information Retrieval (PIR), Oblivious RAM (ORAM), (Partial/Fully) Homomorphic Encryption (PHE/FHE) and split client/server query execution using bucketization as used by Hacigümüs et al. (15). Kohler et al. define these primitives as confidentiality-preserving indexing approaches (CPIs). CPIs can be used to secure specific DBMS functionality such as range queries, string pattern matching, or updates. The authors conclude that the functionality, leakage patterns and performance penalties of the different CPIs varies widely and no silver-bullet CPI has been found or seems likely to be found in the near future. The choice of which CPIs to use for building a secure system will strongly depend on the system requirements.

Saleh et. al. (2016) Saleh et al. published a high-level survey that analyzes the state of the art in query processing over encrypted data. The authors state that the field of processing over encrypted data is yet to be well established. The authors proceed to categorize the state-of-the art in processing over encrypted data in three main categories. These categories contain several subcategories and are visualized in figure 3.1. The authors conclude their research with a listing of several open issues in the research field. For their first category, FHE, they conclude that it is remains infeasible due to performance limitations. For PHE and OPE schemes, the limitations to supported functionality and the inherent leakage of OPE schemes are mentioned as the main issues. Finally for TH the authors state that the primary issue is the high cost and requirement for dedicated hardware.

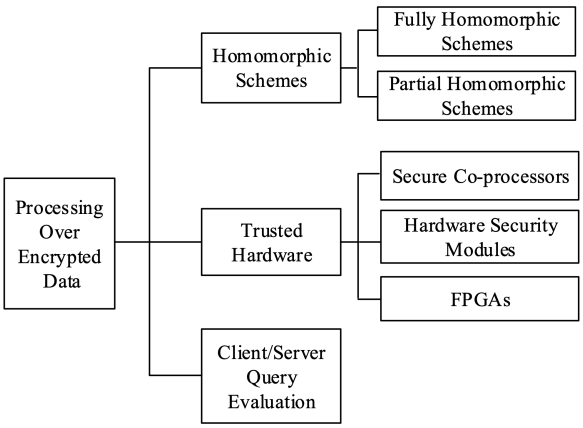


Figure 3.1: Main categories in EDBMS literature according to Saleh et al. (2)

Fuller et. al. (2017) Fuller et. al. (42) published an in-depth survey of both protected search primitives (comparable to the CPI definition from Kohler et. al.) and full systems implemented using these primitives. In their work, the authors cover only cryptographic approaches to EDBMSs and EDBMS primitives, and define TH based solutions as out-of-scope. Fuller et al. categorize their cryptographic primitives into main categories: *Legacy*, *Custom*, and *Oblivious*. The *Legacy* category is similar to the *Partial Homomorphic Schemes* category from Saleh et. al. and covers techniques such as AHE, OPE and DET. The *Custom* category contains a set of techniques using special indexing structures to achieve various levels of security. All techniques in this category are based on relatively new research from between 2013-2016. Finally, the *Oblivious* category contains techniques that aim to hide common results between queries. The approaches in this category use ORAM-like techniques to construct schemes supporting database functionality.

3.1.2 Categorizing techniques

In this section, we will define the categorization of the available techniques for our related work based on the surveys from section 3.1.1 and the latest developments in the field. The resulting categorization is shown in figure 3.2. This categorization is the combination of the categorization by Saleh et al. and Fuller et al. with two main modifications. Firstly, the trusted hardware category is divided into two main categories: dedicated hardware and trusted executions environments. At the publication dates of the work by Saleh et. al., Trusted execution environments were only just beginning to come to the attention of researchers. However, trusted execution environments, and more specifically Intel SGX, can be considered the main focus in EDBMS research. The second modification is the

breaking up of PPE schemes into a separate category. This separation is motivated by the fact that PPE schemes share characteristics on inherent leakage that are very important to their applicability in EDBMS design. PHE techniques found in EDBMS literature such as the Paillier and ElGamal schemes, can provide semantic security and therefore do not suffer from these problems.

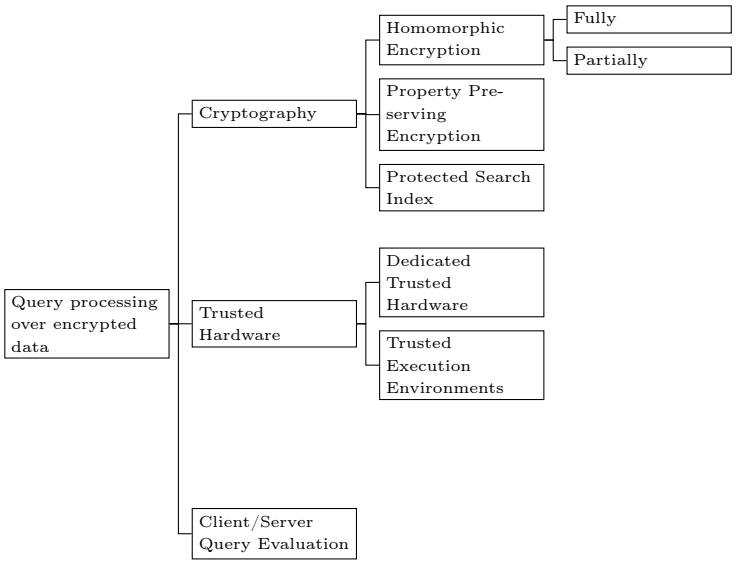


Figure 3.2: Categorization of EDBMS techniques

3.2 Cryptography based

In this section, the most relevant research in cryptography-based EDBMS literature is covered. Using cryptographic approaches to secure outsourced databases is very attractive with respect to the trust model: for the system to be secure the only components that need to be trusted are the client holding the key and the cryptographic schemes that are used. However, the rich functionality required by modern DBMSs is far from trivial to implement using currently available cryptographic schemes. While cryptographic techniques supporting arbitrary computation over encrypted data do exist, as discussed in section 2.2, these techniques suffer from high overheads rendering them infeasible for use in EDBMSs. In this section, we will go over the most relevant work in two types of cryptographic techniques for EDBMSs that are practically feasible.

3.2.1 PHE+PPE

The first category of cryptography-based EDBMS that will be covered, actually combines two of the categories in from figure 3.2: PPE and PHE. The solution proposed in these works uses the combination of PPE and PHE to support a wide range of query processing functionality. Note that PPE and PHE schemes are explained in detail in sections 2.2 and 2.2. The advantage of this approach is that it allows easy integration of existing DBMS. The downside of using PHE and PPE schemes are the inherent leakage of the PPE schemes and the computational overhead of the PHE schemes. Mitigating these issues is the main focus in these works.

3.2.1.1 CryptDB

In 2011, Popa et. al. (3) were the first to demonstrate how a combination of PHE and PPE schemes can be used to build an EDBMS that supports a significant subset of SQL operations over encrypted data. With CryptDB, the authors showed that by combining a server-side proxy with custom User Defined Functions (UDFs) an EDBMS can be built on top of a regular unencrypted DBMS. In figure 3.3 the global architecture of CryptDB is shown.

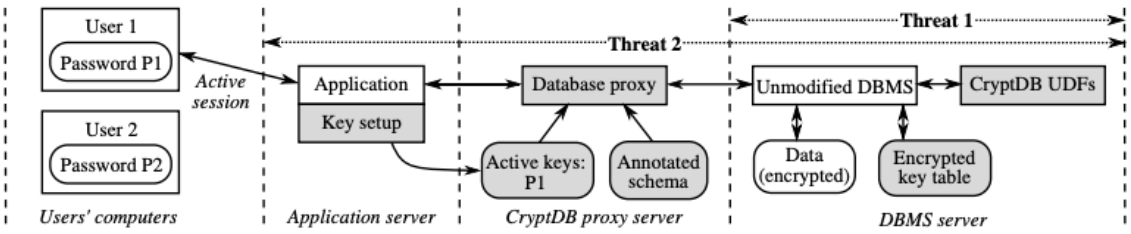


Figure 3.3: CryptDB architecture (3)

Supporting SQL operations with PPE and PHE To understand how CryptDB uses PPE and PHE to run SQL queries on encrypted data, we will look at a simple example. Let's say we have an instance of CryptDB running with a TPC-H (43) database as data. Now we want to run a query on the `orders` table to calculate our total revenue up to a certain date. To do so, the user sends the following query to the application server:

```
SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate < CAST('1995-01-01' AS DATE);
```

The application server forwards the query to the proxy server, which rewrites the query. For the predicate in the **WHERE** clause this means the constant **CAST**('1995-01-01'**AS DATE**) is encrypted with an OPE scheme and can than be compared to an OPE encrypted version of the o_orderdate column by regular query execution. The **SUM** operation is rewritten to use the CryptDB UDFs to do homomorphic addition using a Paillier (44) encrypted version of the o_totalprice column. The query result is returned from the DBMS to the proxy which decrypts the result and returns it to the application server which in turn sends it to the user.

Functionality supported CryptDB supports many different SQL operations through various PPE and PHE schemes. an overview is given in table 3.1. In general, DET and OPE schemes are used for evaluating search predicates on fixed-length datatypes, the searchable encryption scheme by Song et. al. (45) is used for search predicates on strings, and the PHE schemes Paillier and ElGamal are used for arithmetic and aggregation. While the schemes used by CryptDB support many types of queries, there are limitations to the queries CryptDB can process. Firstly, the SEARCH scheme used for the LIKE operator only supports single-word search. Secondly, queries that do both computation and comparison in a single query such as **WHERE** salary > age*2+10 are not supported, as these would require a single scheme that supports both order comparison and additive homomorphic properties. Despite these limitations, the CryptDB supports all functionality in the TPC-C benchmark. In an experiment based on a real-world transactional data trace, the authors demonstrated that CryptDB can support operations on over 99.5% of all encrypted columns.

Type	Scheme	Supported Operations
Randomized Encryption	AES CBC	SELECT, UPDATE, DELETE, COUNT, INSERT
Deterministic Encryption	AES CMC	equality comparators (=, !=, <>, etc.)
Partially Homomorphic Encryption	Paillier (44)	GROUP BY SUM, addition
Partially Homomorphic Encryption	ElGamal	multiplication
Property Preserving Encryption	OPE (46)	MIN, MAX, ORDER BY, order comparators (<, >=, etc)
Searchable Encryption	SEARCH (45)	LIKE, equality comparators (=, !=, <>, etc.)

Table 3.1: CryptDB Encryption schemes with their respective functionalities (3)(11)

Adjustable query-based encryption One of the main concepts in the CryptDB design is its adaptive encryption scheme. To understand the need for this adaptive scheme we should first consider two problems that arise from the encryption schemes used by CryptDB. The first problem of PHE-based systems is the inherent information leakage of their encryption schemes. For some encryption schemes used by CryptDB, this data leakage is very significant. By definition, an OPE scheme leaks the order of all the data, which has been shown to open vulnerabilities to inference attacks (33). Furthermore, DET encryption schemes, by definition, leak equality of encrypted data which also opens the door to similar types of attacks. The second problem is that of storage overhead, especially the overhead caused by Paillier encryption. Paillier has a ciphertext size of 2048bits, which can results in a storage overhead of 32x for 32bit integers.

With these two problems in mind, we can clearly see that a naive implementation that encrypts every database column with every encryption scheme is problematic, as it would result in a very large storage overhead and would leak order information on every column. To mitigate this, CryptDB uses a-priori knowledge of the query load to adjust the selection of encryption schemes for each column to match the query load. For the exact implementation details we refer to the original paper. With the adjustable query-based encryption, CryptDB manages to significantly decrease the storage overhead and data leakage. It should be noted that for systems with unpredictable queries, as is often the case in analytical workloads, this system will be either less effective, or very limiting to which queries can be run.

3.2.1.2 Monomi

In 2013, Tu et. al. published a work presenting Monomi, an EDBMS that builds on the foundation laid by CryptDB. The goal of Monomi was to build a system similar to CryptDB but with a focus on OLAP workloads instead of OLTP workloads. Tu et. al. noted that CryptDB lacked crucial functionality for analytical workloads, only supporting 4 of the 22 queries in the TPC-H benchmark suite, with a median slowdown of 3.5×. With Monomi, the authors aimed to both support more of the TPC-H queries and reduce the overhead. The main concepts from CryptDB were directly adopted, with some modifications and optimizations. The system architecture with a trusted proxy and an unmodified DBMS running special UDFs as shown in figure 3.3 was adopted more or less unchanged. All encryption schemes chosen by CryptDB as shown in table 3.1 were also used. Monomi differs from CryptDB in two main ways. Firstly, they add the concept of spliting client/server

execution to support more complicated queries. Secondly, several practical optimization techniques are introduced.

Splitting client/server query execution The concept of splitting query plans across an untrusted server and a trusted client or proxy is based on earlier work from Hacigümüs et. al. (15)(47)(48) However, where Hacigümüs et. al. used it as the foundation for secure processing, in Monomi it is used as a fallback for queries that cannot be fully processed through the use of PHE and PPE schemes. To illustrate how Monomi uses split client/server execution to extend functionality let us consider how Monomi would process TPC-H Q11:

```
SELECT   ps_partkey,
         SUM(ps_supplycost * ps_availqty) AS value
FROM     partsupp JOIN supplier JOIN nation
WHERE    n_name = :1
GROUP BY ps_partkey
HAVING   SUM(ps_supplycost * ps_availqty) > (
    SELECT  SUM(ps_supplycost * ps_availqty) * 0.0001
    FROM    partsupp JOIN supplier JOIN nation
    WHERE   n_name = :1 )
ORDER BY value DESC;
```

TPC-H Q11 has multiple incompatibilities with the encryption schemes used by Monomi. Firstly, there is no efficient encryption scheme that supports multiplication over encrypted values[1]. Secondly, the query requires checking whether some `SUM()` of each group is greater than a sub-select expression which then computes its own `SUM()`. The combination of the summation and the comparison are incompatible as they require different encryption schemes. To be able to answer a query such as TPC-H Q11 while maintaining confidentiality on the untrusted server, Monomi uses a query planner that automatically splits the query across the local (trusted client-side) and remote (untrusted server side) domain. An example query plan is shown in figure 3.4. For more details on the query planner algorithm we refer to the original paper.

Optimization techniques Both the original CryptDB design and the introduction of the split client/server query execution, suffer from several serious performance issues. To minimize the overhead for analytical workloads, Monomi implements several optimization techniques.

[1]Note that this is a claim made by the Tu et. al., but multiplicative homomorphic encryption schemes do exists and a paper by Popa et. al. present these schemes as viable (11)

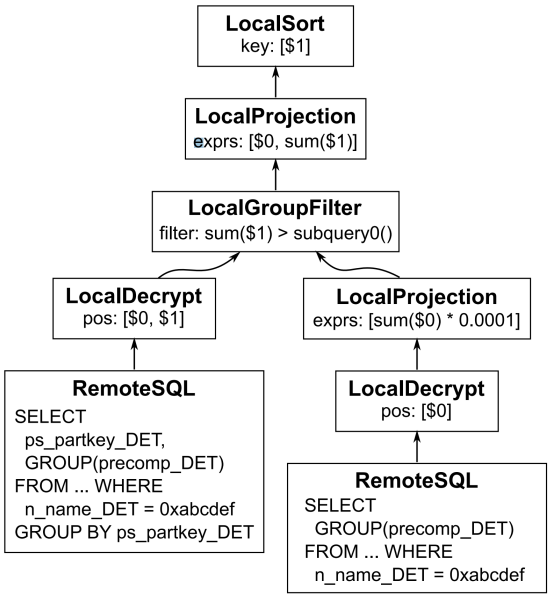


Figure 3.4: Example query plan from Monomi query planner for TPC-H Q11 (4)

Precomputation The first optimization presented is precomputation. Precomputation is used to speed up certain queries by minimizing the need for client-side execution. For example if we consider TPC-H Q11 again, Monomi will precompute `ps_supplycost * ps_availqty` to allow calculating the SUM using Paillier on the server. To determine which columns should be precomputed, the Monomi designer is used, which is explained in the last paragraph of this section.

Space-efficient encryption The second optimization implemented by Monomi is the use of space-efficient encryption methods. As mentioned in section 3.2.1.1, Paillier operates over very large ciphertexts which can result in large overheads if implemented naively. CryptDB worked around this by limiting the amount of columns encrypted with Paillier, leaving further optimization to future work. Since an analytical workload generally depends heavily on aggregates of large parts of columns, Monomi implements an optimization by Ge et. al. (49) which allows packing values from both multiple rows and multiple columns in a single field. This reduces per-row space overhead by 90% for a single 64-bit column encrypted with Paillier. Another significant source of storage overhead due to ciphertext expansion is caused by the DET encryption schemes. Using normal block cipher modes such as AES or blowfish would produce ciphertext sizes of 128bits, resulting in large overheads for smaller data types such as 8-bit integers. To solve this, Monomi uses the AES FFX

block cipher (50), a format-preserving encryption scheme. This results in a 33% lower overhead for the lineitem table of the TPC-H datbase scheme.

Packed Pallier encryption The third optimization applied in Monomi improves the efficiency of queries that aggregate multiple values per row. For example, consider TPC-H Q1:

```
SELECT SUM(l_quantity), SUM(l_extendedprice), ..
FROM lineitem WHERE .. GROUP BY ...
```

Query 1 has total of 7 aggregates over different columns. If implemented naively, this would mean that the Paillier-sum would be encrypted 7 times, which can become computationally expensive as calculating the the Paillier-sum requires multiplication modulo a 2048-bit public key. Fortunately, the method from Ge and Zdonik (49) allows for calculating the sum of multiple packed values in a single computation. Using this method, Monomi ensures that the number of Paillier-sums for a query such as TPC-H is limited to one Pallier computation per row.

Prefiltering The final optimization introduced by Monomi is conservative pre-filtering. Analytical workloads often scan through large parts of a column or table but return only a small part of the data by applying a filter. For Monomi this means that if the filter cannot be calculated server-side, large amounts of data will need to be transferred to the client for client-side filtering. This can result in infeasibly large overheads. To mitigate this, Monomi will generate a conservative estimate of a filter that can be executed on the untrusted data. For more detail on how Monomi generates these filters, we refer to the original paper.

Query planner & designer Similarly to CryptDB, the Monomi design includes a designer to determine the physical database design for a certain workload. The user provides a set of example queries, the original database scheme, and a space constraint. The designer will then estimate an optimal physical design. This design will include which columns are encrypted with which encryption scheme and which columns should be precomputed. Note that the Paillier packing optimization is also applied here as it will look at each query and decide which columns should be encrypted together into a single value. The Monomi query planner is fundamentally not very different from a non-encrypted RDBMS query planner in that it uses a cost model with cardinality estimation estimate the most efficient query

plan. The main difference is that the query plan is split across a local and a remote execution engine, so the cost model needs to include the transfer time of data over the network, client side execution, and decryption time.

3.2.1.3 Seabed

In 2016 Papadimitriou et. al. (20) presented Seabed, an OLAP-oriented, Apache Spark-based (51) system that is largely based on CryptDB and Monomi. Papadimitriou et. al. observed that the approach taken by Monomi does not scale well for analytical workloads. Monomi may achieve real time querying on medium-sized datasets of several gigabytes, but for large, multi-terabyte datasets even simple queries would take hundreds of seconds. Another issue identified by Papadimitriou et. al. is that of frequency attacks. Research published in 2015 by Naveed et. al. (33) demonstrated that the PPE schemes used by CryptDB and Monomi pose serious security threats. Even though intuitively OPE and DET may seem to leak little useful information, Naveed et. al. showed that for some real-world dataset, very significant amounts of encrypted data can be fully attained through inference attacks. The main goal of Seabed is to mitigate both of these identified issues and implement them into an Apache Spark based prototype. The overall architecture of Seabed is similar to CryptDB and Monomi with an untrusted server running a database system and a trusted client-side proxy. The main difference with CryptDB and Monomi lies in the cryptographic schemes used. In the Seabed paper, two new encryption schemes are introduced that are designed specifically to mitigate the aforementioned issues: ASHE and SPLASHE, described next.

ASHE Additively symmetric homomorphic encryption (ASHE) is a symmetric, encryption scheme based on a cryptographically secure pseudo random generator (PRF). By using a symmetric encryption scheme instead of an assymetric encryption scheme, ASHE manages to improve performance significantly. Both encryption speed as the addition operation on ciphertexts are significantly faster with ASHE than Paillier. There are however some limitations to ASHE. Contrary to Asymmetrical AHE schemes like Paillier, where ciphertexts have a fixed length, ASHE ciphertexts will grow as the number of additions increases. This means that for aggregations over large columns, the memory usage and network traffic from the ciphertext can have a significant performance impact. To deal with this, Seabed implements several encoding and compression techniques to make the ciphertexts as small as possible.

SPLASHE Splayed ASHE (SPLASHE) is the second encryption scheme introduced by the authors. SPLASHE is created to prevent the previously mentioned inference attacks on DET encryption. SPLASHE is based on ASHE and its concept is relatively simple. Consider a column C_1 which can take one of d discrete values. Now consider the query `SELECT COUNT(C1)WHERE C1=x;`. Using a traditional DET scheme as Monomi and CryptDB do, the proxy would have C_1 encrypted with DET and change the predicate in the query to `WHERE C1=DET(x)`. SPLASHE takes a different approach and replaces C_1 with a family of columns $C_{1,1}...C_{1,d}$. For each item in C_1 with some value y, $C_{1,y}$ will equal $ASHE(1)$ while all others equal $ASHE(0)$. Now the example query can be executed by changing the column to $C_{1,x}$ and removing the predicate altogether. Note that the basic SPLASHE described here will increase the storage required for C_1 by factor d. To mitigate this, Seabed has implemented a more advanced SPLASHE algorithm that uses a-priori knowledge of the workload to limit the storage overhead by not creating separate columns for all values of d. SPLASHE has some obvious drawbacks. Firstly, it depends strongly on the a-priori knowledge of the query workload. This is very similar to both Monomi and CryptDB who also optimize their database scheme for a specific workload. Secondly, SPLASHE may leak start leaking some information when the distribution of data added to a column changes dramatically.

3.2.1.4 Cuttlefish

In 2017, Savvides et. al. published a paper presenting Cuttlefish (52), another Apache Spark-based EDBMS. The goal of Cuttlefish is to allow unlimited query expressivity while offering the flexibility to work around the fundamental limitations of PHE schemes in an efficient way. Cuttlefish uses a combination of PHE and PPE based on previous work, where both the schemes used by Monomi and CryptDB are used, together with the ASHE and SPLASHE schemes introduced by Seabed. The architecture of Cuttlefish is similar to that of CryptDB, Monomi and Seabed in that a trusted component is required, which is implemented as a client-side proxy. A new insight by Savvides et. al. is the possibility that this client-side component can also be implemented in the server side using a TEE such as Intel SGX. The advantage of this approach is that for queries that need to do additional query processing at the client side, network traffic can be significantly reduced. In their evaluation both approaches are implemented.

The main contribution of Cuttlefish is the secure data type (SDT) annotation system. Cuttlefish abandons the goal of *transparency* offered by previous systems. Transparency in this context means that the systems supports standard SQL and a user does not need to be

aware of the encryption processes. By forcing users to annotate the database scheme with fine-grained restrictions, Cuttlefish is able to offer improved expressivity and performance. Example annotations are $range(from-to)$ to indicate that values fall in a certain range, $+$ for postive values or $enum(value1, value2, ...)$ for fields with a limited number of possible values. The annotations are used by Cuttlefish to perform a large variety of optimizations, such as the precomputation proposed by Monomi, to several new optimizations specific the the SDT annotation system.

3.2.1.5 Symmetria

In 2020, Savvides et. al. published another paper presenting Symmetria (5), yet another Apache Spark-based prototype. Similar to Cuttlefish, Symmetria offers full TPC-H and TPC-DS support. With Symmetria, the authors introduce two new PHE schemes: symmetric additive homomorphic encryption (SAHE) and symmetric multiplicative homomorphic encryption (SMHE). Similarly to the ASHE scheme from Papadimitriou et. al. (20), these schemes are symmetrical and sacrifice ciphertext compactness for performance. SAHE was designed specifically to offer similar performance to ASHE, but with greater query expressivity. SMHE was designed as the first symmetric multiplicative homomorphic encryption scheme since the authors of seabed only presented ASHE for additive homomorphic encryption. Symmetria's main contribution is the integration of these two new schemes into a EDBMS prototype along with a series of query optimizations and compaction techniques for reducing ciphertext size.

Symmetria is evaluated using the TPC-H and TPC-DS benchmarks. For their evalution, the authors rewrote Monomi based on Spark to allow for a direct comparison to previous literature. These results give a good insight into the OLAP performance of current state-of-the-art in PPE+PHE EDBMSs. In figure3.5 the relative performance of the Symmetria and the Monomi-like protoype called Asym are shown for all 22 TPC-H queries. The results for Symmetria with an abbreviation postfix are configurations with an increasing number of optimizations disabled in the order of which they are listed in the legend. The Y-axis shows the normalized overhead compared to baseline Apache Spark. The scale factor is 100 and the test setup consisted of 10 Amazon EC2 m5.2xlarge nodes. The results show a wide variety in overheads with several large outliers, most notably Q06 with an overhead of 2 orders of magnitude. Overall the average overhead is reported to be 5.35× for Symmetria and 20.39× for Asym. The relative storage overheads for the Symmetria and Asym prototypes are respectively 1.99× and 10.70×.

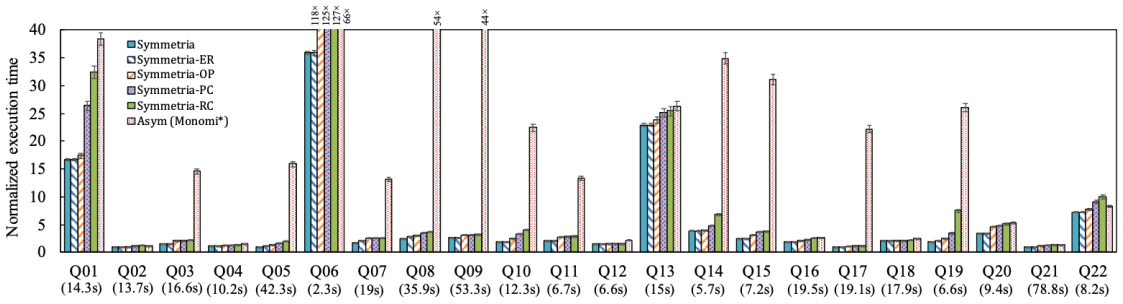


Figure 3.5: Symmetria & Monomi Apache Spark prototype performance (5)

3.2.2 Secure indexing

Besides the PPE+PHE based approaches covered in section 3.2.1, a range of other cryptographic primitives exists that have been researched in the context of EDBMS design. The most prominent of these primitives are based on Symmetric Searchable Encryption (53), Private Information Retrieval by keyword (54), Secure Multi-party Computation (55), and ORAM (56). While the techniques listed in this category have been successfully used to build systems with rich functionality close to that of unencrypted DBMS (57)(58), in general these systems are further from industry adoption than their PHE+PPE and TH counterparts for several reasons: Firstly, these systems tend to be more difficult to implement into existing DBMS systems, for example due to their execution model requiring two separate, non-colluding servers. Secondly, many of these techniques have limitations in which functionality can be supported, similarly to the PHE+PPE approaches from section 3.2.1.

3.2.2.1 Blind Seer

In 2014 Pappas et. al (57) published a work presenting Blind Seer. Blind Seer is a database prototype supporting boolean search queries. Its security properties and architecture are very different from the solutions presented so far. Blind Seer uses a three party setup which requires two non-colluding servers: a client sending the queries, a server holding the data, and an index server that aids in the secure querying process. Using this setup, Blind Seer achieves significantly higher security guarantees that PHE+PPE systems as it achieves not only data confidentiality, but also query confidentiality. In Blind Seer, data is indexed with a search tree where each node contains encrypted bloom filters storing a set of all keywords contained in their child nodes. To securely query the data, queries are

transformed into boolean circuits and the client and server jointly traverse the search tree using Yao's Garbled Circuits (59).

In their evaluation, the authors compare Blind Seer against MySQL on a 10TB database and demonstrate a 20% to 300% overhead. This evaluation does however depend on a high speed local network between the client, server and index server. Since the server and index server need to be non-colluding, this may not be the case in a practical setting where the non-colluding servers are separated by a lower bandwidth, higher latency network. Another possible problem with the Blind Seer system is the possibility for false positives due to the use of bloom filters, which is a tunable parameter set to 10^{-6} in their evaluation.

3.2.2.2 Arx

In 2017, Popa et. al. published a paper presenting Arx. Arx is a MongoDB-based EDBMS prototype but according to the authors, its design should apply to other databases as well. The main motivation behind Arx are the inference attacks that PPE based systems such as CryptDB have been shown to be vulnerable to (60)(33). To mitigate this problem, the PPE encryption schemes used in PPE-based EDBMS have been replaced by two new secure indexing structures: ArxEQ and ArxRange.

ArxEQ ArxEQ is used for equality queries and uses a regular index but encrypts the values with strong encryption before insertion. To prevent leaking the equality property to the server, each occurrence of an identical value in the index is concatenated with a counter value and hashed with a cryptographically secure hash function. These hashes are used as keys for the index. To be able to do lookups, the client-side proxy maintains a counter of the number of occurrences of each value in the index. To perform a lookup, the client-side proxy generates a list of all cryptographic hashes that have been inserted by looking at the counter value. Note that this results in the size of search tokens scaling linearly with the number of occurrences of a value in a column.

ArxRange ArxRange is used for range queries and replaces the ORE schemes in PHE+PPE EDBMS. It builds a tree over the relevant keywords and stores a garbled circuit (61)(59) at each node of this tree. The main challenge for this index is to prevent the communication steps as is present in a system such as Blind Seer. To achieve this, a technique from literature on Garbled RAM (62) is used to chain the garbled circuits in the tree together. This allows for a lookup throughout the entire tree in a single round of interaction. The garbled circuits stored in the nodes of the tree can be used only once: using the same

circuit twice would break security. Therefore, each index lookup is follow by a repair step: in this step the client proxy needs to send new garbled circuits to rebuild the index. Note however, that only *LogN* nodes need to be rebuilt for each lookup.

ArxAgg With the ArxRange index, a special optimization is possible for a common type of queries: ranged aggregation queries. This index is called ArxAgg and can offer significant performance improvements over the traditional Paillier addition used by PHE+PPE systems. ArxAgg works by storing a pre-computed aggregation result in the nodes of the ArxRange tree. A lookup in this index results in a set of *LogN* partial aggregates that are sent to the client-side proxy and the final aggregation can then be computed after decrypting the partial aggregates.

3.3 Trusted Hardware based

In this section, we will cover the second main category of EDBMS literature, those who base their security on a specially designed trusted component running in the untrusted server environment. The trusted hardware (TH) components used for the EDBMS design generally support two main features: *Isolation* and *Remote Attestation*. Isolation means that certain parts of the code and memory are hidden from the rest of the server hardware, allowing decryption, processing and optionally re-encryption of the encrypted data. *Remote Attestation* is a process that allows the remote validation of the integrity of a secure hardware component. Through remote attestation a client can verify that the trusted hardware running on the server side is in a consistent state and not compromised by an adversary. Throughout TH-EDBMS literature, a variety of hardware types have been used such as smart USB keys (63), FGPAs (6), and Secure Coprocessors (SCPUs) (40). Recently, research on trusted hardware has been dominated by a relatively new type of trusted hardware: trusted execution environments (TEE). In this section, the most relevant TH-EDBMS literature will be covered starting with the earlier works based on dedicated trusted hardware, followed by the more recent works using TEEs.

3.3.1 Dedicated hardware

The main concept of using trusted hardware goes back to the early 2000s (64)(65)(66). In this early work, the trusted hardware used is generally a secure coprocessor, a dedicated computer-on-a-chip that is implemented as a PCI extension card. This early research was

mainly theoretical and explored how secure coprocessors could solve some of the privacy issues from the DBaaS model.

3.3.1.1 TrustedDB

One of the first publications to present a prototype with full SQL support is TrustedDB by Bajaj et. al. (40). TrustedDB is a database prototype that allows query processing over encrypted data. TrustedDB is designed to work with a IBM 4764-001 PCIX secure coprocessor. The design goal of TrustedDB is to offer the SQL functionality of an unencrypted DBMS while protecting the confidentiality of the data from a honest-but-curious cloud provider. The TrustedDB architecture consists of two separate query engines: an unmodified, unsecure DBMS to handle the queries over public data, and a highly modified SQLite core running inside the secure processor. Queries are sent to the server encrypted and forwarded to a query parser running inside the secure processor. The query parser splits the queries into a private and a public part and forwards those parts to the corresponding query engine. Due to the limited performance of the IBM coprocessor, query processing speed on private data is significantly slower than public data. Therefore the design of TrustedDB allows a database designer to take advantage of the fast, unencrypted DBMS for non-sensitive data by only encrypting the sensitive column. The authors evaluate the performance overhead of TrustedDB through a series of benchmarks and remark that while TrustedDB comes at a significant overhead it is still orders of magnitude faster than cryptographic solutions such as FHE[1].

3.3.1.2 Cipherbase

In 2013, Arasu et. al. published a work presenting Cipherbase. Cipherbase was inspired primarily by CryptDB and TrustedDB. Arasu et. al. recognized that the design of TrustedDB where essentially a full DBMS is ran inside the trusted hardware leads to significant performance issues. For example, the AES decryption algorithm used inside the SCPU used by TrustedDB to decrypt the encrypted pages, only achieves decryption speeds in the order of tens of MB/s where the same operation on hardware assisted components such as CPUs can reach multiple GB/s. To work around this performance limitation, Arasu et. al. propose a completely different design. Values are encrypted per individual value and an FPGA is used to construct the trusted component that allows for computation. The authors describe this setup as *simulated fully homomorpic encryption*. The operations

[1]It should be noted that FHE is not considered a feasible technique for secure query processing, with performance overheads of many orders of magnitude

supported are listed in figure 3.6. Using these operations, a query plan is constructed just like in a regular DBMS. An example query plan is shown in figure 3.7. This architecture

Plaintext Operation	*Primitive in TM*
$\sigma_{A=5}$	$\mathsf{Dec}(\overline{A}) = \mathsf{Dec}(\overline{5})$
π_{A+B}	$\mathsf{Enc}(\mathsf{Dec}(\overline{A}) + \mathsf{Dec}(\overline{B}))$
$\bowtie^{hash}_{A=B}$	$\mathsf{Hash}(\mathsf{Dec}(\overline{A}))$; $\mathsf{Dec}(\overline{A}) = \mathsf{Dec}(\overline{B})$
AGG(SUM(B))	$\mathsf{Enc}(\mathsf{Dec}(\overline{B}) + \mathsf{Dec}(\overline{partialsum}))$
INDEX OPERATIONS	$\textsc{FindPos}(\mathsf{Dec}(\overline{k}), \langle\, \mathsf{Dec}(\overline{k}_1), \ldots, \mathsf{Dec}(\overline{k}_n)\, \rangle)$
RANGELOCK	$\mathsf{Dec}(\overline{v}) \in [\mathsf{Dec}(\overline{l}), \mathsf{Dec}(\overline{h})]$

Figure 3.6: Cipherbase DBMS operations on encrypted values supported through FPGA based TH architecture (6)

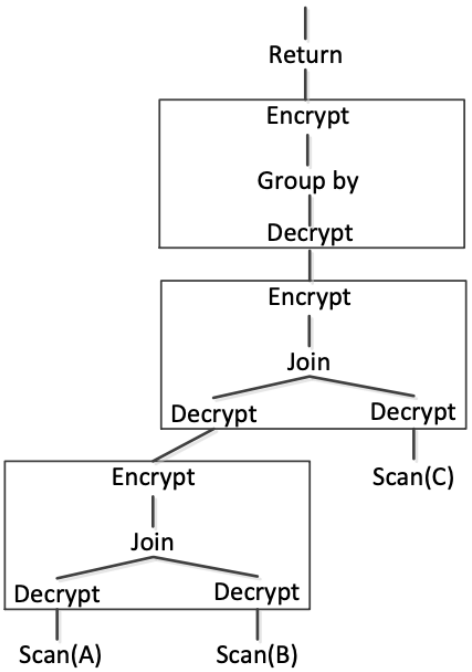


Figure 3.7: Cipherbase example query plan (6)

has several advantages over that of TrustedDB. Firstly it allows offloading a larger part of the query processing to the untrusted part of the architecture utilizing cheaper and/or faster commodity cloud servers as much as possible. Especially when only relatively few columns need to be protected this will be significantly faster than the TrustedDB design. Secondly the Cipherbase design requires very little modification to the existing DBMS as it only requires writing hooks for the operations on the encrypted data types. This allows

for easier software development and the usage of an industrial strength query engine with little modifications. Finally, by keeping the functionality of the TH simple, the hardware performance of FGPAs can be used to efficiently perform the secure operations.

3.3.2 Trusted Execution Environments

In 2015 Intel launched SGX, a feature supported by most of their CPUs, that allowed the creation of so-called *secure enclaves*. Intel SGX falls in a category of TH solutions called Trusted Execution Environments (TEE). While other TEEs did exist, most notably ARM TrustZone, the widespread usage of Intel CPUs for cloud infrastructure and the promise of low computational performance overheads sparked the interest of many EDBMS researchers leading to an extensive body of work on SGX based EDBMS design. In this section we will cover the most relevant works in TEE based EDBMS literature. While research using other TEE solutions does exist, all works in this section focus on Intel SGX as it is by far the most prevalent. As a sidenote, Microsoft has developed software-based TEE technology which they use for their Cloud DB service, which is covered in section 3.4.1.

3.3.2.1 EnclaveDB

In 2018 Priebe et. al. pubished a paper presenting their SGX-based EDBMS protype, EnclaveDB (7). EnclaveDB has Full SQL compatibility and provides confidentiality for both queries and data. Additionally it guarantees integrity and consistency of the database. Priebe et. al. remark that this combination of security properties can not be matched by PPE+PHE systems as those can not protect query privacy or database integrity and consistency. Also this is an improvement over dedicated hardware approaches such as TrustedDB and Cipherbase that do not guarantee integrity and consistency.

The architecture of EnclaveDB consists of a Microsoft SQL Server instance running on the untrusted server. Inside the SGX enclave runs a modified Hekaton engine, Microsoft SQL Server's in-memory query engine. This architecture is shown in figure 3.8. Data is split at table-granularity between sensitive and insensitive tables. Sensitive tables are stored in secure enclave memory, while insensitive tables are stored outside the enclave. Queries can be either on insecure tables or on secure tables, but not both. When a query on an unsecure tables is performed it is handled by the SQL Server instance. When a query on secure data is requested by the user, it is compiled client-side into a stored procedure. This procedure is then passed to the in-enclave Hekaton engine. While EnclaveDB theoretically

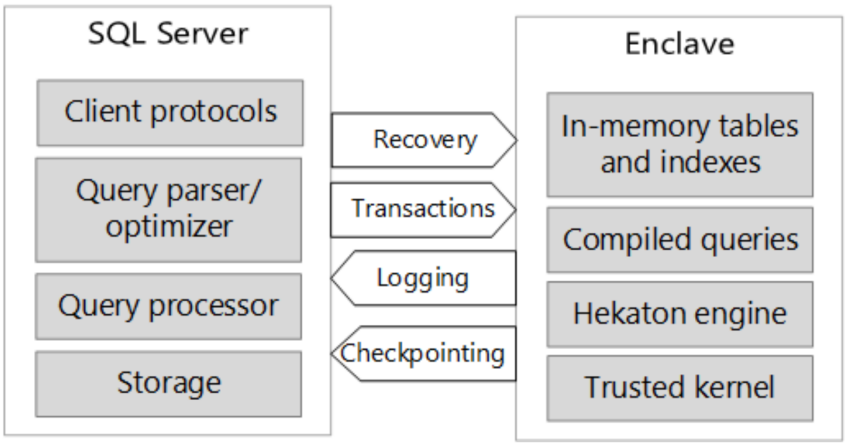


Figure 3.8: EnclaveDB server-side architecture (7)

offers very strong security guarantees combined with very low overheads (< 40% for TPC-C), currently it does not work in practice. The reason for this is that EnclaveDB is designed around the existence of a large amount of enclave memory that is available. Since enclaves of these size are not supported, the authors created a performance overhead simulation to benchmark their system. In their simulation, enclaves of 192GB are assumed. Whether or not enclaves this size will be supported in the future remains unknown. The most recent implementation of Intel SGX supports 256MB of total secure memory which results in 192MB being available to all enclaves in practice. Note that while current generation SGX enclave memory is limited in size, a paging mechanism exists to create larger enclaves but this imposes very large performance overheads.

3.3.2.2 StealthDB

In 2019 Gribov et. al. published a paper presenting StealthDB (8). StealthDB is similar to EnclaveDB in the sense of it being an SGX-based EDBMS focusing on OLTP workloads, but takes a completely different approach. Firstly, StealthDB does not assume the availability of large enclaves. Gribov et. al. state that it is very much an open issue whether larger enclaves can be supported efficiently and therefore include a limited secure memory size in the requirements for the design. Secondly StealthDB does not aim to provide query confidentiality nor does it aim to provide the lowest possible leakage profile. StealthDB instead opts for a "reasonable" leakage profile comparable tot the state of the art in PHE+PPE (stated to be Cipherbase by Gribov et. al.). Additionally, the StealthDB design is very simple to implement and has a very small Trusted Computing Base (TCB).

The TCB is the set of all software and hardware that is trusted in the trust model. The smaller the TCB, the more secure a system is.

The architecture of StealthDB is completely different from the EnclaveDB design and is more similar to that of Cipherbase. Similarly to Cipherbase, it opts to minimize the amount of code executed inside the trusted environment. For their design Gribov et. al. consider three different approaches for StealthDB which are shown in figure 3.9. In the

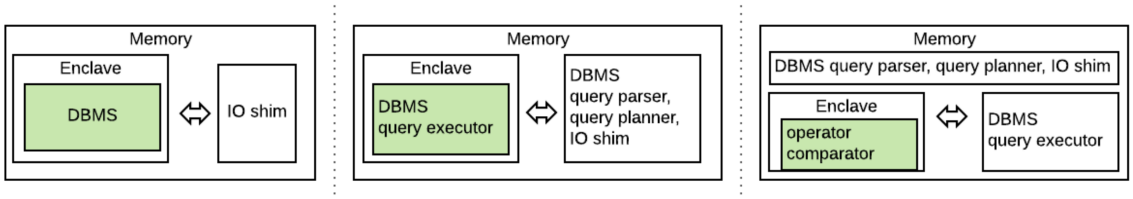


Figure 3.9: StealthDB architecture consideration (8)

first design the entire DBMS resides inside the enclave. This is the approach taken by EnclaveDB. The second design moves several components outside the enclave. Gribov et. al. argue that both the first and second design are not feasible as they would incur large overheads of 3× to 9× only for reading and serializing data inside the enclave. Their design follows the third architecture where only arithmetic operators and comparators are implemented inside the enclave. To achieve this, the operators are implemented as extensions on top of an unmodified Postgres instance. With the approach taken by StealthDB, usage of enclave memory is limited and therefore avoids the expensive paging mechanism. However this design does suffer from significant overhead from repeated entry and exit of the enclave mode, an operation that comes at a significant overhead of roughly 9k cycles per entry (67), this has a significant impact on performance. To mitigate this, StealthDB uses an optimization supported by the Intel SGX SDK called Switchless Enclaves (67). This Optimization reduces the overhead by roughly 1 order of magnitude.

3.3.2.3 CryptSQLite

In 2019 Wang et. al. presented CryptSQLite, an EDBMS based on the embedded database SQLite. CryptSQLite is an SGX-based EDBMS with an architecture following the first of the three designs for an SGX EDBMS described in figure 3.9. CryptSQLite places the entire SQLite engine inside the enclave and only runs a lightweight shim on the untrusted server for request handling, disk I/O and remote attestation. Data is stored as a single database file on the file system and encrypted at page granularity with the authenticated encryption

scheme AES GCM. A Merkle tree is built and maintained to ensure the integrity of the database pages under the threat of an active attacker. Queries are executed by the SQLite engine inside the enclave by loading and decrypting encrypted pages from the database file into the enclave memory. Due to CryptSQLite residing entirely inside the enclave, and data being encrypted at page granularity, the security guarantees offered are very strong and comparable to those of EnclaveDB. Similarly to EnclaveDB, CryptSQLite suffers from the same memory limitation as it needs to store data inside the enclave to be able to query it. However, Wang et. al. do not mention this limitation explicitly, therefore it is unclear how CryptSQLite handles the dataset used for their evaluation, which exceeds the enclave size by factor 5. Regardless of which approach is used for their prototype, it will inevitably be responsible for a high performance overheads as it will require repeated entry of the enclave and/or decrypting of data.

3.3.2.4 EncDBDB

In 2020, Fuhry et. al. published (9) a paper presenting EncDBDB. EncDBDB is a database prototype built on top of MonetDB, a column-store OLAP-oriented DBMS. Fuhry et. al. acknowledge the limited amount of secure memory in SGX which is ignored by other work such as CryptSQLite and EnclaveDB. Also they state that a leakage profile like the one from StealthDB or PHE+PPE based solutions are a limitation in the usefulness of those systems. In their work the authors focus on a practical system with configurable leakage properties, for a specific type of query: range queries on strings. Fuhry et. al. make the observation that modern OLAP DBMS such as MonetDB rely heavily on string compression techniques such as dictionary encoding for their efficiency. For certain types of datasets these compression techniques may even be necessary for the feasibility of doing analytics over the data. To be able to balance the trade-off between performance and leakage profile, EncDBDB implements a set of dictionary encoded storage layouts for string columns.

Fuhry et. al. have defined 9 different storage layouts which differ from each other in two axis: frequency and order. Each axis has three possible values which result in 9 permutations and thus 9 different layouts which are shown in figure 3.10. The three order options are: sorted, rotated and unsorted. These options are pretty much self-explanatory and mean that the dictionary values are stored respectively: sorted starting at the lowest value, sorted starting at a random offset, or in random order. The three frequency hiding options are: revealing, smoothing and hiding. The frequency revealing option applies maximum dictionary encoding: identical strings will be encoded to the same

		order options		
		sorted	rotated	unsorted
repetition options	frequency revealing	ED1	ED2	ED3
	frequency smoothing	ED4	ED5	ED6
	frequency hiding	ED7	ED8	ED9

Figure 3.10: EncDBDB dictionaries showing combinations of leakage profiles (9)

dictionary key and therefore equality between strings is leaked. Frequency smoothing uses a randomized function and a parameter bs_{max} to insert between 1 and bs_{max} times an encrypted copy of a string value in the dictionary. For each occurrence in the column the id points to a random copy of the value limiting the leakage. Finally the frequency hiding option will effectively apply no compression at all by adding a new value to the dictionary for each encoding. This option will prevent all equality leakage.

3.3.3 Access pattern hiding

So far the solutions presented have focused mainly on data confidentiality and integrity while leaving out a specific attack vector: access pattern leakage. Access pattern leakage is where the adversary learns which parts of the data are accessed. These access patterns have been shown to allow attacks revealing significant amounts of information in database systems (68)(69)(70), especially for attackers with other means of access to the unencrypted data. In the PHE+PPE based systems that have been covered in chapter, preventing leakage of access pattern is very difficult, as the server needs to be able to operate on the encrypted values and will therefore learn which tuples were accessed. In TH-based systems, preventing access pattern leakage is possible, though not all systems try to hide the access patterns: StealthDB, EnclaveDB, and TrustedDB consider access pattern leakage to be out of scope. CryptSQLite also suffers from high-level access pattern leakage, but its authors argue "that such information is very limited to the attacker". Cipherbase, the TH-based EDBMS described in 3.3.1.2, implemented several oblivious operators based on the work by Goodrich et al. from 2011 (71). More recently, research was published specifically focusing on oblivious TH-based EDBMS that we will briefly cover in this section.

3.3.3.1 Opaque

In 2017, Zheng et. al. published an Apache Spark-based oblivious query processing prototype called Opaque. The goal of Opaque is to efficiently implement oblivious distributed data analytics supporting a wide range of queries. Opaque uses Intel SGX for its implementation, but is designed to work with possible future TEE technology in mind. This is important as the current implementation of Intel SGX leaks memory access patterns when accessing the secure memory (72). However, research (73)(74) has been published with enclave designs that do protect against these types of access pattern leakage at little additional overhead. Opaque offers three modes of operation each with different security profiles: encryption-only, oblivious, and padded oblivious. In their evaluation, encryption-only mode was found to have an overhead of up to 2.4x, while oblivious mode resulted in an overhead of up to 46x. Padded mode was not evaluated experimentally, but would incur even higher overheads, especially for queries with filter operations with low selectivity.

Encrypted mode Firstly, encryption-only mode provides no protection against access pattern leakage, but only ensures data confidentiality, data integrity and computation integrity. No attempt is made to hide any access patterns.

Oblivious mode When set to oblivious mode, Opaque uses a set of oblivious operators to allow oblivious query execution. The oblivious operators in Opaque are based on oblivious sorting. The oblivious sorting algorithm consists of intra-machine and inter-machine sorting and is based on a sorting technique called sorting networks. Additionally, an oblivious filter operator is implemented. The oblivious filter operator is very simple: instead of filtering out tuples, tuples are marked with a 1 for passing the filter or 0 for being rejected by the filter operator. Using the oblivious filter and oblivious sorting operators, an oblivious aggregate and oblivious merge join operator are constructed.

Padded mode While the oblivious operators hide the access patterns, the result set size of each operators is still leaked to the adversary. To mitigate this, Opaque supports an oblivious padded mode, which ensures that none of the relational operators reduce the output size except for the final operator. This is achieved using a filter push up that moves all filter operations to the end of the query.

3.3.3.2 ObliDB

In 2019 Eskandarian et. al. published ObliDB, an oblivious EDBMS prototype that aimed to offer significantly improved performance to Opaque. ObliDB is also based on Intel SGX but built with future TEE improvements in mind, similarly to Opaque. It also assumes the oblivious memory available to the TEE is a limited resource whose size can be configured. Also the security guarantees are similar to Opaque and a similar padded mode exists to hide intermediate results sizes. ObliDB improves on the Opaque design in several ways. Firstly, it adds support for insertions and updates. Secondly, it allows for efficient queries on small subsets of data in a table such as point queries or range queries with low selectivity. Finally, it improves performance with several optimizations and new oblivious operators.

In ObliDB, data can be stored in two modes: flat or indexed. In flat mode, data is stored sequentially in encrypted blocks. In indexed mode, an ORAM is used with a B+ tree stored inside. For the flat layout, obliviousness is achieved by scanning the whole table, the indexed layout relies on the ORAM to provide oblivious access. To achieve best performance for a wide range of queries, both storage layouts can be combined. ObliDB uses a variety of oblivious operators to cover different query types: four different select operators, three join operators, and two aggregation/group-by operators. Some of these operators are new, others are based on the operators presented by Opaque. ObliDB uses a query planner to choose which operators to use based on which storage layouts are available, the amount of oblivious memory available to the TEE, and statistical information on the input and output table sizes. Along with its new set of operators and query planner, ObliDB also introduces a set of new optimizations that further help improve performance. In their evaluation, Eskandarian et. al. conclude that ObliDB performs similarly to Opaque when using flat storage, but can improve performance by up to 19x when using the indexed storage method. Their performance approaches unencrypted Spark SQL performance with only 2.6x relative overhead.

3.4 Industry adoption

As demonstrated by the literature covered in this chapter so far, research into practical EDBMS has been extensive. However, industry adoption of these techniques has been very limited. For example, Google has experimented with adding PHE to its BigQuery client (75) but the project has since been abandoned and remains an experimental plugin. Several companies are mentioned repeatedly in EDBMS literature as using PHE+PPE based techniques such as Ciphercloud, Vaultlive, and IQCrypt (Mentioned in (68) and

(76)). However, many of the companies mentioned no longer exists or when they exists, or seem to no longer be presenting PHE+PPE based EDBMS functionality. To the best of our knowledge, the only well-known system in use is the Always Encrypted (AE) functionality offered by Microsoft for its SQL server cloud service.

3.4.1 Microsoft Always Encrypted

In 2020 Microsoft published a paper (77) describing both its initial version Always Encrypted v1 (AEv1) launched in 2016 and its updated second version AEv2, which was launched in 2019. In their paper the authors describe how, in spite of the very limited functionality offered, AEv1 has found usage by a variety organisations such as large insurance companies, healthcare providers and financial institutions. With the recent developments in regulatory compliance requirement due to for example GDPR, Microsoft expects this demand to increase.

As mentioned, AEv1 supported only a very limited feature set: columns could be encrypted deterministically with AES in CBC mode, to support equality operations over these columns. As described in their paper (77), AEv1 suffered from a severe limitation. When switching an existing column to AE mode, the data would need to be sent to a trusted machine for encryption to get encrypted. The resulting latency could be as long as a week for some customers with large databases. Furthermore, this would result in a lot of network traffic resulting in prohibitive cloud provider costs. Additionally, this would also occur each time an encryption key is rotated.

In AEv2, Microsoft integrated TEE techniques into AE to solve its main problems. Firstly, using a TEE AEv2 allows for the support of equality, range comparisons, and LIKE pattern-matching predicates on columns encrypted with randomized encryption. Secondly, the TEE allows encryption of data without the requirement of moving the data across a network to a trusted clients, allowing for easier, cheaper migration and key rotation. Interestingly Microsoft did not choose to use Intel SGX as their TEE techology. Instead it opted to uses Virtualization-based Security (VBS) (78). VBS has an inferior trust model compared to Intel SGX as it require the entire CPU and hypervisor to be trusted. Microsoft mentions that an AE version supporting Intel SGX is still being worked on.

3.5 Open issues in current literature

To finalize and conclude this related work section we summarize the studied literature by listing the main open issues found through the analysis of the research.

Information leakage from PPE Schemes PPE Schemes inherently leak information to an adversary. This information leakage has been shown to allow inference of significant parts of the database (68). Especially ORE has been shown to be especially dangerous as the level of security ORE schemes achieve is poorly understood. Recent work on ORE (79)(80) demonstrates that finding an efficient, low-leakage ORE schemes is still very much an open issue.

Efficient AHE and MHE schemes Classical PHE schemes for addition an multiplication such as ElGamal and Paillier come with serious performance overheads. Optimizations such as those by Ge et. al. (49) provide some performance gain, but overheads remain high. Recent work (20)(5) has looked into faster AHE and MHE schemes, but these schemes sacrifice ciphertext compactness for performance, possibly resulting in very large ciphertexts for certain queries. Additionally, these techniques require heavily on prior knowledge on the expected workload.

TEE availability TEEs has been the main focus of recent TH based EDBMS research. However, availability of TEEs at cloud providers has been limited. Even ARM Trustzone and Intel SGX, the most well-known technologies, have limited availability at cloud providers. For example, AWS has no support for instances with TrustZone or SGX, Azure only supports instances with Intel SGX.

Intel SGX limited secure memory Intel SGX has only limited secure memory available: 192MB for the most recent CPUs. Exceeding this limit severely impacts performance due a slow paging mechanism. This limitation poses a challenge for designing fast EDBMS, as these will generally exceed this limit by many orders of magnitude for normal workloads. Designing an EDBMS that works around this limitation or designing TEE technology supporting larger enclaves remains one of the main open problems.

Intel SGX security Finally, a recurring theme in all SGX based EDBMS literature, is the persistent existence of a variety of attacks. Intel SGX has a history of many breaking attacks and during this Master's project alone, multiple new attacks have been published (81)(82). Additionally, Intel SGX does not protect against memory side-channel attacks (72), unlike other, more experimental TEE architectures such as Sanctum (74) and RISC-V Keystone (83).

4

A DuckDB EDBMS

In this chapter, we outline the design of the DuckDB-based EDBMS which we will refer to as EDuckDB from now on. The design consists of two main parts: the choice of EDBMS primitives used and the positioning of the requirements of the EDBMS with regards to the performance-security-functionality trade-off. Firstly, the two main types of EDBMS are described in section 4.1. Secondly, three different use-cases for EDuckDB are covered along with their corresponding threat models. For each threat we analyze the security offered by each of the two designs. With the analysis of the security properties of both designs and the analysis of the learnings from the existing literature from chapter 3, we then explain our choice for a TH based solution. Finally, the requirements for our EDBMS design are outlined in 4.5.

4.1 EDBMS models

As described in chapter 3, there are two main approaches to construct EDBMS in current literature. Both approaches have their own distinct characteristics with regards to performance, functionality and security. To analyze which is most suitable for our goal we outline the two approaches by describing their global architecture and their corresponding trust model.

CRYPTO-EDBMS Firstly we will consider the cryptography based EDBMS as described in section 3.2. We will refer to this model as CRYPTO-EDBMS. Figure 4.1 describes the architecture of the CRYPTO-EDBMS design. The design consists of two main components, the EDBMS Proxy and the EDBMS server. The EDBMS proxy handles the

encryption and decryption of the data in the queries. Its main tasks include transparently encrypting secure data values in queries, transparently decrypting data in returned queries, and client-side query processing that could not be performed server-side. Another task of the proxy is to optimize the choice of encryption schemes for each column and optionally generating pre-computed columns. The other main component is the EDBMS server. This is generally built on top of an existing non-encrypted DBMS and provides the unencrypted DBMS functionality extended with special homomorphic operators and/or secure index handling. For our research, we have only considered the two party designs and omit three party designs such as Blind Seer described in sections 3.2.2.1. The main reasons for omitting >2 party designs are that two party designs fit the use-cases better and two-party designs can be much easier integrated into existing DBMS technology.

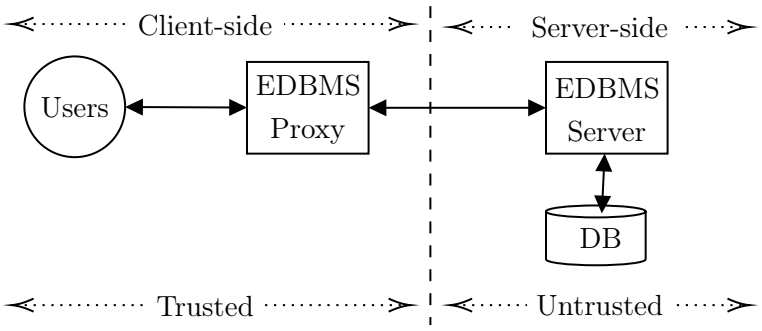


Figure 4.1: CRYPTO-EDBMS model

TH-EDBMS TH-EDBMS is the second model we consider. It represents an EDBMS based on the techniques defined in section 3.3. The global architecture and trust model are depicted in figure 4.2. The TH-EDBMS model consists of an EDBMS server and an EDBMS TH component. Differently from the CRYPTO-EDBMS model, no client-side component is required generally. The trust model is also different from the CRYPTO-EDBMS as a trusted server-side component is required. In the TH-EDBMS model there are two components that can vary significantly in their functionality. The EDBMS server component handles the requests, but most other functionality can be performed by either the EDBMS server or EDBMS TH components. For more details on this, see section 6.2.

4.2 Encrypted DuckDB Use-cases

To be able to design EDuckDB, we will need to understand how the EDBMS would be used in the real world. With an understanding of the use-cases, we can determine the require-

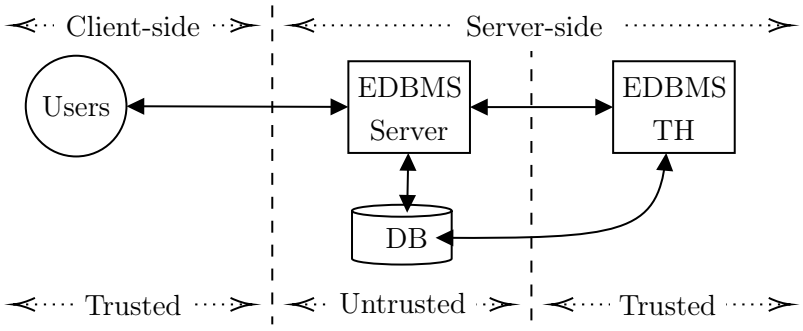


Figure 4.2: TH-EDBMS model

ments such a system would have and what threats can be expected and need protecting against. In this section we define three different use-cases and list their main threats. For each threat we identify the the current state-of-the-art security techniques and analyze how its security would benefit from applying either of the two EDBMS models.

4.2.1 Use Case 1: Local machine

The first use-case we define is that of a database running locally on the same machine that is used by the user to enter queries. For example, an analyst running EDuckDB on a laptop to perform OLAP or ETL workloads. In this use-case the query processing, data storage, and user input and output are all happening on the same physical machine. For this use-case we identify two main threats.

Threat 1: Physical access to device The first threat is an attacker with physical access to the device running the database. An example of a threat from this type is a stolen laptop with the database stored on it. For this threat model we assume the device is trusted initially, but becomes completely untrusted after loss or theft of a device. A commonly used approach to counter this threat is to use full disk encryption (FDE) solutions such as FileVault (84). These solutions encrypt the entire disk and often require a password on boot or on waking up from sleeping or hibernating. FDE solutions can be purely in software or use special hardware components. When FDE is implemented only in software it can come with high overheads since the CPU will need to decrypt and re-encrypt data read/written from/to the disk. Usage of dedicated hardware can reduce the overhead to negligible amounts. Security-wise, FDE can effectively protect against physical attacks in most common situations such as a stolen/lost laptop. There are some limitations however: with FDE, the decryption key needs to be stored on the device to be

able to decrypt/encrypt pages as they are read/written. The key is vulnerable to being stolen as long as it resides in memory through attacks such as cold booting (85) or DMA attacks (86). In similar style, FDE does not protect data stored in memory, which means that any data stored in memory during loss/theft, such as an in-memory database, are vulnerable to the aforementioned attacks. When we consider how the EDBMS models could improve the security of the system under this threat, we find that both CRYPTO-EDBMS suffers from exactly the same limitations and vulnerabilities as FDE. TH-EDBMS on the other hand, could improve security by preventing the cold booting and DMA attacks if we assume the TH technology used is secure against those types of attacks.

Threat 2: Compromised machine A second threat for this use-case is when the local machine is infected by a remote attacker. In this case FDE would be completely useless as the decryption key is available to the attacker as soon as the user logs in. Using an EDBMS could offer some security guarantees but not against every type of attacker. If we consider an active attacker, we cannot offer any security guarantees, because an active attacker could do anything the user can do and for example manipulate the DuckDB application to dump the whole database. When considering passive attacks for this use-case, an attacker could steal the key and the data from memory or storage. With regular encryption or PPE we can not prevent this attack since the key would reside in memory and thus be available to the attacker. However, we could use a trusted hardware approach such as Intel SGX to protect the key and decrypted in-use data. Note that all queries and their results would be available to an attacker.

4.2.2 Use-case 2: Edge Computing

The second use-case we define is in securing Edge Computing, for example in Internet-of-Things (IoT) applications. Consider a network of security cameras that scan number plates of cars driving by and store that information locally on a storage inside each camera. By using Edge Computing, queries on number plate data can be done quickly by sending a query to each security camera without a need for the camera to constantly send its data to a central database. For this use-case we identify two threats.

Threat 1: Physical attack The first threat we consider is physical attack, also called the node capture attack (87) in this context. In this attack the attacker has full physical access to the a device. In our example, this would mean the attacker has access to one or more security cameras, reads their disks and/or memory, and learns which number plates

drove by over the period for which the camera stores its data. Just like in our first use-case, FDE could be deployed to make it harder for an adversary to read the data if it is stored in a database on the disk. However, similar to the first threat in our first use-case, the key will need to be stored in memory in the device and will therefore be susceptible to attacks like cold-booting. We will now consider how the two EDBMS models could offer increased security over a non-encrypted database.

First consider CRYPTO-EDBMS. In this model we can offer some additional security over an unencrypted database, but there is a caveat. As discussed in section 3.2, PHE+PPE based EDBMSs rely on a variety of encryption schemes, some symmetrical, some asymmetrical. For our EC use-case however, no symmetrical schemes will be able to provide any security as the key needs to be present on the device to be able to insert data into the database and this key would be available to an attacker in the node capture attack. Now while using only asymmetrical cryptography for a CRYPTO-EDBMS design is possible, it would come at significant performance overhead as is indicated by the tendency in PHE+PPE literature to move away from asymmetrical cryptography (20)(5). The practicality of this performance overhead is questionable.

Now consider TH-EDBMS for the EC use-case. For this design we note that increased security can be provided if and only if a TH architecture is chosen that is secure against attackers with physical access. While TEEs like Intel SGX are often considered in the unsecure cloud provider trust model, these models generally assume the HbC attack model, which we have explained in section 2.3, and exclude active physical attacks. Recently, it has been shown that Intel SGX is vulnerable to a physical access attack by Chen et. al. (88) which Intel has explicitly stated will not be fixed. As stated by Chen et. al., this casts clear doubt on the fact whether any TEE solution can ever be secure under physical attacks.

Threat 2: Software compromise The second threat considered is when the software running on an Edge node is compromised by an adversary. In this attack either the application(s) or OS running on the edge nodes may be compromised by an attacker. The main method to protect a system consists of either reducing the total amount of code that can contain vulnerabilities, also known as the trusted computing base (TCB), or by making it harder for an attacker to effectively exploit any vulnerabilities. This concept is also know as system hardening. For example, one could use a security-focused Linux distribution such as Alpine Linux on their edge nodes. We now consider our if our two EDBMS models could improve the security of a hardened edge node. Firstly, we consider CRYPTO-EDBMS. This model suffers from the same issues as under a physical attack:

there is no way to securely store a symmetrical key on the edge node meaning that only asymmetrical schemes can be used. Again, the practicality of the performance of such systems is questionable. Secondly, we consider the TH-EDBMS design. We note that this design fares very well under this threat model. TH is generally designed with active attackers included in the threat model. For example, Intel SGX is specifically designed against the threat of an active attacker software. A system using TH-EDBMS as a storage on their edge nodes would be able to improve system security by reducing its attack surface to the TH component and the code running inside the TH.

4.2.3 Use-case 3: Cloud service

The third use-case is for DuckDB to be used as (part of) as cloud service. The classic example is an outsourced database service such as AWS RDS. In this use-case the trust and threat models similar to the outsourced database model used by most EDBMS literature with and untrusted server and a trusted client. For this use-case we identify to main types of threats, which are based on the types generally distinguished in the literature: the active and the passive attacker.

Threat 1: Active attacker Firstly, we will analyze the threat of an active attacker. The active attacker has full control over the system and can control and manipulate all server-side hardware and/or software in its attack. An example attack in this threat model is a malicious cloud provider employee actively trying to attain confidential information from the customer database for his own gain. To protect against this threat while maintaining functionality, usage of an EDBMS is required. CRYPTO-EDBMS will most likely not be able to provide any meaningful amount of security under this threat model as attacks as demonstrated by Grubbs et. al. (69). In their attack, Grubbs et. al. demonstrate how an EDBMS-like system for web applications, Mylar (89), is vulnerable to a dictionary attack on its searchable encryption scheme. In their analysis, the authors generalize this attack to demonstrate that all PHE+PPE systems have fundamental security issues under the threat of an active attacker. More recent PHE+PPE based systems (5)(58) as covered in section 3.2 do not claim security under active threats but only under passive threats. TH-EDBMS can protect against active attackers. As described in section 3.3, several approaches have claimed security under the presence of active attacker (72)(7)(21). Note that not all TH-based approaches automatically provide this protection, such as is the case for lower security systems such as StealthDB (8). Also note that similarly to use-case 2,

attacks to the physical hardware and side-channel attacks are generally not included in the attack model of TH-based solutions.

Threat 2: Passive attacker The passive attack model is also known in this context as the Honest-but-Curious model and represents a cloud provider who does not interfere with the normal operation of the service in any way, but can read all memory, storage, network traffic and CPU state. This attack model is the attack model that is generally used in PHE+PPE literature. CRYPTO-EDBMS would therefore be able to provide security in against a passive threat for this use-case. As for TH-EDBMS, it supports the stronger active attack model and therefore also holds under this weaker attack model.

4.3 Choosing the best fitting model

Based on the use-case analysis of section 4.2, we can conclude that the TH-EDBMS model is more flexible than CRYPTO-EDBMS as it can offer security guarantees against more of the threats we defined. We now directly compare the two models on their security, performance and functionality to come to the final conclusion that TH-EDBMS is the most suitable approach for our our use-cases.

Security Security-wise, both of the models have weaknesses. The TH-EDBMS model relies on trusting a hardware component being available on the server-side such as a secure coprocessor or TEE. First of all, implementing this hardware securely has proven to be hard, as described earlier in section 3.3. Furthermore, when the attacker has physical access to the TH, some types of attacks might even be fundamentally impossible to mitigate (88). CRYPTO-EDBMS is theoretically the more secure model as it does not require trusting any server-side component. However, CRYPTO-EDBMS suffers from other issues. Most importantly, the PPE schemes on which most implementations rely, have serious weaknesses that can be easily abused by both active and passive attackers. While alternative schemes that leak less data do exist, such as secure indexing approaches like Arx, these are less suitable for OLAP workloads. The reason for this is that Arx uses secure indexes to be able to perform comparisons over secure data. These indexes are efficient for point queries when small amounts of comparisons need to be made. However for OLAP workloads, where large parts of columns are scanned and filtered, these indexes are not efficient.

Performance Performance-wise there are large differences between the two, especially for our OLAP-oriented domain. As can be seen from the TPC-H results of Savvides et. al. (5), even in a state-of-the-art PPE+PHE based design, some queries have very large overheads of multiple orders of magnitude exist. As mentioned before, secure indexing approaches like Arx do not perform well for OLAP workloads. Since OLAP workloads depend heavily on the filter operation and CRYPTO-EDBMS systems either need PPE or secure indexes to be able to provide filtering on encrypted data, the OLAP performance of a CRYPTO-EDMS is unlikely to have low overheads. TH-EDBMS has significantly better performance characteristics. System such as StealthDB (8) demonstrate that reasonable overheads can be achieved when allowing for data leakage comparable to CRYPTO-EDBMS systems. Another important performance consideration is storage overhead. Especially in the context of OLAP workloads, where queries access data volumes in the order of the database size, minimizing storage overhead is important for achieving good performance. For CRYPTO-EDBMS storage overhead can be very high due to the need for encrypting columns multiple times with different schemes to provide the required functionality. Depending on the workload this overhead can increase the database size by multiple times. TH-EDBMS perform generally better in this regard. If sufficiently large encryption granularity is chosen, storage overhead can get negligible with only a few percent or less.

Functionality Finally we compare the supported functionality. Here TH-EDBMS is clearly superior as it imposes no fundamental functional limitations. Systems such as EnclaveDB or CryptSQLite manage to run entire DBMS systems inside TH allowing for any functionality also supported by the used DBMS. CRYPTO-EDBMS systems generally have significant limitations functionality-wise. Because FHE remains impractically slow with current technology, CRYPTO-EDBMS need to rely on cryptographic schemes that do not support arbitrary computation. While many simple operations such as addition, multiplication, or comparison are supported, more complex operations such as user-defined functions (UDF) over encrypted data are impossible. Also when queries require chaining operators such as the query `SELECT SUM(a*b)from table;`, this proves to be problematic as no PHE scheme supports both addition and multiplication. As described in section 3.2 there are mitigations to this problem such as client-side execution and precomputation, but these often come at a significant performance and/or storage overhead.

4.4 Information leakage

In a perfect EDBMS there would not be any data leakage at all. However, as we have discussed in chapter 3, in practice achieving zero data leakage is not possible or practically infeasible. In this section we will discuss the different types of data leakage that exist in EDBMS, what different classes of accepted leakage are found in literature, and finally what leakage pattern we will focus on in this research.

4.4.1 Direct vs Indirect

Information leakage can happen in two main ways: directly or indirectly. Direct data leakage happens when data is directly visible to an adversary. This can happen in various ways: for example, parts of the data or meta-data may be unencrypted or encrypted with PPE schemes that inherently leak data. Direct data leakage is the easiest to protect against as it generally happens in an easily predictable manner. Direct data leakage is the direct consequence of a shortcoming/property of the whole system. The second type of data leakage is indirect data leakage, also known as side-channel leakage. Indirect leakage happens when an adversary can deduct information on the data from information leaking from the system implementation. An example of indirect leakage is when an adversary can see the query response times: this allows the adversary to deduct information on the query result set size. Indirect information leakage is much harder to protect against as the existence of side channels can be hard to notice. Figure 4.3 shows a diagram with a non-exhaustive list of leakage that may occur in a TH-EDBMS.

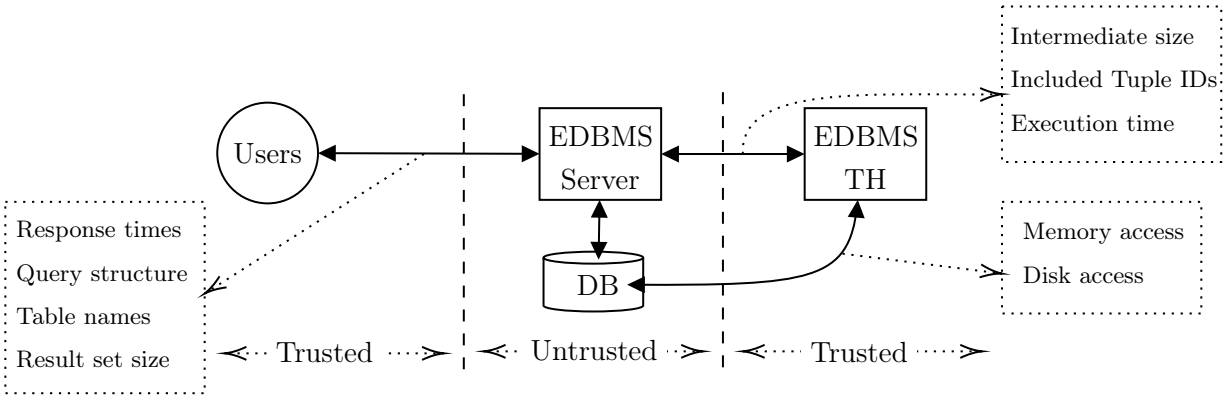


Figure 4.3: Examples of information leakage in the TH-EDBMS model

4.4.2 Leakage vs performance

Furthermore, reducing side-channels completely is often practically impossible: consider an EDBMS which sends queries over the public internet and that wants to completely eliminate side-channels. To achieve this, the EDBMS would need to pad all queries to the maximum query length and all result sets to their maximum size. It would also need to delay all query responses to a constant response time equal to the worst case response time. The resulting system is infeasible for most OLAP workloads. This leads to the conclusion that all practical EDBMS systems need to permit some level of information leakage. When designing an EDBMS, this leakage should be clearly defined.

4.4.3 Leakage patterns found in literature

In the EDBMS literature leakage patterns vary widely. We categorize them into 4 main classes. We will outline each category and give examples of EDBMS for each class. It should be noted that these categories are intended for roughly outlining the leakage characteristics of an EDBMS, exact leakage patterns are complex and can differ in subtle ways that can be crucial for specific security requirements.

Class 0 An EDBMS with leakage class 0 leaks a significant amount of information directly and indirectly. Furthermore the precise amount of information leaked is either difficult to understand, or not well understood at all, leading to unexpected attacks allowing retrieval of large amounts of the data. This class contains systems such as CryptDB, Monomi, Cuttlefish and Symmetria. These systems rely on PPE and/or Searchable encryption schemes which leak data in an unpredictable and/or poorly understood manner. An example is the OPE scheme used by CryptDB. This scheme by Boldyreva et. al. (90) leaks additional information which makes it vulnerable to attacks such as the attack by Grubbs et. al. (68). We think EDBMS with information leakage of this class are not suitable for real-world use as their exact security limitations are hard to understand.

Class 1 EDBMS of leakage pattern class 1 also leak information both directly and indirectly, but leak direct information in a well defined, predictable manner. Example EDBMS literature with this leakage class are Arx and StealthDB. Both these systems leak order and equality information for data that is passed through operators that require knowing those properties. Other properties that are generally leaked in systems in this class are the full queries (minus the encrypted constants), which tuples are included in a query results,

and various types of side-channel information. While systems with leakage class 1 are more usable than those of class 0, due to their direct information leakage, inference attacks as described in (91) can still relatively easily reveal large amounts of encrypted data.

Class 2 Class 2 EDBMS leakage patterns are those that do not directly leak information. For EDBMS to have class 2 leakage patterns, results of individual operations need to be hidden from the adversary. In these systems, the only leakage that occurs is through side-channels. Side-channels are not significantly addressed, however, which means that adversaries can still deduct significant amounts of secure data through side-channels such as access patterns, intermediate result size, or response times. Examples of systems that fall into this category are CryptSQLITE and EnclaveDB. In these systems entire queries over secure data are encrypted and processed inside the TH which hides information such as the query structure and operator results.

Class 3 Class 3 EDBMS leakage is when significant measures have been taken to mitigate leakage through side-channels such as timing and access-patterns. Systems in this category are ObliDB and Opaque. These systems use Oblivious operators to hide which parts of the data are touched by each query. Additionally, padding is used to hide intermediate result set sizes. EDBMS with class 3 leakage attempt to eliminate direct and indirect leakage as close as practically feasible to a perfect, non-leaking EDBMS.

4.5 Encrypted DuckDB

4.5.1 Requirements

In this section we will outline the requirements for our EDBMS prototype EDuckDB based on the finding in the previous sections and the research goals in section 1.2. An overview of the requirements is given in table 4.5.1. For each requirement we will briefly explain

Property	Requirement
Performance	Minimal overhead
Functionality	No significant limitations
Threat models	Honest-but-curious or Malicious
Trust model	TH-EDBMS
Leakage	Class 1 or Class 2

Table 4.1: Requirements for encrypted DuckDB design

it and give the motivation. Firstly, the performance of EDuckDB. As described in our research goals in in section 1.2, the goal of this research is to create an OLAP EDBMS with minimal performance overhead. Also, due to a lack of literature on optimized encrypted OLAP DBMS, no precise number can be given. To validate this requirement, a baseline implementation should be created to which we can compare the EDBMS design. Secondly, the functionality supported by DuckDB should not be fundamentally and significantly impaired by the design of EDuckDB. Other research on TH based EDBMS have demonstrated that full SQL functionality is possible for these systems and especially for OLAP workloads having full SQL support is crucial. Thirdly, the adversary models that will be used are the Honest-but-curious and malicious models. Ideally, EDuckDB would be secure under the malicious threat model, but the weaker honest-but-curious model is generally considered as a valid model for the use-cases we are targeting with EDuckDB. In the evaluation, both models will be discussed. Fourthly, the trust model will be identical to the general trust model followed by all TH based EDBMS, which is the trust model depicted in figure 4.2. Finally, cvwe now consider which leakage classes are suitable for the EDuckDB prototype. Class 0 leakage patterns will not be considered. An EDBMS with leakage patterns that are not clearly defined or based on encryption schemes that make it hard to understand what information is leaked is not going to provide enough security. Class 1 leakage is a viable option for EDuckDB. For some applications leakage may be acceptable when the data is not of high sensitivity. Especially if an EDBMS is able to keep performance overhead low, class 1 could be useful for uses where performance is more critical than security. Class 2 leakage is another viable option for EDuckDB. Running large parts of the query processing in the trusted hardware seems feasible as many TH based systems exist that seems to translate well to DuckDB. Class 3 leakage will not be considered for implementation to keep the scope of the research manageable. Implementing oblivious execution will require significant changes to operators and does not match as well with our research goal of designing an EDBMS with minimal overhead as class 2 and class 1 systems. However in the future work section 7.2 we discuss how the EDuckDB prototype could be expanded with oblivious execution.

4.5.2 Research setup

Now with the scope and requirements of EDuckDB clearly defined, the remainder of this research is divided into two parts: Baseline EDuckDB and EDuckDB with Intel SGX. In chapter 5 we will cover Baseline-EDuckDB which will cover the aspects that are independent of the chosen TH architecture. Then, in chapter 6 we will present the SGX-based

prototype. There are two main reasons for this structure. Firstly, the implementation of a TH independent implementation will provide a baseline for the SGX based implementation. Since there is no previous work allowing for direct comparison, establishing this baseline will give an idea how close the SGX based implementation is to ideal performance. Secondly, the TH-independent implementation will provide results that are useful in a wide range as they are not specific to one TH architecture but should apply to any vectorized OLAP EDBMS following the TH-EDBMS model.

5

Baseline Encrypted DuckDB

In this chapter, we will describe and evaluate our baseline encrypted DuckDB implementation. This is done for two reasons. Firstly, to get a baseline to compare our TEE based implementation with. This is useful, because in the literature there are no candidate systems for direct comparison. By themselves, these results give a good understanding of what performance can be expected when a OLAP-optimized DBMS processes analytical workloads over encrypted data. Secondly, with these experiments we compare various methods to encrypt the data and select which encryption scheme to use. This can then be used in Chapter 6 to build the TEE based implementation.

5.1 Encryption scheme

The first major consideration for EDuckDB is the choice of the encryption scheme. In this context, the encryption scheme refers to not only which algorithm is used to perform the actual encryption process, but also how data is grouped together into plaintexts and what parameters are used for the encryption algorithms.

5.1.1 Granularity

First, we look at the encryption granularity. Encryption granularity means at which level data is combined into buffers which will be the plaintexts that get encrypted into ciphertexts. As described in section 2.2, an encryption algorithm takes a buffer of a certain size and encodes it into a ciphertext. For a database, this means that a choice needs to be made on how the data in the database is divided into these buffers. This choice will greatly impact the performance, functionality and security properties of the resulting EDBMS. We will discuss the main granularities found in the literature and evaluate their suitability for

our design. In traditional data-at-rest database encryption such as Transparent Data Encryption (TDE) offered by most large DB providers such as Oracle IBM and Microsoft, the main granularity that is chosen is a physical page, meaning that pages are encrypted/decrypted as single buffers when they are read/written to disk. Since pages are typically 4KB, this means that data is encrypted in chunks of 4KB. In EDBMS literature, however, a variety of encryption granularities are found.

Per-value The first possibility for encryption granularity is per-value encryption. This means that each value is encrypted separately and scanning a column of data requires that the decrypt function is called for each value in the column. One of the main advantages of this approach is ease of implementation: many DBMS already offer functionality to encrypt individual values. A simple way to implement this is to define special encrypted data types and define their corresponding cast operations as the encrypt and decrypt functions. Since many DBMSs offer functionality to define custom types and cast operators, this can be easily implemented on top of a standard DBMS. Another advantage is that no significant changes need to be made to the execution model: operations over encrypted data can be implemented just like any operator for a custom data type would be implemented. This granularity is common in CRYPTO-EDBMS such as CryptDB and Monomi, as these systems rely on PHE and PPE schemes that will generally only work when values are encrypted individually[1]. For TH-EDBMS systems, encrypting individual values is also not uncommon: Microsoft AEv2, StealthDB, and Cipherbase all encrypt individual values. For all these systems the choice for per-value encryption is similar to that in CRYPTO-EDBMS: it allows use of traditional query engines with minimal modifications and easy mixing of encrypted an unencrypted data in a database. Finally, another reason to use per-value encryption is to be able to support encrypting columns with different keys in a row-store layout EDBMS: since data is stored with all values of a row contiguous in memory, encrypting columns with different schemes is only possibly when values are encrypted individually.

Per-page Another granularity found in EDBMS literature is page-level. Page-level encryption moves the encryption/decryption process to a different place in the architecture. Here, a paging mechanism such as is supported by many operating systems, is expanded

[1]There are exceptions to this such as a Paillier optimization used in Monomi, which packs values together and allows operations on the grouped encrypted values, see section 3.2.1.2

with encryption/decryption functionality: pages that are paged into the applications memory are decrypted and pages that are paged out are encrypted. This is commonly used to abstract away the memory limit of TH architectures to code running inside the TH. Example systems using this granularity are CryptSQLite, EnclaveDB and TrustedDB. The first two are built around the paging mechanism that Intel SGX provides, the latter implements its own paging module for an IBM SCPU. In per-page encryption, most of the secure query processing is done from within the TH and pages are copied and decrypted from unsecure memory on a page miss from within the TH. This approach can achieve significantly higher throughput than per-value encryption on queries that require scanning large parts of the data. The downside is that it only works for TH that support this mechanism, such as Intel SGX, or require implementation of a custom paging module. Note that for some TH such as Intel SGX, significant overhead is introduced by this mechanism, which we cover in more detail in section 6.3. Also note that some systems such as TrustedDB support switching between page-level and per-value encryption.

Per-block A third option is block encryption. Similarly to per-page encryption, data is encrypted while grouped together. The main difference is that instead of relying on the OS or a paging module to handle decryption, decryption is now handled by DBMS components such as the Buffer Manager or a scan operator. In per-block encryption the data is stored in a columnar database format with the columns encrypted in blocks of a configurable size. This granularity is used by systems such as ObliDB and Opaque. Per-block encryption allows for more fine-grained management of the encrypted data by making the query processor itself responsible for encryption and decryption. This has the advantage of being more flexible when designing a TH-EDBMS system as it allows more control of TH memory usage and easier integration of oblivious query processing operators. Additionally, the size of the encryption buffers is not tied to the page size, allowing the encryption block size to be tweaked for optimal performance depending on the workload. Finally, supporting different keys for different columns is implemented more easily in a per-block granularity than per-page.

Why is encryption of blocks/pages so much more efficient? There are two main reasons why encrypting at a larger granularity is much more efficient. The first reason is that most semantically secure symmetric encryption schemes require storing some extra data for randomization, authentication and key management. The size of this data ranges from several bytes to more than 60 bytes per encrypted buffer. For example, the encryption

used by Microsoft for their Always Encrypted service uses AES in CBC mode resulting in 4 byte integers encrypting to ciphertexts of 65 bytes. For an encrypted column of integers this results in a storage overhead of 16.25x. In table 5.1 several common symmetric encryption schemes are shown with their relative ciphertext storage size. The second reason

	Buffer size				
Cipher	4096B	1536B	576B	64B	8B
ChaCha8	100.2%	100.5%	101.4%	125.0%	200.0%
AES-CTR128	100.3%	100.8%	102.1%	118.8%	250%
xSalsa20	100.6%	101.6%	104.2%	137.5%	400.0%

Table 5.1: Relative ciphertext storage size for some common symmetric stream ciphers (12)

encryption is significantly more efficient on larger buffers is that encryption algorithms have an initialization overhead. Since this initialization overhead occurs only once per encrypted buffer, this overhead is amortized when buffer size increases. In table 5.2 the performance in cycles per bytes is given for several common symmetric encryption schemes. These benchmarks are taken from the eBACS benchmark results of an 2019 Intel Xeon Gold 6248 CPU (12).

	Buffer size					
Cipher	Long	4096B	1536B	576B	64B	8B
ChaCha8	0.28	0.29	0.39	0.53	2.75	14.50
AES-CTR128	0.63	0.68	0.76	1.11	4.03	32.25
xSalsa20	0.69	0.80	1.13	2.16	11.47	141.75

Table 5.2: Cost of encryption in cycles per byte for some common symmetric stream ciphers (12)

Encryption options in EDuckDB Now we know what options we have for encryption and what is chosen in the related work, we can draw some conclusions for EDuckDB. Firstly, both block and page encryption could be viable for EDuckDB, the approaches from related literature using these encryption granularities have no fundamental limitations with regards to the requirements, as identified in section 5.1.2.1.

For per-value encryption however, this is not the case. We will now explain the reasoning behind this. While StealthDB and Microsoft AEv2 have demonstrated that implementing a per value encrypted TH-EDBMS can be done relatively easily, these systems are row-store DBMS aimed at a OLTP workloads. In these workloads the most common types of queries are point look-ups and updates. For these workloads, only relatively little data needs to be

decrypted. For our requirements however, we need to support OLAP workloads. OLAP workloads generally consist of large scans, joins, filters and aggregates that process large parts of columns sequentially. To achieve good decryption performance on large scans, data should be encrypted in large granularities, as shown in table 5.2. As we will see later in our benchmarks using block granularity in section 5.3.2, even with decryption times of a few cycles per byte, overall performance overhead can be significant. Using per-value encryption would result in decryption times of 1 or 2 orders of magnitude larger which would result in very high performance overheads. Another problem with per-value encryption for EDuckDB is the large storage overhead. In efficient, in-memory OLAP DBMS, memory bandwidth is a scarce resource. To efficiently make use of the available bandwidth, compression techniques are often used. Due to this compression, data size of 1 or 2 bytes per-value are very common. For data sizes this small, storage overheads can easily reach between 10x-30x for common encryption schemes.

5.1.2 Encryption algorithm

The second main consideration for EDuckDB is which cryptosystem to use. The cryptosystem is the combination of encryption/decryption algorithm, key size, initialization vector/nonce generation, and optionally authentication scheme. The choice of these cryptosystem parameter can have significant consequences on its performance and security. For example, the performance overhead of using AES-CTR256 over AES-CTR128 is in the order of 30-50% on modern hardware. Security-wise, the selection of the randomization components such as the IV or nonce needs to be carefully chosen to prevent attacks such as nonce reuse attacks (92). For selection of the best cryptosystem we first analyse what we need for the requirements defined in section 5.1.2.1, then we analyse what is used in the EDBMS literature and common security protocols, and make a selection of schemes to use throughout further experiments.

5.1.2.1 Requirements

For EDuckDB, we have several requirements regarding the cryptosystem. Firstly, the cryptosystem should be symmetrical. Symmetrical cryptosystems have much better performance and require smaller key sizes. Also because TH is used, asymmetric encryption is not necessary for inserting values from the server-side, since we can rely on the TH to handle this. Secondly, we need both an unauthenticated and an authenticated scheme. Since our requirements of EDuckDB specify both the HbC and malicious trust models,

we need the unauthenticated scheme for maximum performance when assuming the HbC model and the authenticated scheme for data integrity under the malicious model. Thirdly, the scheme should ideally allow random access decryption to allow optimizations that can decrypt only part of an encrypted buffer. Random access decryption can also allow parallelization of the decryption process. Fourthly, the encrypted buffers need to be able to be updated easily and securely under the threat of a persistent attacker. Finally, the overhead of the scheme should be minimal. This should hold for both architectures with hardware AES support and hardware without it.

5.1.2.2 Considered schemes for EDuckDB

AES When choosing a symmetric cryptosystem most common choice is AES. AES is the standard algorithm for symmetric encryption chosen by the US National Institute of Standards and Technology (NIST) in 2001. This scheme is widely regarded as the standard symmetric encryption scheme. This is reflected by widespread protocols such as TLS 1.3 that define AES as the default cryptosystem. Also in EDBMS literature analyzed in Chapter 3 we find that all of them use AES when strong, randomized encryption is required. The next choice is which encryption mode to use. AES supports many different modes for a variety of use cases. Looking at the TH-EDBMS literature we find that the GCM and CTR modes are used primarily. With GCM being an authenticated encryption mode and CTR an unauthenticated, the combination of these schemes matches our requirements well: they support random access, allow parallel decryption for performance and, when using hardware acceleration, are among the fastest symmetric schemes. The specific schemes we will consider are AES GCM128 with 16byte tag and 12byte IV and AES-CTR128 with 12 bytes IV as these values are being specified as the most suitable specified in the NIST standard (92).

Salsa/ChaCha One main problem with AES is that its performance without hardware acceleration is relatively slow. An alternative to AES that can be an improvement when no hardware acceleration is available is the Salsa/ChaCha family of ciphers (93). These ciphers perform significantly better without hardware support, and on modern server CPUs may even perform slightly better than hardware-accelerated AES (94). Partly for this reason Salsa is the only non-AES cryptosystem included in the TLS 1.3 standard. From the Salsa family we select several schemes for testing EDuckDB. Firstly, we pick Salsa20 as it is the default stream cipher from the NACL library by Bernstein (95). Also we include Salsa20/8, which is a reduced round version of the same algorithm. A reduced-round

cryptosystem will trade security for performance. While not directly making a scheme insecure, reduced-round cryptosystems have a higher chance of being broken in the near future than their full-round counterparts. Finally we select XSalsa20, which is a modified Salsa20 that allows longer, 24byte IVs. This allows randomly generating IVs without ever needing to re-key. This can be useful in an EDBMS as it allows insertion and updating without needing the keep track of how often a key is used. For encryption schemes with shorter IVs such as AES-CTR, GCM or Salsa20, there are limits to how often they can be invoked for encrypting data to prevent a nonce reuse. This number of invocations is generally very large, e.g. 2^{32} for AES-GCM (92). However, in the context of databases, this limit can be reached easily: for illustration lets assume buffers of 4KB are encrypted with AES-GCM. The key lifespan will be reached after $2^{32} \times$ 4KB $\approx$ 17.18TB is written to the database with the same key.

Cryptosystem evaluation Now we evaluate the performance of different cryptosystems on our experiment machine to determine the cost of decryption. The specifications for this machine are listed in Appendix A, and will be the machine used for all further experiments. The encryption performance is crucial to query performance as we will discuss in section 5.2. In Figure 5.1 the results of our microbenchmark is shown. For this benchmark a buffer of a variable size is repeatedly decrypted until 1GB of data has been decrypted in total. The cryptosystem implementations used are taken from three different sources: Firstly for AES-CTR and AES-GCM the OpenSSL library is used. Then for Salsa, the NACL library is used. This NACL libary is also used for an non hardware accelerated version of AES-CTR. Finally, an experimental ChaCha8 AVX[1] implementation is used from the Supercop library (96). ChaCha8 is a variant on the reduced round Salsa20/8 and this experimental AVX implementation should give a good indication of the maximum Salsa/ChaCha performance. In the results we can clearly see the effect of the initialization cost for all cryptosystems. Furthermore, there is a slight slowdown visible as the buffers exceed the CPU cache size at >= 16MB. In the context of EDuckDB we can conclude that for unauthenticated encryption, using buffers of roughly 265 bytes or larger will be sufficient to achieve near-maximum performance. For unauthenticated encryption using AES-CTR128 we can expect a decryption cost of around 0.7cpb for optimal buffer sizes, while for authenticated decryption this is around 1cpb. For the Salsa ciphers, we can see that the standard Salsa20 performs significantly worse than AES-CTR. Additionally, the

[1] AVX is an extension to the x86 instruction set to allow a single instruction to process multiple values per instruction (SIMD).

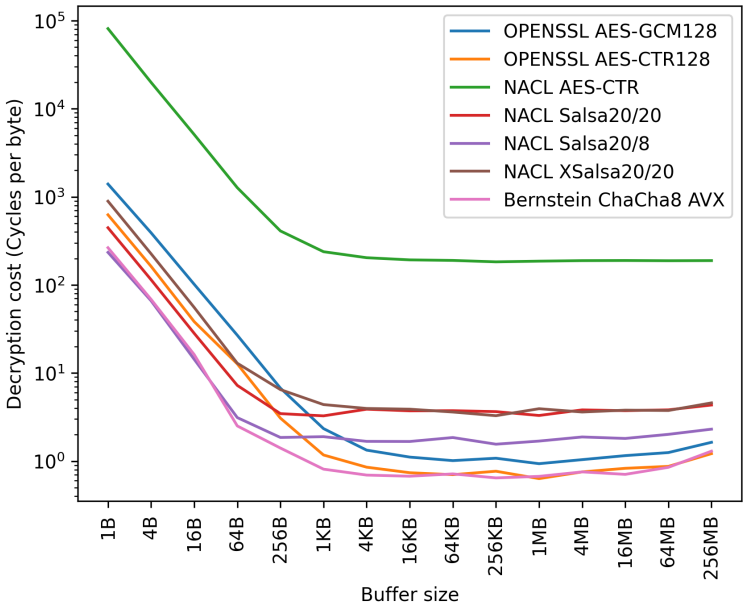


Figure 5.1: Decryption cost microbenchmark

ChaCha8 AVX implementation is only slightly faster than AES-CTR due to the CPU not supporting the latest AVX-512 SIMD instructions, which provides up to 2x performance increase over the older AVX2 instructions.

5.2 Implementing Encrypted DuckDB

For implementing encrypted query execution, there are two main execution models that we consider: Block-level encryption and Vector-level encryption. Both execution models encrypt batches of values together, but the difference is which DBMS component performs the decryption.

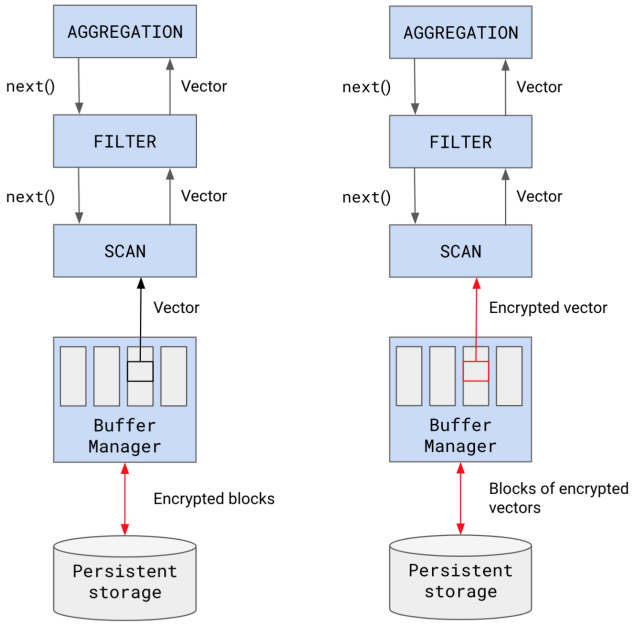
5.2.1 DuckDB memory management

To understand the two encrypted implementations, we first need to cover the basics of DuckDB memory management. In DuckDB, a buffer manager is in charge of handling memory management for the database. The buffer manager hands out memory buffers that can be used by the database internally. These buffers are used to store the actual data that is stored in the database, but is also used to store intermediate data or when operators need additional memory to perform their operation. For example, when a hash join is performed, the hash join operator stores its hash table in buffers from the buffer

manager. Buffers handed out by the buffer manager can be of two types: Buffers or Blocks. Buffers are used for data that only needs to reside in memory, such as intermediate data and in-memory tables. Blocks are used for data that needs to be backed by persistent storage. Blocks can be read from and written to storage through another component of DuckDB, the Block Manager. The Block Manager manages the reading, writing and allocating of blocks to the storage. It also makes sure the blocks are not corrupted in storage by calculating a checksum of the data on writing and verifying the checksum on reading. Via the Block Manager, the Buffer Manager is able to read data from storage when it is requested by an operator, or write back buffers to storage when they were modified by an operator. DuckDB can run in two different modes: as a transient, in-memory database, or as a persistent, file-backed database. As an in-memory database, all data in the database is stored in buffers, and the Block Manager is never used. As a file-backed database, DuckDB stores its data in a single file through the Block Manager. Finally, to allow for efficient I/O, DuckDB splits columns into separate parts called Segments. Each segment holds a contiguous part of the column starting from a certain offset. By default the block size in DuckDB is set to 256KB, which means that for the default vector size of 1024, each segment of a 4-byte integer column holds 256KB / 4KB = 128 vectors.

5.2.2 Encrypted blocks

The first implementation is block encrypted EDuckDB. For this implementation, a small modification is made to the Block Manager: instead of only calculating/verifying a checksum of the data as its read/written to disk, the Block Manager also encrypts/decrypts the data. This means that all data that is stored on disk, including the database headers, are encrypted. For the block size, the default value of DuckDB is 512KB, which is sufficiently large for efficient encryption/decryption as we saw in figure 5.1. At this block size encryption has a negligible storage overhead between 0,0015% and 0,0053% for our selected cryptosystems. Figure 5.2(a) shows a schematic depicting block encrypted EDuckDB processing a simple query containing a filter and an aggregate. Encrypted blocks are pulled into memory by the buffer manager through the Block Manager where they remain unencrypted in memory for the lifetime of the buffer. This implementation is globally identical to the architecture of CryptSQLite (19). It should be noted that Block-level encryption only has an effect when running in file-backed mode, as for in-memory databases DuckDB never stores data in blocks.



(a) Block-level encryption

(b) Vector-level encryption

Figure 5.2: EDuckDB execution model

5.2.3 Encrypted vectors

The second baseline implementation is vector encryption. As explained in section 2.1, vectors are a unit of query execution in vectorized query engines that group together batches of tuples for performance reasons. In vector encryption, we propose to use these vectors as the granularity of encryption. Instead of encrypting the blocks entirely, such as in block encryption, the data is encrypted one vector at a time by the vectorized insert operation. Decryption is performed by the vectorized scan operator. The execution model for a simple analytical aggregation query is depicted schematically in figure 5.2(b). For each encrypted vector, a separate IV and/or tag is stored with the encrypted data. The data for an encrypted segment therefore consists of a consecutive array of encrypted buffers each containing the data for a single vector. This layout is shown in figure 5.3. Vector encryption has several advantages over encrypting blocks. Firstly, by encrypting smaller buffers, the re-encryption cost of updating data in a column is lower. When a modification is made to an encrypted block, the whole block needs to be re-encrypted. For vector encryption this cost is significantly lower and also configurable through adjusting the vector size. Secondly, vector encryption allows implementing secure memory-efficient code when

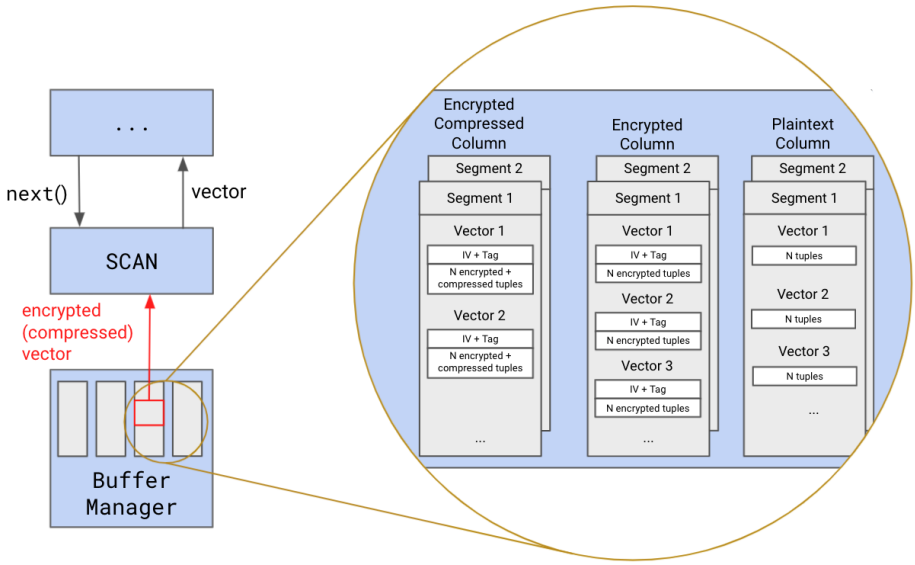


Figure 5.3: Encrypted vectors storage layout

integrated into a TH-EDBMS. This is because only the vector that is currently being processed needs to be decrypted, while the block encrypted execution model does not allow for this fine-grained management of secure memory. A possible weakness of vector encryption compared to block encryption is that the reduced buffer size will lead to higher storage and performance overhead. However, this is not a significant difference as even at the default vector size of 1024, both encryption throughput and storage overhead are very close to optimal. For example, for a vector of 4-byte integers, the resulting 4KB buffer has a storage overhead between 0.20% and 1.37% for the encryption schemes we have selected in section 5.1.2.2. Also, in figure 5.1, we have already seen that 4KB is enough to achieve near optimal throughput. Another encryption granularity that was considered is segment encryption, where not the vectors, but the segments are encrypted as single buffers. This would result in a lower storage overhead and allow for even higher query performance. Single vectors could be read from the encrypted segments by using the random access decryption functionality of our selected encryption schemes. However, the segment encryption has negligible performance gains over vectorized encryption, as performance is already near-optimal, while suffering from the same re-encryption costs as block encryption. Therefore, segment encryption was discarded as vector encryption is a more balanced trade-off. Finally, the concept of vectorized encrypted query processing, as presented in this section, is new in EDBMS literature. The closest work, encryption granularity-wise, is ObliDB. In their work, columns are encrypted in sections of 512 tuples,

but their execution model uses the value-at-a-time model.

In vector encryption, besides encrypting the data itself, there another component that contains information on the contents of columns, the zone-maps. Zone-maps are a well-known optimization technique in analytical query processing that allow scans with with a filter predicate to skip parts of a column during a scan. In DuckDB the zone-maps are implemented as the Min/Max value for each segment. This means that for each segment, DuckDB keeps track of the smallest and the largest value in that segment, which allows a scan to skip the segment if none of the values in the segment can match the filter predicate. Since the zone-maps contain data directly based on the contents of the columns we also encrypt these zone-maps. This means that on each zone-map lookup, an additional decryption step is performed. Finally, for this research only the numeric encrypted segments in DuckDB have been implemented, the string segments are left to future work. This means that for the remainder of this research, this implementation and the SGX-based implementations that are based on it, only support fixed-length data types such as integers and floats.

5.2.4 Adding compressed execution

To reduce the decryption overhead associated with query processing over encrypted data, we propose to use columnar compression. As discussed in section 2.1, compression is a common technique in OLAP DBMS design. For EDBMS, reducing the total data volume becomes even more beneficial as it will reduce the total amount of data that needs to be decrypted upon querying. In the context of OLAP workloads, a compression factor of more than 3x can be expected (29) using compression schemes such as PFOR, PFOR-DELTA, and PDICT, reducing the total amount to data to be decrypted on scanning by an equal amount. Vectorized decompression algorithms can be implemented efficiently and achieve low decompression times, lower than the decryption time of our selected encryption schemes. Therefore, the use of columnar compression in a vector encrypted database is likely to result in a net performance gain. We note that the combination of compression and encryption will only be useful when the data is compressed first and then encrypted, as the encryption process maximizes the entropy of ciphertexts meaning that they will not be compressible to any meaningful extend. Another important note is that the use of compression introduces a new way of indirect information leakage: If an attacker knows the compression ratio of an encrypted vector, he can deduce information on the contents of that vector as the compression ratio directly depends on the contents. While we will

remain aware of this newly introduced leakage, we leave a thorough security analysis of the information leakage associated with encrypted compressed columns to future work.

5.3 Evaluation

In this section we experimentally evaluate the two encrypted implementations of DuckDB. The goal is to first establish the performance efficiency of the vector encryption by comparing it to the block encryption implementation. With the efficiency of vector encryption established, we will use vector encryption as the baseline encryption to compare against the SGX-based implementations in Chapter 6. The second goal of the evaluation is to quantify the overhead of encryption in DuckDB. Finally, aim to quantify the impact of columnar compression on the decryption overhead and demonstrate how effective it can be in reducing decryption overhead. To do this, we start by defining the benchmark setup used in section 5.3.1, followed by the results in section 5.3.2

5.3.1 Benchmarking setup

For evaluation of the different implementations, benchmarking is done using the built-in DuckDB benchmark runner. This benchmark runner allows running several microbenchmarks, as well as the OLAP benchmark suites TPC-H and TPC-DS. The benchmark runner generates a test database, optionally performs a cold run to warm the caches, and then runs a test query for a configurable number of times to get an average result for each query. In all our benchmarks the default 5 runs per query were performed. As a benchmarking suite, we chose to focus on the TPC-H suite. TPC-H is the industry standard benchmark for OLAP-optimized DBMSs and aims to illustrate decision support systems that examine large volumes of data through complex queries (43). TPC-H is also the primary benchmark found in OLAP-oriented EDBMS literature (5)(19)(52)(4). To be able to evaluate both the block encryption and vector encryption implementations, we use two different modes. The experiments are run on the machine specified in Appendix A, which is used throughout this thesis.

File-backed mode For this mode, the database is first generated into CSV files. Then DuckDB is started in file-backed mode and all data is inserted into their respective tables. DuckDB is then restarted to start the checkpointing process and write all data into blocks. Now between every run of the benchmark, DuckDB is restarted to clear the tables stored in memory and CPU cache. A single cold run is done to make sure the OS page cache

is warm and we get consistent results across the 5 benchmark runs. The goal of the file-backed benchmark mode is to be able to compare the block encryption and vector encryption implementations.

In-memory mode The in-memory benchmark mode follows mostly the same setup as file-backed with a single modification: DuckDB is not restarted between benchmark runs. This means that the cold run will cause the buffer manager to keep all necessary blocks in memory and the CPU caches warm for all the benchmark runs. Since all benchmark in this research have a DB size smaller than the available memory on the system, this mode tests performance of the system as an in-memory database. The goal of the in-memory benchmark mode is to represent both the in-memory mode of DuckDB, as well as the file-backed mode of DuckDB where data has already been loaded into memory by the Block Manager.

Compressed execution To evaluate the effect of compression in EDuckDB, ideally we would use actual compression functionality in DuckDB. However, at the time of this research, compression has not been implemented yet, as it is expected to be implemented in DuckDB later in 2021. To be able to evaluate compression in the current version of DuckDB, we added a query to our TPC-H benchmark suite to emulate compressed storage and compressed execution. The query is based on TPC-H Q06 and emulates how DuckDB would process Q06 if it were to support compressed execution using the PFOR compression scheme. To achieve this, the columns used by Q06 were manually converted to a smaller data type that would still fit the data: `l_shipdate` was converted from DATE to SMALLINT, `l_discount` from DOUBLE to TINYINT, `l_quantity` from INT to TINYINT and `l_extendedprice` from DOUBLE to INT. Then a new query was added to the benchmark suite named `Q06_C`. This query is identical to Q06, except that it searches the compressed columns and it decompresses the data before the aggregation operator:

```
SELECT SUM(CAST(l_extendedprice_compressed AS BIGINT) * CAST(
    l_discount_compressed AS BIGINT)) AS revenue
FROM lineitem
WHERE l_shipdate_compressed >= 1994 AND l_shipdate_compressed < 1995 AND
    l_discount_compressed BETWEEN 5 AND 7 AND l_quantity_compressed < 24;
```

The average resulting compression ratio for the 4 columns involved in `Q06_C` is 3x, which is representative for PFOR on typical OLAP workloads (29).

Fixed-length data type queries As stated before, we have limited our scope to fixed length data types for the vector encryption. This poses a limitation when evaluating the implementation using the standard TPC-H benchmark as only 2 of the 22 queries operate on only fixed-length data: queries 4 and 6. As a work-around, we extended TPC-H with a set of fixed-size data queries. In TPC-H, several columns of the VARCHAR type contain strings from a very limited set of possible values. For example, `l_returnflag` has only 3 possible values: "A" "N" or "R". 10 of these columns are duplicated into equivalent dictionary encoded columns. For example, `l_returnflag` is duplicated into `l_returnflag_dictkey` with all "A" values encoded as integer key 0, "N" gets key 1, etc. With these additional columns, 8 queries from the TPC-H benchmark can be rewritten into fixed-size queries that are functionally equivalent. The modified queries are post-fixed with `_MOD`: `Q01_MOD`, `Q02_MOD`, etc. Combined with queries that already operate of fixed-length data types, this results in a total of 10 queries in our fixed-length data type benchmark suite. To evaluate the representivity of the fixed-length modified TPC-H queries, we ran the benchmarks both in their modified version and the original query. As can be seen in figure 5.4, for some queries such as Q03, the fixed-length variants

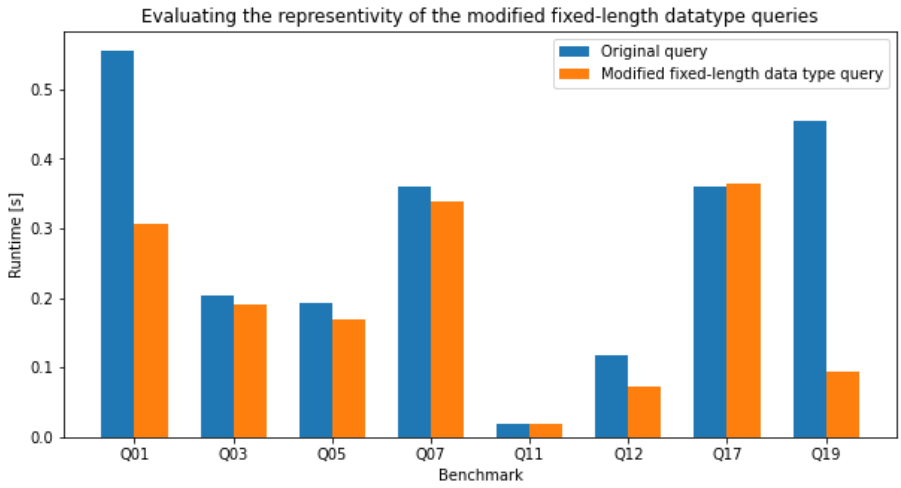


Figure 5.4: Modified fixed-length data type queries are partially representative to original queries

perform close to the original query, while others perform very differently. With this result, we show that most of the selected queries are representative to their original query, whilst other should be seen as completely different queries.

5.3.2 Results

5.3.2.1 Comparing block encryption to vector encryption

To compare the performance of the block encryption and vector encryption implementations, the fixed-length data type queries were ran on both implementations. These queries are chosen as the vector encryption implementation currently does not support variable-length data types. The benchmarks were run in file-backed mode because the block encryption implementation decrypts data as its read from the file. The vector size is left to the default value of 1024 and the cryptosystem selected is OpenSSL AES-CTR with it being the fastest scheme. The scale factor is set to SF1, which results in a database of roughly 1GB in size. The results of this experiment are shown in figure 5.5. In this graph the run-times are normalized to the unencrypted DuckDB run-times. From the results we can see that both vector encryption and block encryption perform very similarly, with some minor differences. There are some differences between the two methods, which can be explained by some queries benefiting more from ability of vectorized encryption to limit unnecessary decryption, while for other queries, the slightly improved decryption throughput of block encryption is an advantage. Either way, from these results we can see that the vector encryption implementation is a viable approach to query processing over encrypted data when compared to the baseline block encryption. To further improve performance, an optimization was attempted in the vector encryption implementation to allow partially decrypting vectors using random access decryption within the vectors. However, this optimization did not provide any measurable performance increase but did come at the cost of increased complexity and was therefore abandoned.

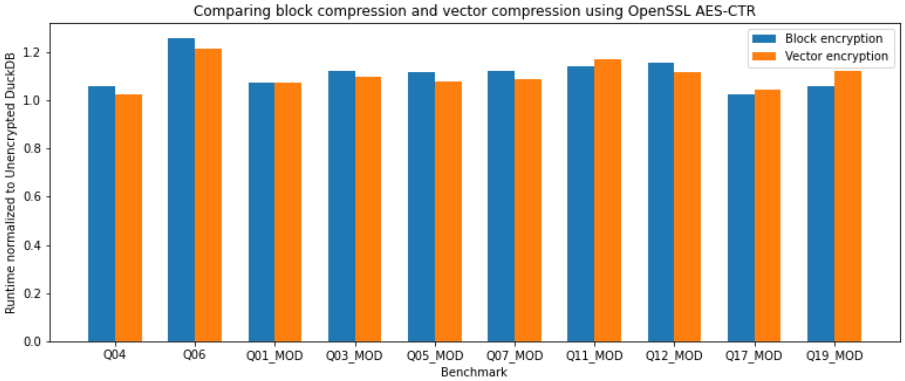


Figure 5.5: Vector encryption outperforming block encryption for most queries

5.3.2.2 Quantifying overhead of vector encryption

With vector encryption established as a viable approach, we now evaluate the decryption overhead for vector encryption when using different cryptosystems. The goal is to quantify the expected overhead of encrypted data in vectorized query processing. This quantification will later be used as a baseline for the SGX-based implementations to be able to determine how close they are to optimal. Additionally, we aim to demonstrate the importance of maximizing decryption throughput on query performance. The experiment that was done contains the fixed data type queries, this time in in-memory mode. In-memory mode is faster that file-backed mode and therefore the relative overhead decryption is the largest, giving a realistic measurement for when EDuckDB is used as an in-memory database. The vector size is again set to the default 1024 and the scale factor is SF1. Figure 5.6 shows the results of these benchmarks. The results show that a large variance exists between the different schemes with an average overhead of 22% for AES-CTR and 102% for xSalsa20. This demonstrates the importance of selecting a fast cryptosystem for efficient query processing over encrypted data. The results also show that the cost of authentication through the AES-GCM scheme comes at an overhead of 36% on average, which shows that at a relatively small additional overhead, data integrity can be protected, which is essential when building EDBMS resistant against active attackers.

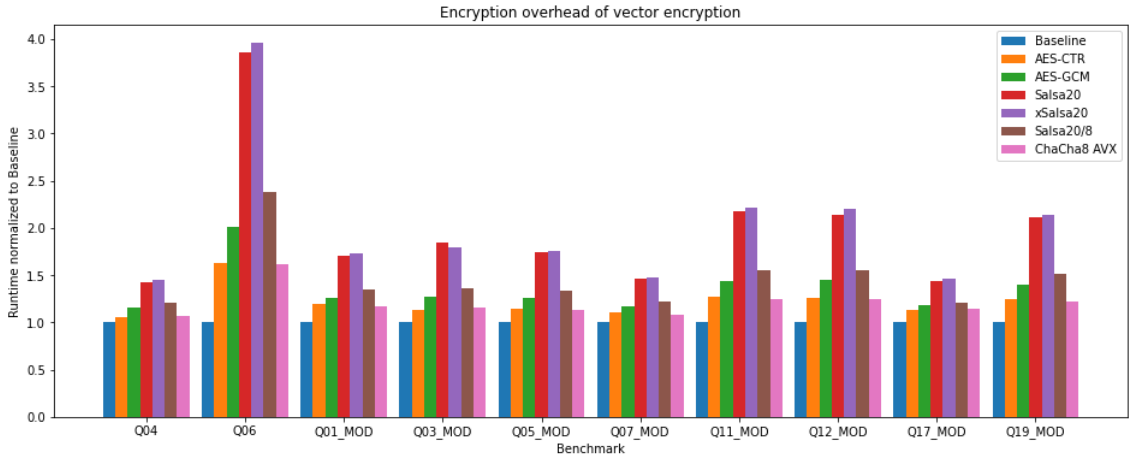


Figure 5.6: Vector encryption outperforming block encryption for most queries

5.3.2.3 Impact of compression on encryption overhead

Finally, we analyze the impact of columnar compression on the encryption overheads that we have seen so far. To do this, we used the emulated compressed version of `Q06` as described in section 5.3.1. The emulated compressed version was run together with the uncompressed version for all considered cryptosystems. The scale factor and vector size are set to 1024 and SF1 respectively. In figure 5.7 we have shown the results of the experiment. From the results, we find that the impact of compressed execution with a compression ratio of 3x on encryption overhead is very significant. As is to be expected, especially, the slower cryptosystems benefit from the reduced data volume. However, even for OpenSSL AES-CTR, the decryption overhead decreases from 63% to only 30% over unencrypted DuckDB. If we combine this result with the results from 5.3.2.2, we can deduce that the worst case decryption overhead for all queries is 30% if compressed execution is used. Furthermore we find that the average reduction in absolute encryption overhead is 2.46× ranging between 2.12× for AES-CTR and 2.78× for the slowest scheme, xSalsa20. These results clearly show the benefit of columnar compression in encrypted query processing. An interesting experiment we leave to future work is to run the same experiment with actual compressed execution for all TPC-H queries when DuckDB has implemented compressed execution.

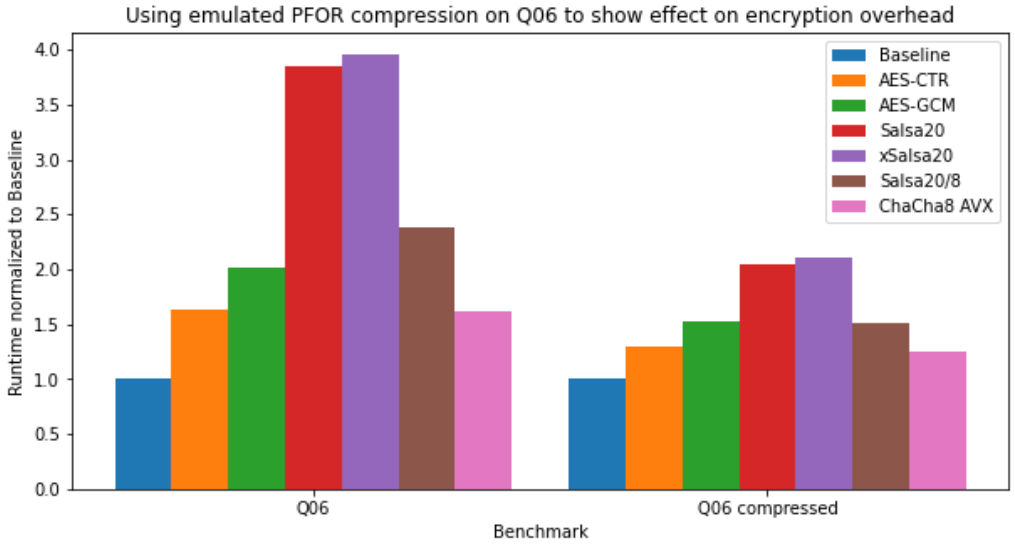


Figure 5.7: Using compressed storage and compressed execution significantly reduces encryption overhead

5.3.2.4 Summary

We now summarize the main findings of this section. Firstly, both vector encryption and block encryption can allow query processing over encrypted data in DuckDB at a reasonable, but non-negligible overhead. This, combined with its architecture where memory stays encrypted in-memory, makes vector encryption a suitable base for an TH-EDBMS based on DuckDB. This will be further explored in Chapter 6. Secondly, we have demonstrated that columnar compression, a technique well-known in database research, can significantly reduce the overheads from query processing over encrypted data. Due to the high efficiency of modern vectorized decompression algorithms while still maintaining good compression ratios, total data volume can be reduced resulting in less data to be decrypted. With the efficiency of modern columnar compression schemes, this can result in an overall decrease of decryption overhead. In our experiments using emulated PFOR compression, a reduction in overhead of 2.12× was seen for AES-CTR.

6

Encrypted DuckDB with Intel SGX

In this chapter, we build on the basis established in chapter 5 to design a DuckDB-based TH-EDBMS. Firstly, we will discuss the most suitable TEE for our requirements. Then, the different options for splitting a DBMS in a secure and unsecure part are discussed. Thirdly, we present two different, viable approaches along with the prototypes we implemented to validate them. Finally, we validate the prototypes by experimentally evaluating their performance and analyzing their security.

6.1 TEE selection

As a TEE, we selected Intel SGX for this research. In this section, we will first explain this decision by exploring the alternatives. Then, a brief overview of SGX is given. Finally, an analysis of the main performance bottlenecks for SGX systems are evaluated.

6.1.1 SGX alternatives

Intel SGX is an obvious choice in TEE for several reasons. Firstly, it is the most widely used TEE in recent EDBMS literature and is well documented. Secondly, it is available on most recent x86 CPUs which are found in both server and consumer hardware, which makes it both interesting from a practical research point of view, as well as having strong real-world relevance. However, SGX is not the only option and we will briefly touch on the most relevant alternatives here.

AMD SEV-SNP Intel's main competitor on the x86 market, AMD, offers a TEE-like technique similar to SGX. The technique is called AMD Secure Encrypted Paging with Secure Nested Paging (SEV-SNP) and aims to offer protection for virtual machines

(VM) running on cloud hardware. AMD SEV-SNP also offers memory encryption and remote attestation, but differs from SGX slightly. For example, SEV-SNP has a different trust model. Instead of a small piece of code that runs inside the enclave, SEV-SNP protects the memory between VM and hypervisor and between VMs. This means that the guest operating system running inside a VM with SEV-SNP enabled needs to be trusted. SEV-SNP has two large advantages over SGX. Firstly, it does not require any modifications to applications. SEV-SNP is implemented in the hypervisor, guest OS and physical hardware, which means that applications can run without any modifications or even recompilation. Secondly, its performance is superior to SGX by a large margin (97), especially for applications that require lots of memory such as a DBMS. For this research however, SEV-SNP however was deemed less suitable for several reasons. Firstly, the amount of research available on AMD SEV-SNP is significantly smaller to that of SGX both with regards to EDBMS literature as well as literature on the details of AMD SEV-SNP itself. Secondly, the remote attestation process as is currently implemented has been shown to be broken (98). The remote attestation step is crucial for the use cases specified for EDuckDB. Thirdly, due to the trust model of SEV-SNP including the host operating system, the TCB is significantly larger which limits the maximum security that can be achieved. Finally, using SEV-SNP is less interesting from a database research perspective, as running a DBMS application in SEV-SNP does not require any modifications. While not chosen as TEE for this research, SEV-SNP is an interesting technology as we will further discuss in section 7.2.

ARM TrustZone Another available TEE technology is ARM TrustZone. TrustZone was launched in 2004 and provides security primitives necessary to implement different types of trustworthy systems. While TrustZone supports creating a TEE similarly to SGX, it lacks the sealing and attestation features of SGX (99). As mentioned before, the attestation feature of TrustZone is important for the EDuckDB use cases and threat models. Furthermore, TrustZone research is mainly focused on mobile and IoT applications as this is what ARM chips are commonly used for. This has resulted in a significantly lower amount of interest in TrustZone from EDBMS researchers with only limited works available (100). Finally, as x86 is, at least for now, the leading architecture in the cloud, our main use case, we decided against using TrustZone for this research.

Academic TEE solutions In TEE research, several academic systems have been proposed, most notably Sanctum (74) and AEGIS (101). While these solutions are very inter-

esting and sometimes offer significant improvements over industry solutions, using them experimentally is significantly more challenging due to limited availability. Furthermore the real-world relevance of an academic TEE-based EDBMS is limited when the required TEE is not widely available on cloud hardware.

6.1.2 SGX overview

We will first give a brief overview of SGX. SGX is a set of hardware instructions that allows the creation and management of special regions in memory called enclaves. The memory set aside by SGX is called the Processor Reserved Memory (PRM). The PRM is protected by the CPU from all non-enclave accesses, including those from the kernel or hypervisor and DMA access from peripherals (102). Inside the PRM resides the Enclave Page Cache (EPC). The EPC holds all code and data that belongs to enclaves. Data inside the EPC can only belong to a single enclave and enclaves cannot access EPC pages from other enclaves. Enclaves are initialized by untrusted software running on the host OS. This software consists of the SGX Driver and the SGX Platform SoftWare (SGX PSW). On initialization of an enclave, this untrusted software allocates secure memory by assigning pages to a newly generated enclave id. The initial state of the enclave containing the code and data is then copied into the secure memory by the CPU. After this copying process, the CPU calculates a cryptographic hash of the initial state called the *measurement hash* and the enclave is ready for execution. After initialization, a remote party can engage in a software attestation process to verify both the enclaves measurement hash and the integrity of the system it is running in. When an enclave is initialized, the only way for control flow to move between unsecure code and enclave code is through the special CPU instruction EENTER. With the EENTER instruction, the CPU makes sure that the control flow is moved to the enclave is a secure predictable way. When the EENTER instruction has succeeded, the CPU is now in *enclave mode* and can execute the code running inside the enclave. Code running inside the enclave can access the data stored inside the enclave to perform computation over it in a secure way, but can also access the unsecure memory. When the code inside the enclave wants to return control flow to the untrusted code, the EEXIT instruction is used. Note that this explanation is a simplification of the entire process: the code for entering, exiting, and hardware exception handling is very complex, for a detailed understanding we refer to the excellent work by Costan et. al. (102) who explain SGX in great depth in over 100 pages. To allow development of applications for SGX without requiring programmers to understand the complexities of using the SGX instructions directly, Intel has provided the Intel SGX Software Development Kit (SGX

SDK). The SGX SDK provides an abstracted, easy to use, interface for the programmer to specify in what ways the enclave can be entered and exited. These abstractions are called Enclave Calls (ECall) and Outside Call (OCall). ECalls are predefined function in the enclave that can be called from outside the enclave. OCalls are the opposite, i.e. functions in untrusted code that can be called from the enclave code.

To develop an application for Intel SGX, the code and data for an application is split into a secure part and an unsecure part. The entry point is in the unsecure part which calls on the SGX PSW to initialize the enclave through functions defined in the SGX SDK. The ECalls and OCalls are defined in special header files written in the Enclave Definition Language (EDL). A simplified example application is shown in Listings 6.1, 6.2, and 6.3. In this example application, one ECall and one OCall are specified. To compile this application, the SGX SDK provides scripts to compile the `Enclave.cpp` and `Enclave.edl` into a shared object and header files. Then the `App.c` gets compiled with the generated header files. `App.o` is linked against the headers from the SGX PSW which provides the functions to initialize the Enclave.so shared object into a new enclave.

```
int main(int argc, char *argv[]) {
    initialize_enclave();
    ecall_secure_function("abc", 14);
}
void ocall_print_string(const char *str) {
    printf(str);
}
```

Listing 6.1: SGX example application: `App.c` (untrusted)

```
void ecall_secure_function(const char* str, int num) {
    OCall_print_string(str);
    return strlen(str) + num;
}
```

Listing 6.2: SGX example application: `Enclave.c` (trusted)

```
enclave {
    trusted {
        public void ecall_secure_function([in, string]const char* str, int num)
            ;
    }
    untrusted {
        void ocall_print_string([in, string] const char *str);
    };
}
```

Listing 6.3: SGX example application: `Enclave.edl`

EDL files such as the example in listing 6.3 specify the C functions that are callable as ECalls and OCalls along with *annotations* for the function arguments. To securely performance ECalls and OCalls, the arguments need to be verified on every call. This verification step prevents a malicious host from providing arguments of unexpected shape that can compromise enclave security. Verification of the arguments can be done in two ways: by data marshalling or by manual checks from the programmer. Data marshalling in SGX is done by code generated automatically by the SGX SDK using the annotations which specify how to marshal the arguments. This code verifies that the argument is of the expected shape. In the example application, the `ecall_secure_function` function has c string argument that is passed to the enclave. With the `[in,string]` annotation, the marshalling code that is generated will perform the following steps on each ECall: verify `str` is a pointer to a null-terminated array of `char`, verify that he entire array is located inside the untrusted memory, copy the array into secure enclave memory. In EDL, different types of annotations exist for different types of arguments, both for arguments used as input and as output. While data marshalling in SGX allows securely passing arguments to ECalls and OCalls, it can be very expensive due to the buffer always being copied in and/or out of the enclave on each call. For this reason, the data marshalling process can be disabled by using the `[user_check]` annotation. With the data marshalling disabled, the programmer of the enclave code is responsible for verifying the integrity of the buffers and to prevent any malicious manipulation by the untrusted host.

6.1.3 SGX Performance

For SGX, there are several causes of performance overhead: ECall/OCall overhead, EPC paging overhead, and increased LLC miss cost.

ECall/OCall overhead Every time the execution flow jumps between trusted enclave code and untrusted code through ECalls or OCalls, this causes significant overhead. This overhead has multiple causes, both direct and indirect. Firstly, some direct overhead is caused by the EENTER and EEXIT instructions. These instructions are complex instructions and both take thousands of cycles. Also the SGX SDK adds code to each ECall/OCall which adds an additional thousand cycles. The total overhead in cycles has experimentally been found to be roughly 8k cycles for OCalls and 9k for ECalls (67). This is around 50x slower than a regular system call. Note that this overhead does not even include the data marshalling of function arguments. Additionally, there are also indirect costs associated with ECalls/OCalls. Every time the EEXIT instruction is called, the TLB is flushed.

Flushing the TLB will result in more TLB-misses and consequently some overhead from walking the page tables. While this overhead will depend on the type of application, Orenbach et. al have performed an experiment doing hash tables lookups over a 2MB hash table and have shown that the performance overhead of these TLB flushes can be more over 200% (13).

Memory decryption on LLC misses Another important cause of performance overhead in SGX comes from last level cache (LLC) misses. In SGX, the data stored in the EPC is encrypted with authenticated encryption at all times. When an enclave accesses data from the EPC, its integrity is checked and it is decrypted before being stored in the CPU caches and usable by enclave instructions. This means that LLC misses for EPC memory accesses come at a significant overhead. Orenbach et. al. experimentally determined the overheads for LLC misses, their results are shown in table 6.1.

Operation	Sequential access	Random access
READ	5.6×	5.6×
WRITE	6.8×	8.9×
READ and WRITE	7.4×	9.5×

Table 6.1: Relative cost of LLC miss in enclave (13)

EPC Cache misses The final main source of overhead in SGX is the cost of paging when overflowing the EPC. In SGX, the PRM and consequently the EPC is of fixed size: for the CPU of our machine this size is 128MB, for the latest Intel CPUs it is 256MB. For many applications, and especially (EDBMS), this is a restrictively limited amount of memory. Fortunately, the SGX driver supports a paging mechanism to allow enclaves to use more memory than fits in the EPC. When under PRM pressure, the SGX driver evicts pages from the EPC by encrypting them and storing them in unsecure memory. When the enclave accesses a page that is not in EPC, the paging mechanism retrieves the page from unsecure memory by decryption and integrity checking before storing it in EPC. This process is expensive for two main reasons. Firstly, it requires a extra copying, decrypting and integrity checking step on each EPC miss similarly to when a LLC miss occurs, but now combined with a write to memory. Secondly, the paging mechanism is implemented in the SGX driver, which executes in user-land in non-enclave mode. This means that for every EPC miss, an OCall is required, which gets very expensive. Experimentally, the overhead of a single EPC page miss has been shown to be roughly 25k cycles for the paging

process, 7k cycles for the required OCall (13), and another 8k cycles of indirect slowdown due to the TLB flush and increased LLC cache pollution. As the indirect slowdown can vary per application, between 32k and 40k cycles is the expected total cost for EPC page misses.

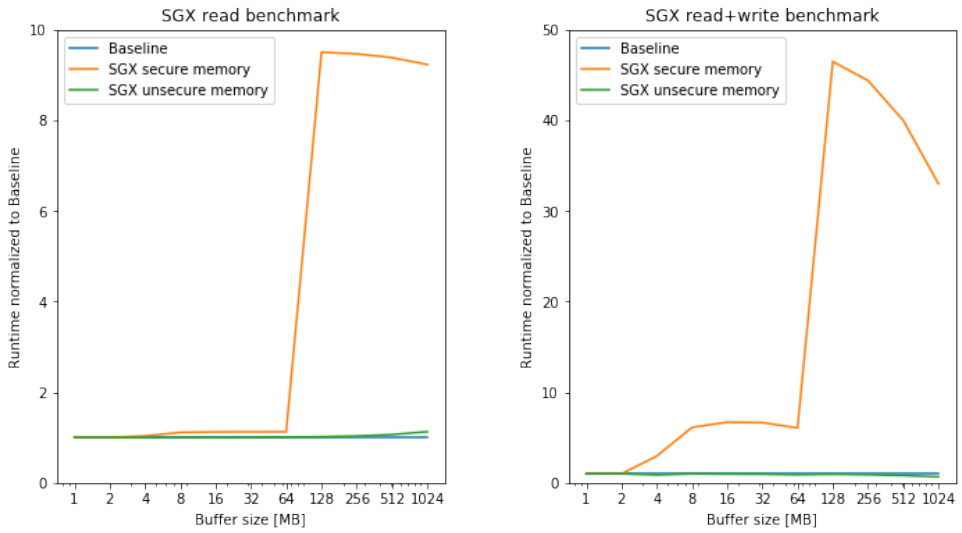
SGX benchmark To experimentally evaluate the performance of SGX, we ran two micro-benchmarks on the experiment machine specified in Appendix A. Both benchmarks are small C programs created with the SGX SDK. The first benchmark, shown in Figure 6.1(a), executes a simple read benchmark that repeatedly sums the values in a buffer. In the results, we can just see that for buffers over 2MB, a small performance overhead exists. This small overhead is presumably caused by the increased LLC miss latency. For buffer sizes larger than the available secure memory, we clearly see the performance overhead increase massively to around 9× due to EPC paging. For the read+write benchmark shown in Figure 6.1(b), a fixed integer value is repeatedly written to each position in a C array. Here we see a significantly larger performance overhead for buffers over 2MB. Similarly to the read benchmark, when the available EPC memory is exceeded, overhead increases massively. We note that for these benchmarks, the cost of ECalls/OCalls is not included in the results as the complete benchmarks are run from inside the enclave. The results demonstrate that, without the expensive ECalls or OCalls, accessing unsecure memory can be done without any significant penalty, while accessing secure memory can be very expensive, depending on the memory requirements.

6.2 Partitioning DBMS code for SGX

When designing any SGX application, an important design decision is how to partition the application in a secure and an insecure part. The choice of where to split the application will depend on the specific requirements of an application. In this section, different design choices for partitioning EDuckDB are explored.

6.2.1 Design choices

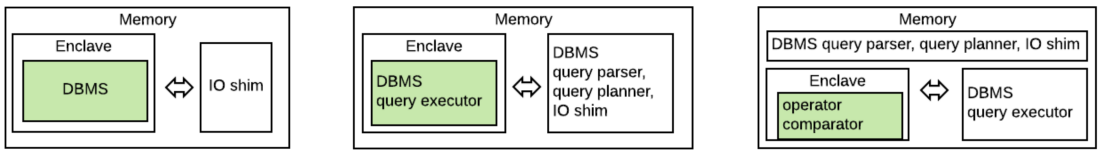
As described in earlier work (8), splitting an EDBMS can be done in various ways. In Figure 6.2 three models are described that can be used for the code split. Note that these models are not exhaustive as a DBMS can contain various components that can either be placed in the secure or the insecure part. In the full-dbms-split model depicted in Figure 6.2(a), as much as possible is placed inside the enclave, all regular DBMS components



(a) Read (b) Read and Write

Figure 6.1: SGX memory performance benchmarks

run inside the enclave and only a small I/O shim exists in insecure memory to provide the enclave code access to storage, network and/or user input. In the minimal-dbms-split model shown in Figure 6.2(c), the bare minimum is placed inside the enclave. Only the primitive operators (e.g. +, -) and comparators (e.g. <=, ==, !=) are implemented. The middle-dbms-split model in Figure 6.2(b) takes an intermediate, more fine-grained design approach where some components are placed into the enclave and some are placed outside the enclave.



(a) full-dbms-split (b) middle-dbms-split (c) minimal-dbms-split

Figure 6.2: Three design choices for partitioning an encrypted database. (8)

6.2.2 Partitioning evaluation

To analyze the partitioning design choices, a set of evaluation criteria are formulated based on the EDuckDB requirements and the SGX characteristics explored in sections 5.1.2.1 and

6.1. In table 6.2 each evaluation criterion is listed with their corresponding requirement as well as a description. Even though both the evaluation criteria and the partitioning design

Evaluation	Requirement	Description
TCB	Security	Total lines of code (LoC) that runs in enclave
Attack Surface	Security	Total amount and complexity of ECalls + OCalls
Leakage	Security	Information leaked in system to untrusted components
EPC Swapping	Performance	Total amount of EPC swapping
ECall/OCall	Performance	Total ECall + OCall count
Enclave execution	Performance	Total number of instructions executed in enclave mode
Simplicity	Implementation	Simplicity of DBMS application development
Integration	Implementation	Ease of integration of existing DBMS application

Table 6.2: Evaluation criteria for partitioning design

models are very high-level, some important findings can be drawn from this analysis. Here the most important difference between the models will be discussed

6.2.2.1 Security

For the security evaluation the first important difference is the size of the Trusted Computing Base (TCB). As mentioned before, the TCB is the set of components that are trusted. In secure system design, an important factor of security is to keep the TCB as small as possible. Clearly, the more components are placed in the enclave, the larger the TCB, and therefore the lower the overall system scores on security. Secondly, for the leakage criterion we can conclude that minimal-dbms-split scores the worst as it leaks information on comparator results by design. While full-dbms-split is by no means free of information leakage by design, it does allow integration with techniques to limit information leakage through for example using ORAM. Finally, for attack surface we can identify a difference but not a clear advantage to either model because while the minimal-dbms-split will most likely have a larger number of entry points into the enclave, each secure function call will be relatively simple. The full-dbms-split model suffers theoretically needs only one entry point, one that accepts a SQL query, that means that an untrusted string be parsed which is significantly more complex to secure well.

6.2.2.2 Performance

As we have seen in section 6.1.3, designing a high performance SGX application will require a design that is considerate of the performance pitfalls of SGX. Evaluating the performance

criteria, the first important difference is in the number of ECalls/OCalls. The full-dbms-split needs only a single ECall when starting the application and OCalls for accessing the host operating system for storage or network access. This number is expected to be very low, for example reading a block from disk would require a single OCall. The minimal-dbms-split requires an ECall for each individual comparison/operation which, in the context of OLAP workloads, will probably mean huge amounts of ECalls are necessary[1]. When considering the likeliness of EPC swapping however, we see that the full-dbms-split model has a high likeliness of triggering EPC swapping as it need to store all application code and memory inside the EPC. The minimal-dbms-split model is more likely to be able to efficiently limit EPC usage. Finally, as the full-dbms-split model runs most code in enclave mode, this model score low on enclave execution as it will likely incur more overheads due to expensive LLC misses than a minimal-dbms-split.

6.2.2.3 Implementation

For the implementation evaluation criteria, the two extremes full-dbms-split and minimal-dbms-split perform similarly. Minimal-dbms-split requires only rewriting very simple comparison and operation functions that can be integrated into an existing DBMS (8). Full-dbms-split can be done in one of two ways: either by recompiling the application with the SGX-SDK, or by using a LibOS solution like Graphene-SGX (10). Either approach will require little rewriting of existing code. The middle-dbms-split model scores poorest on this criterion as it will require significant restructuring of application source across a secure and an unsecure part. Also the interface between these parts will need to be define as ECalls/OCalls.

6.2.2.4 Overview

In table 6.3 an overview of the analysis is shown. The performance of each partition design is scored with '+', '+/-' and '-'. '?' is used to indicate that no meaningful score could be given.

[1] In OLAP workloads, the number of values processed by queries tends to scale linearly with database size. Due to the low cost of simple operators or comparators, the total cost of these queries remains feasible. For ECalls however, this does not result in a feasible performance as their cost is many orders of magnitude larger than that of basic operators or comparators.

Category	full-dbms-split	middle-dbms-split	minimal-dbms-split
1a: TCB	-	+/-	+
1b: Attack surface	+/-	?	+/-
1c: Leakage	+	?	-
2a: EPC swapping	-	?	+
2b: ECall/OCall	+	?	-
2c: Enclave execution	-	?	+
3a: Design simplicity	+	-	+
3b: Integration	+	-	+

Table 6.3: Evaluation of different partitioning designs.

6.2.3 Partitioning in existing EDBMS literature

To further analyze the choice of the partitioning design, several of the most relevant works on TEE-based EDBMS are analyzed.

StealthDB The paper on StealthDB (8) discusses this topic most explicitly and the basis of the analysis in section 6.2.2 stems from their work. In their analysis the authors directly compare the three models in Figure 6.2 on their suitability for an OLTP EDBMS. The authors reason that the full-dbms-split model is unsuitable due to the limited size of the EPC. Even with future releases of Intel SGX promising larger EPC sizes, the Merkle trees used by Intel SGX to protect memory integrity does not scale well. Regarding security, it is noted that this approach does have the potential to leak less data if the intel SGX side-channels are sufficiently mitigated. The authors consider the middle-dbms-split to suffer from the same scaling issue as full-dbms-split regarding the limited EPC as in the query execution step tables and indexes need to be loaded into the enclave. Additionally, even if the data necessary for query execution fits within the EPC limit, reading and deserializing data into the enclave is expected to have a performance overhead of at least 2.8x. Finally, the middle-dbms-split model makes the task of mitigating sidechannel attacks non-trivial. The minimal-dbms-split ends up being chosen for its sparing use of secure memory, low TCB size and easy mitigation of SGX sidechannel attacks. An important note is that the overhead of ECalls/OCalls is not considered a significant issue due to the intended OLTP workload of their system. The reason for this is that contrary to OLAP worloads, where the number of values touched by a query tends to scale linearly to the DB size, in OLTP workloads the number of values touched by queries is typically only constant or logarithmic with respect to the DB size.

EncDBDB EncDBDB (9) implements a range operator for an encrypted string column that is integrated into MonetDB, an OLAP DBMS. In the EncDBDB design, all data is stored encrypted into unsecure memory. Only the relatively small (1129 LoC) range operator itself is run inside the enclave. This approach is most similar to the minimal-dbms-model as used by StealthDB, except that it is implemented as at the physical operator layer instead of the primitive operator level.

EnclaveDB EnclaveDB (7) implements an OLTP EDBMS based on SQL server and a modified Hekaton engine. In their design, all tables, indexes, the modified Hekaton engine and a trusted kernel live inside the enclave while the query parser/optimizer, query processor and storage remain in unsecure memory. This layout mostly resembles the middle-dbms-split model. An important observation is that the entire database with indexes resides in secure memory. To be able to support datasets larger than the EPC limit without prohibitively slow paging overhead, the authors make the assumption that future implementations of intel SGX will support significantly higher EPC limits. However, at the time of this research, no source was found suggesting significantly larger enclaves will be supported in the near future[1].

ObliDB ObliDB (72) is a database prototype implementing various common database operators in an oblivious way using Intel SGX. These oblivious operators are combined in a prototype database with a secure query planner and supports common SQL functionality such as SELECT, INSERT, UPDATE, DELETE, GROUP BY. In their design, nearly all components are placed in the enclave while the data remains encrypted in unsecure memory. Since the query planner and query execution are placed in the enclave this design most closely matches the full-dbms-split model. The choice for this model can be explained by their security requirement of oblivous query exection. To allow for obliviousness ORAM is integrated in the query execution meaning that the query execution must be inside the enclave.

TrustedDB TrustedDB (40) is early EDBMS work using a IBM secure coprocessor as a TEE. TrustedDB splits the DBMS into two parts each with their own database engine. For the untrusted engine MySQL is used, while the IBM SCPU runs a modified SQLite engine. Even though the IBM SCPU used in this research has completely different performance

[1]Note that while writing this Thesis, Intel did release a version supporting significantly larger enclaves, which we discuss in Section 7.2

characteristics to Intel SGX enclaves, the design considerations are very similar: A paging mechanism is used to provide the SCPU with enough secure memory for large secure tables, execution inside the secure enclave comes at performance overhead and communication between the SCPU and CPU comes at a significant overhead.

6.2.4 Partitioning DuckDB

After evaluation in section 6.2.2 and analysis of partitioning in related literature in section 6.2.3, we can conclude that there is no clear winner to the partitioning problem from a high-level perspective for our requirements. Both the minimal-dbms-split and the full-dbms-split seem like viable options for this research, and we therefore focus our research on comparing these two models in their suitability for EDuckDB. Note that the middle-dbms-split will not be considered in this thesis as it is significantly more complex and does not fit the time constraints of this Master thesis. To compare the full-dbms-split and minimal-dbms-split models, two prototypes will be compared. Firstly, the full-dbms-split model will be analyzed in section 6.3. Secondly, in section 6.3 a prototype is analyzed which represents the minimal-dbms-split model, but does share some characteristics from the middle-dbms-split model as we will explain it that section.

6.3 DuckDB entirely in enclave

In this section, we discuss the prototype following the full-dbms-split partitioning model. For this model, as much of DuckDB as possible will be place inside the enclave to minimize the number of ECalls and/or OCalls required.

6.3.1 Running DuckDB from an enclave

To run DuckDB entirely inside an SGX enclave, two different approaches are possible. The first approach is to manually rewrite and recompile DuckDB using the SGX SDK. The main difficulty is the limited subset of the libc and libc++ libraries that is available to code inside the enclave. The reason for this is that system calls are not directly callable. To work around this problem, a so-called shim layer needs to be implemented that defines OCalls that allow the unsecure host to provide the necessary system calls. The main advantage of this approach is that it can be tailored exactly to the specific application that is running inside the enclave, keeping the TCB to a mininum. Additionally, as the shim layer is as small as possible, the limited EPC can be optimally used by the application

itself. An example of a system using this approach is CryptSQLITE (19), who define a shim layer with the 29 syscalls that their base DBMS, SQLite, requires.

Another approach is to use an existing shim/LibOS library. Right from the launch of SGX, researchers have acknowledged the problem the need to rewrite and recompile existing applications to run them inside enclaves. To solve this, different systems have been presented to support running unmodified applications in SGX such as Haven (103), Scone (104), Panoply (105), Graphene (10), and Occlum (106). All these systems use similar approach but move variable levels of functionality into the enclave. Now for the EDuckDB prototype, both the manual shim method and building on top of an existing shim/libOSB would work. However, as using an existing shim/libOS does not pose any significant problems our limitations for our research goals, we opt for the more time-efficient approach of leveraging an existing shim/libOS. Choosing which of the aforementioned approaches to pick is straightforward as currently of all these systems only Graphene is both open-source, under active development, well documented, and the closest to being production ready[1]. We note that while for this research, the shim/LibOS approach is taken, but for a production-ready version, the SGX SDK build is most likely preferable for the aforementioned advantages regarding security and performance. As DuckDB is an embeddable DBMS with no external dependencies, the SGX SDK build is clearly conceivable.

6.3.2 Graphene-SGX

We start by giving a brief overview the Graphene architecture. Different LibOSs use different methods. For example, Haven pulls a significant part of the OS functionality into the enclave. On the other end, Scone and Panoply implement small shim layers over an API layer like the syscall table or libc interface. Graphene follows a similar approach to Haven and it places a libc implementation, a library OS, and a shield layer in the enclave along with the application. In Figure 6.3, the layout of an application running in Graphene-SGX is shown. Below the bottom black line the untrusted Linux kernel is shown with the SGX drivers. Above the black line, which depicts the the Linux syscall interface, the platform adaption layer (PAL) of Graphene is shown. The PAL implements functions of the host application binary interface (ABI), against which the library OS is programmed. Above the PAL the enclave components are shown, starting with the shield code, shield data, manifest and file hashes. These components are responsible for verifying the security and integrity of the calls made from the library OS to the PAL. Additionally, these components

[1]Graphene is currently in a prerelease and are expecting to launch a first production version in Q3 2021 (107)

verify the hashes of any files or shared libraries that the enclave code may require. Next is the Library OS, which works by either handling functions that can be done in the enclave itself, or by delegating system calls to the PAL through the shield layer. Finally, on the top of the diagram we see the shared libraries for the application and the application itself.

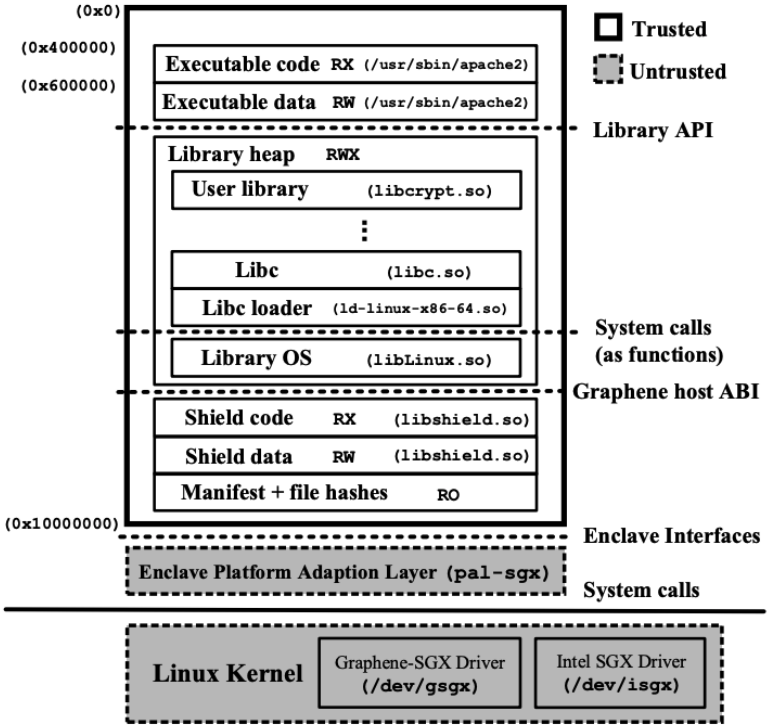


Figure 6.3: Graphene-SGX architecture (10)

6.3.3 DuckDB master branch in Enclave

First, we demonstrate the performance of a naive EDuckDB implementation that runs regular unencrypted DuckDB inside Graphene. DuckDB was run from the master branch with no modifications[1]. For the precise versions of both DuckDB and Graphene that are used, see Appendix A. To run DuckDB in Graphene, a manifest file was created that specified our desired configuration to allow the benchmark runner of DuckDB to run. This manifest is included in Appendix B. In this configuration we specify the enclave size and which files and shared libraries need to be accessed. For the enclave size 16GB was chosen,

[1]Note, a small modification was required to allow the benchmarks to run: the disabling of file locking as this is not supported in the version of Graphene that was used. However, this has no impact on the results.

as this was the largest possible enclave size that would work without crashing. These large enclaves are needed as the same environment was used for all Graphene experiments and enclave size for our version of SGX needs to be defined at enclave initialization. In Figure 6.4, the benchmark results for running TPC-H SF1 on the DuckDB master branch with vector size 1024 in Graphene are shown. As can can be seen in the graph, naively running DuckDB in Graphene leads to a slowdown of 22x for the queries that did not crash or time-out. Since in this benchmark we are storing over 1GB in EPC and most queries will touch more than the EPC limit of memory, these overheads were to be expected and are consistent with the SGX read+write benchmark from Figure 6.1(b).

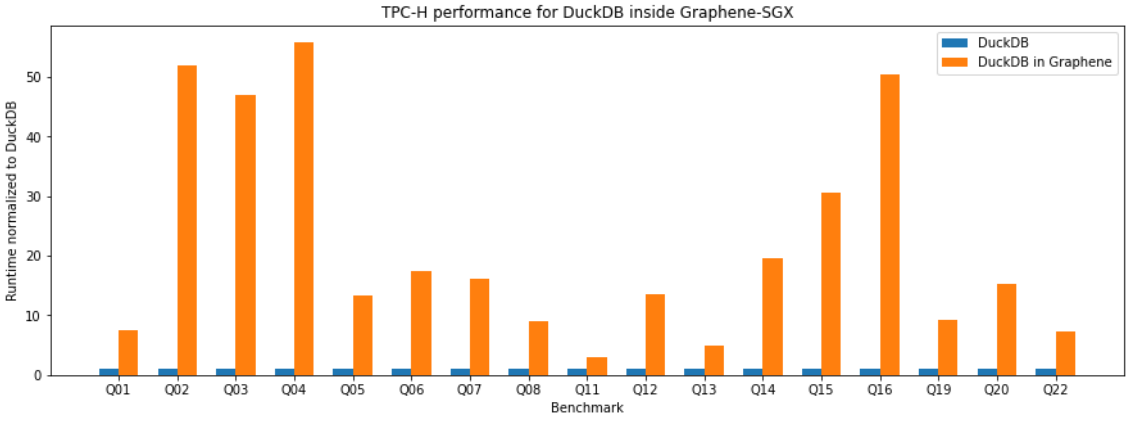


Figure 6.4: Benchmark results for DuckDB running inside Graphene-SGX (09, Q10, Q17, Q18, Q21 omitted due to timeouts and crashes when run inside Graphene)

6.3.4 Graphene-aware DuckDB

The poor results in section 6.3.3 can be attributed mainly to the cost of EPC swapping. We will go into greater detail on this in section 6.5. To mitigate this problem, the EPC usage needs to be drastically reduced, because even with the newest version of SGX with twice the amount of EPC, only a slight improvement is to be expected. To achieve this, we implemented Graphene-aware EDuckDB (GA-EDuckDB). For GA-EDuckDB, the buffers allocated by the DuckDB buffer manager for storing the data are moved to unsecure memory. To prevent loss of confidentiality for these buffers, we use the vector-encrypted implementation from sections 5.2.3. To allow the buffer manager to allocated both secure and unsecure memory, we extended Graphene with two system calls to allocate and free buffers of unsecure memory. The vector-encrypted EDuckDB implementation was made modified to use this system call to allocate memory for table data, while using regular

memory allocation for all other allocations. The resulting GA-EDuckDB implementation fits in the limited EPC with a TPC-H SF1 database loaded in-memory. To evaluate the improvement of GA-EDuckDB over DuckDB, we run the TPC-H benchmarks on the experiment machine defined in Appendix A at scale factor 1 with the default vector size of 1024. In Figure 6.5, the results of this experiment are shown. Note that due to the use of our EDuckDB implementation, only the fixed-length datatype queries defined in Secion 5.3.1 are used for this benchmark. In the results we can see that all queries show significant improvement over running DuckDB in Graphene. Especially queries with no large intermediates such as Q06 benefit massively. Others, such as Q04, which contains a hash join of 144k tuples, remain at well over one order of magnitude overhead. In section 6.5 we will analyze the behaviour of GA-DuckDB in more detail and compare it to the implementation of section 6.4

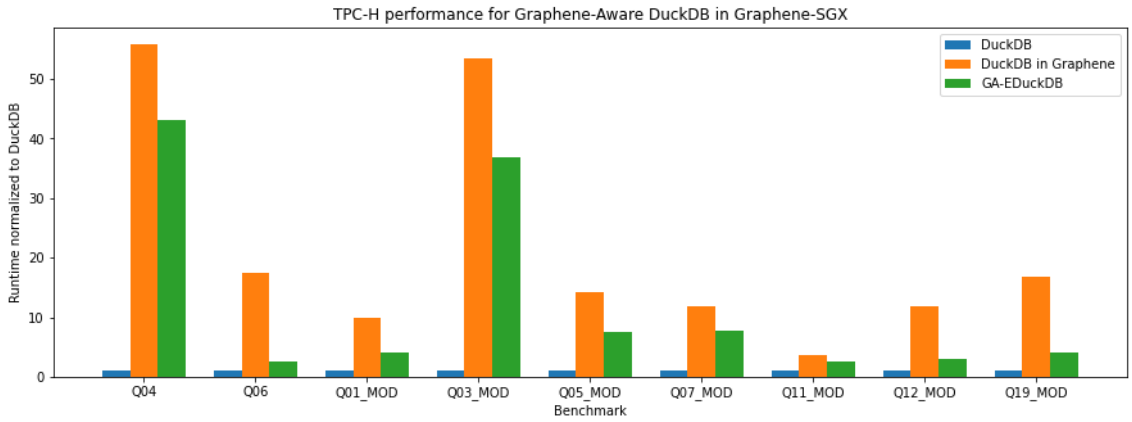


Figure 6.5: Benchmark results for GA-DuckDB in Graphene (Q17 omitted due to timeouts/crashing)

6.4 DuckDB with operators in enclave

In this section, we discuss the prototype following the other partitioning model, the minimal-dbms-split. For this model, the minimal amount of code is placed inside the enclave to minimize the execution time spent in enclave mode and minimize the EPC usage. The goal is to reduce the LLC miss overhead and the EPC swapping overhead at the cost of more ECalls and more information leakage. Using this model will results in more fine-grained access pattern leakage and also some direct leakage comparable to an ideal CRYPTO-EDBMS or a system such as StealthDB.

Previous work using the minimal-dbms-split model for an SGX based EDBMS used per-value encryption combined with logical operations and comparators as ECalls, essentialy emulating PHE and PPE schemes using TH. However, for an analytical workload this approach is impractical. Firstly, as discussed in section 5.1, per-value encryption imposes large overheads making encryption very inefficient. Secondly, performing an ECall for every value is extremely expensive in OLAP workloads. Consider TPC-H Q06 at SF1. In this query 114160 tuples are produced by the FilterScan operation. For this query, the overhead of ECalls of only the aggregation operator would already be over 764%[1]. Finally, this approach inhibits the efficient use of the vectorized query execution model. One of the goals of vectorized query execution is to defined operators in terms of simple loops containing easily optimizable code that modern CPUs can efficiently executed using SIMD instructions. Replacing the simple operations in the body of these loops with ECalls makes CPU-efficient query processing very hard.

The solution is to use vectorized encryption and move the vectorized database operators inside the enclave. These enclave operators take one or more encrypted vectors as parameter, decrypt the whole vector inside the enclave, use the same code to perform the operation and if necessary, re-encrypt the resulting vector. Using this approach, the ECall cost is amortized, the encryption throughput is efficient because it operates on buffers instead of values, and the same CPU-optimized code can be used as in the regular vectorized operators. To implement the vectorized SGX operator EDuckDB (VSO-EDuckDB), the vector-encrypted implementation from section 5.2.3 was used as a starting point, but instead of decrypting the vectors immediately on scanning, the scan operator produces vectors containing encrypted data. An example VSO-EDuckDB query plan is shown in Figure 6.6. A consequence of the decision to re-implement query operators in SGX is that implementing VSO-EDuckDB is significantly more work than, for example, StealthDB. Due to time constraints, the implementation of VSO-EDuckDB was limited to the operators required to execute TPC-H Q06 and its emulated compression variant. This query was chosen for the limited amount of operators required to implement the query, while still being a good representative of a typical analytical query. The operators implemented are listed in table 6.4 A total 27 ECalls were implemented to support the required operations. The reason for different operators to have different numbers of ECalls implemented is due to the fact that ECalls can not be defined as templates. Also some operators such as Select, required implementations of the operator with different parameters. Finally, the Intel SGX SDK contains an optimization option called *switchless mode*. This optimization

[1]This value is calculated by using the ECall cost from section 6.1.3

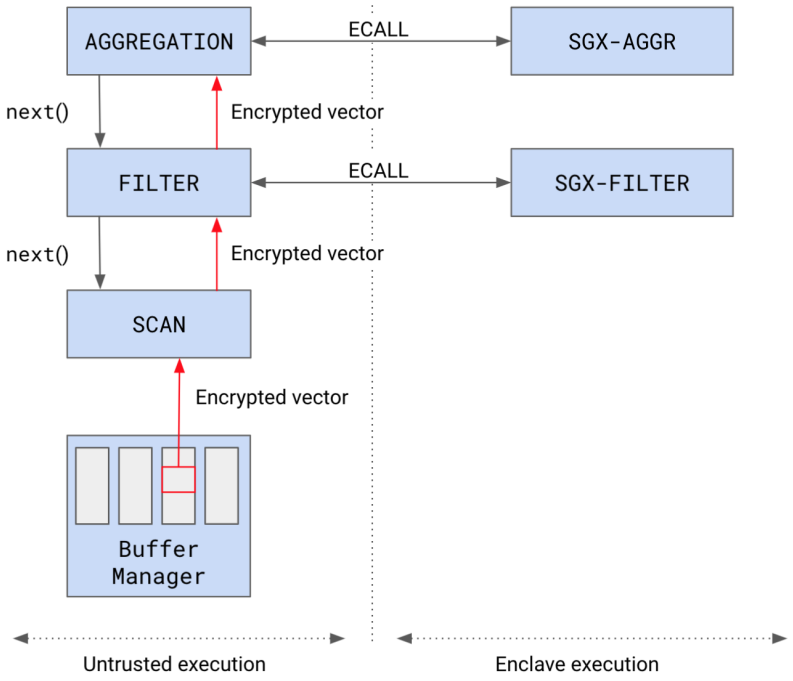


Figure 6.6: VSO-EDuckDB example query plan for TPC-H Q06

Operation	ECalls
Binary multiplication	2
Unary aggregate	5
Cast	3
Zone-map helper functions	7
Select	10

Table 6.4: Vectorized SGX Operators implemented in VSO-EDuckDB to support Q06 and compressed Q06

is described in the paper by Tian et. al. (67). The optimization works by running two threads: one is running inside the enclave, the other is running outside the enclave. With this multithreaded setup, the ECalls and OCalls are replaced by asynchronous requests placed in a shared buffer. This optimization essentially trades CPU cores for more efficient ECalls/OCalls. With the switchless optimization enable ECalls take 1.5k cycles and OCalls take 1k cycles which is almost an order of magnitude shorter. For our detailed analysis we include the results with and without this optimization enabled, where the optmization-enabled version is called VSO-EDuckDB-S.

6.5 Evaluation

6.5.1 Performance comparison

To compare the performance between the VSO-EDuckDB and GA-EDuckDB implementations, TPC-H Q06 was ran at scale factor 1, as Q06 is the only query that both implementations can run. The same machine from appendix A is used, but this time a vector size of 8192 is chosen, which we will explain in section 6.5.2. The results are shown in Figure 6.7. In the graph, we see that both GA-EDuckDB and VSO-EDuckDB add a significant amount of overhead, respectively 153% and 200%. Interestingly, the amount of overhead for the two implementations is comparable even though their execution models differ significantly. When enabling the switchless optimization for VSO-EDuckDB, we can see overhead drop significantly, coming close to the uncompressed, non-SGX encrypted, vector implementation which is at 75% overhead.

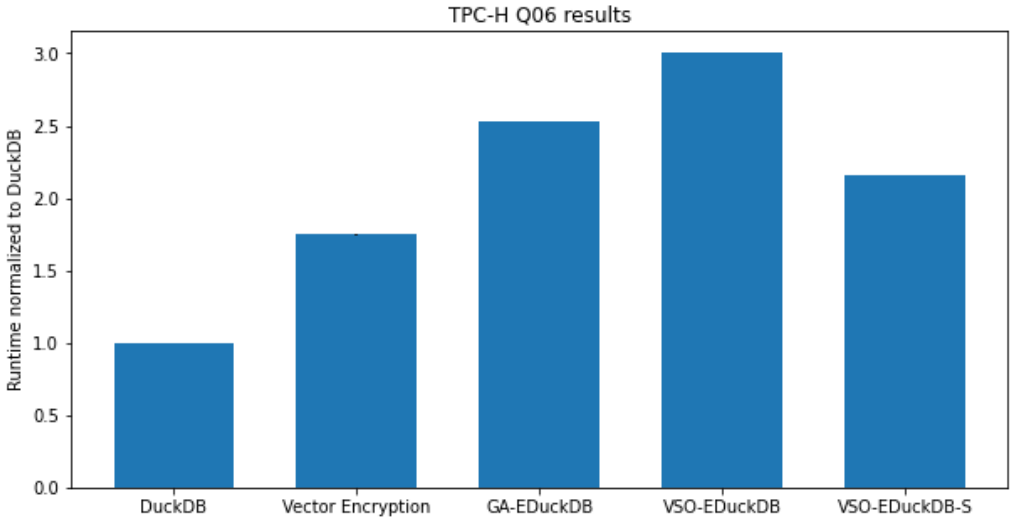


Figure 6.7: Performance evaluation of all EDuckDB implementations on TPC-H Q06

6.5.2 Impact of vector size

In vectorized query execution, the vector size is a parameter can be used to optimize performance. In an unencrypted DBMS using vectorized execution, the optimal vector size depends on several factors, such as the workload, the hardware used, and characteristics of the database such as compression ratios. For DuckDB, the vector size is hard-coded and is set to 1024 by default. At this vector size, most vectors will fit in the L1 cache to ensure low L1 cache misses for fast vector operations. In Figure 6.8, a plot of TPC-H is shown

for different vector sizes of DuckDB. In the graph it can be seen that the optimal vector

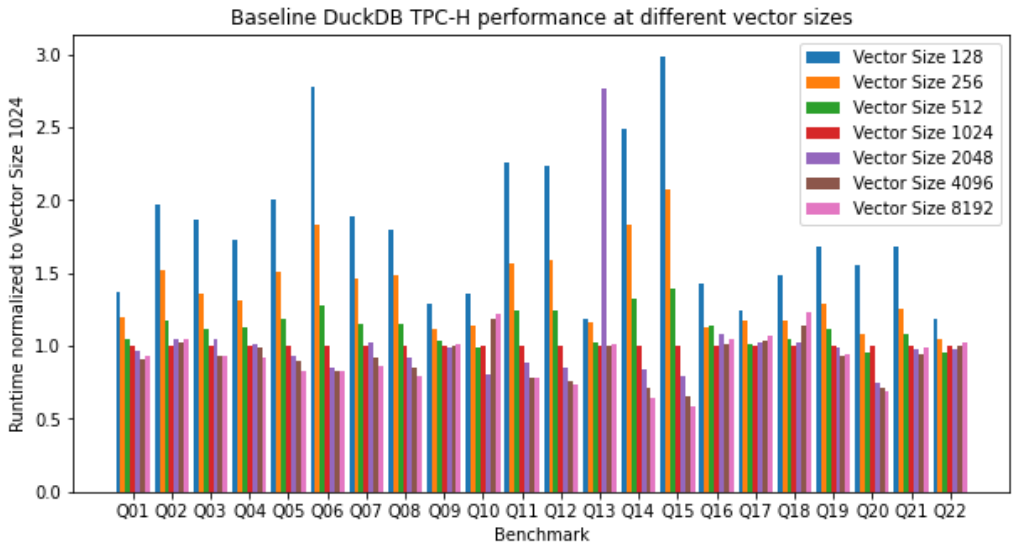


Figure 6.8: Impact of vector size on baseline DuckDB TPC-H performance

size differs per query, but generally around 1k is a safe choice close to the optimum. For EDuckDB however, the optimal vector size changes radically. In Figure 6.9, the performance for EDuckDB running TPC-H Q06 is shown for different vector sizes. The results

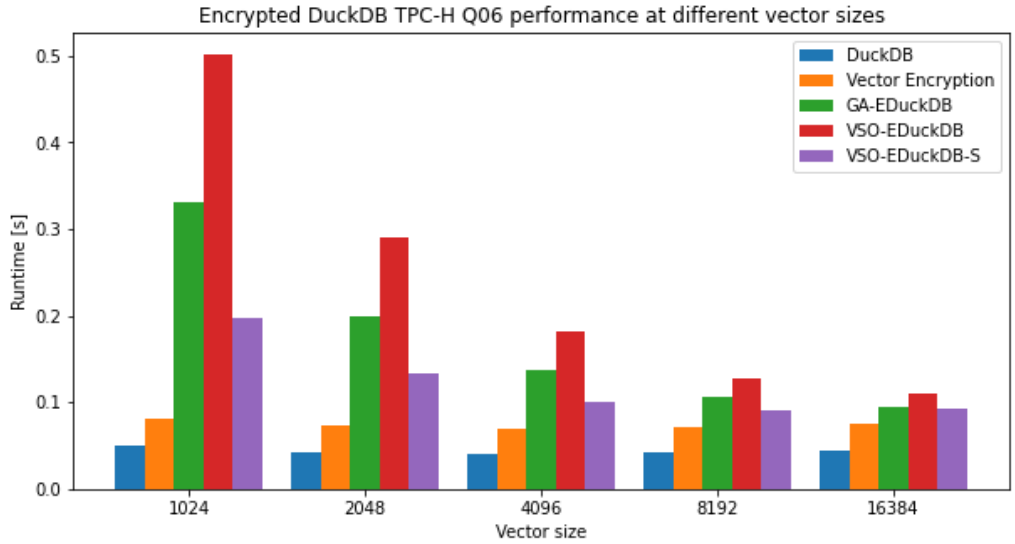


Figure 6.9: Impact of vector sizes on encrypted implementations

show a large impact of vector size on performance for all SGX implementations, while the Vector Encryption implementation and DuckDB vary only slightly. Note that the optimal

vector size has shifted upwards significantly compared to unencrypted DuckDB. The reason for this shifted optimal vector size can be explained by the performance bottlenecks for SGX we have explained in section 6.1.3. Firstly, for VSO-EDuckDB the total amount of ECalls depends directly on the vector size and increasing the vector size reduces the amount of required ECalls by the same factor. This also explains why VSO-EDuckDB-S, where the ECalls are significantly less expensive, has much less performance gain when increasing vector size. Interestingly, GA-EDuckDB also benefits from a large vector size significantly. While we were unable to conclusively establish the cause of this effect, we will briefly explore the possible causes. Looking at the main SGX performance bottlenecks from section 6.1.3, we can exclude the ECall cost as a source of overhead as no ECalls are made during the benchmark execution. Secondly, we can exclude EPC paging, as no EPC paging was detected during the benchmark execution. This leaves only the increased LLC miss cost as a source of significant overhead. With LLC misses being much more expensive in secure memory, as shown in Table 6.1, a possible explanation would be that the increased vector size reduces the LLC misses to secure memory. However, to measure and confirm this theory, a detailed performance analysis should be performed, which was left to future work due to the complexity of profiling SGX code.

6.5.3 Impact of compression

To determine the impact of compression on the SGX-based implementations, the emulated, compressed version of Q06 was used that is described in section 5.3.1. In Figure 6.10, the results are shown for the uncompressed and compressed Q06 query for all implementations at different vector sizes. For GA-EDuckDB the effect of compression is similar to the effect we have seen before on the vector-encrypted implementation in an absolute sense. The vector size has no significant impact on this effect. For VSO-EDuckDB, the emulated compression results in a higher number of ECalls, which results in a larger total overhead. As the vector size increases, reducing the number of required ECalls, this effect diminishes and at 16k vector size, compression results in a small performance improvement. When enabling the switchless optimization, ECalls are significantly cheaper and therefore the effect is less pronounced. The best performing SGX based setup in this experiment is compressed GA-DuckDB with a vector size of 16k, which has a performance overhead of 66%, demonstrating that a well chosen vector size together with compression can reduce decryption and SGX overheads to reasonable amounts.

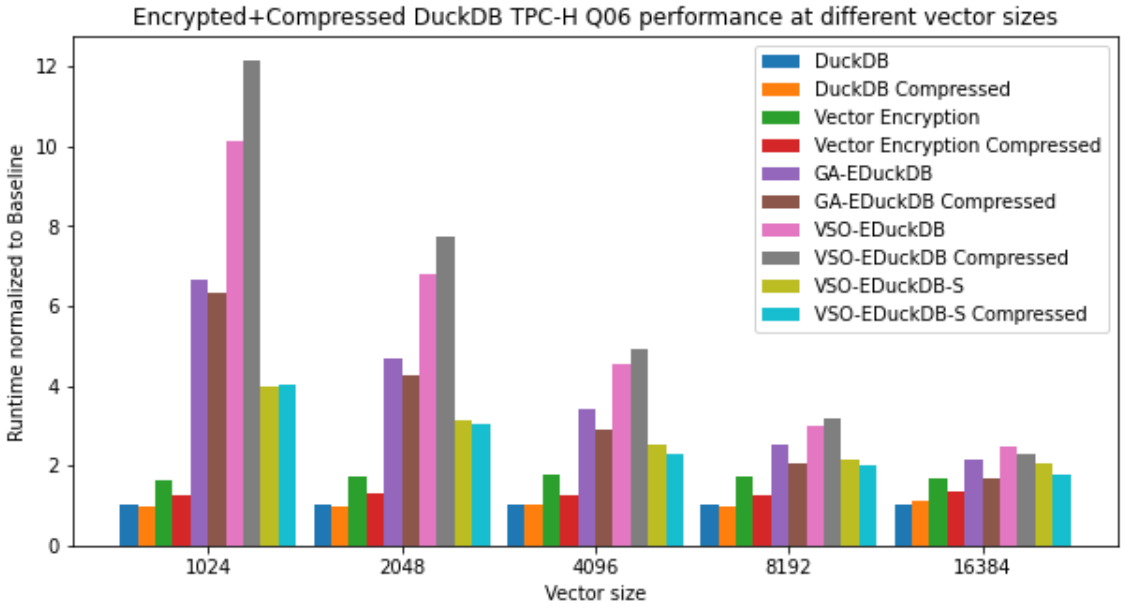


Figure 6.10: The impact of vector size on TPC-H Q06 compressed and uncompressed

6.5.4 Scaling

Thusfar, a TPC-H scale factor of 1 has been used, which uses a database of around 1GB. In the real world, databases can be much larger, up to several orders of magnitude. To see how the different implementations scale compared to unencrypted DuckDB, TPC-H Q06 was run at different scale factors. In our experiment we were limited by the GA-EDuckDB implementation which would crash at scale factors over 8 on our machine with 32GB RAM. The vector size is set to 8192. The results are shown in Figure 6.11. Note that performance deteriorates up to SF2 for all implementations, likely due to the data fitting less and less in the system caches. Above scale factor 2, the overhead compared to DuckDB remains close to linear for all implementations. While the results for Q06 in the previous section show good scaling properties, this is mainly because Q06 does not require a lot of memory for intermediate results. As seen before in Figure 6.5, some queries perform reasonably well while others result in over 30x slowdown. This large difference in queries can be explained by the overflow of the EPC. To demonstrate this, a benchmark based on a simple join query on a TPC-H SF2 database was run. The query is shown in Listing 6.4.

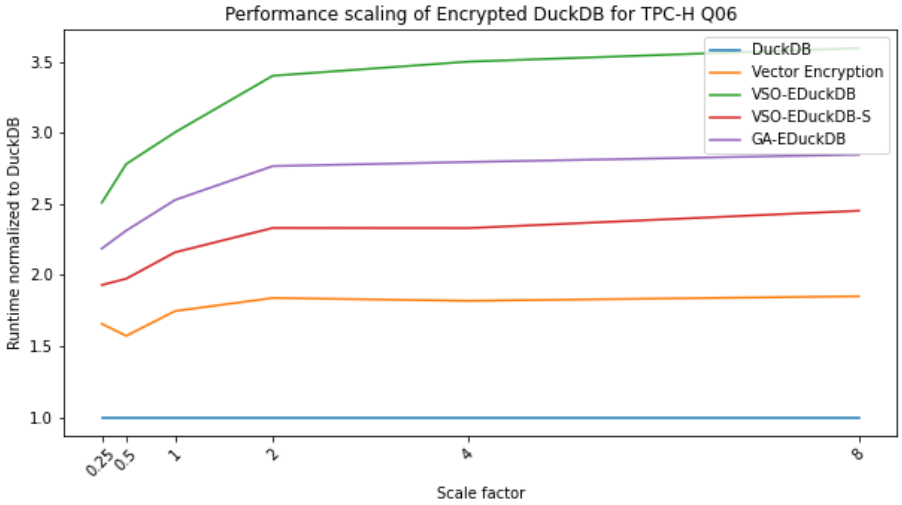
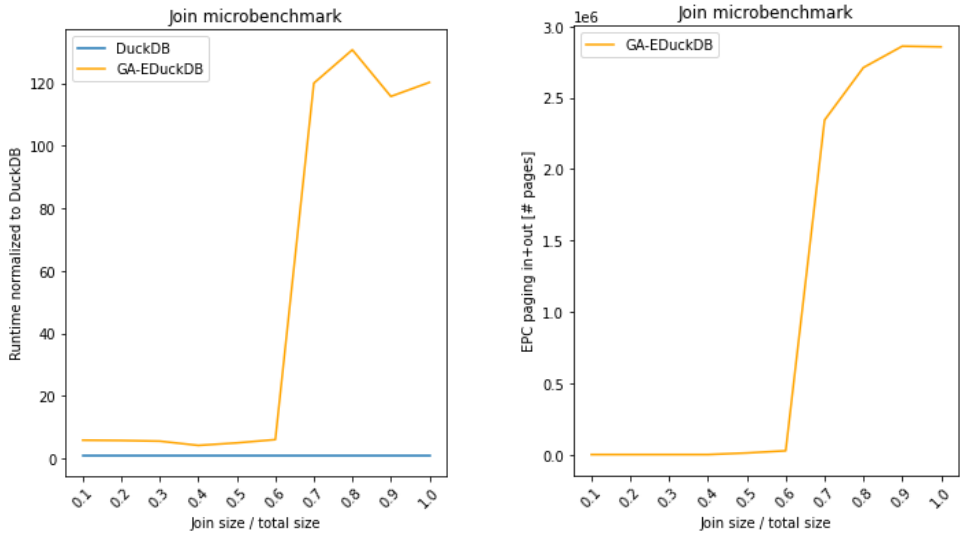


Figure 6.11: Scaling characteristics of Encrypted DuckDB implementations

```
SELECT  avg(o1.o_orderkey)
FROM orders o1, (
    SELECT o_orderkey
    FROM orders
    LIMIT joinsize) as o2
WHERE o1.o_orderkey = o2.o_orderkey;
```

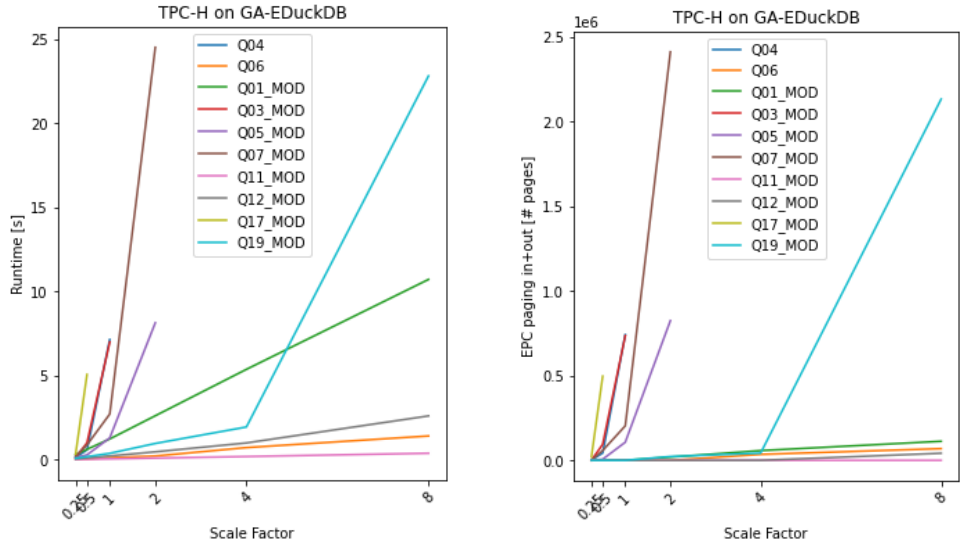
Listing 6.4: Join microbenchmark

In this query the join size was set beteen 0 and 1 times the size of the orders table to gradually increase the join table size between 0 and 1.5 million tuples. Then, to measure EPC paging, a monitoring tool sgxstat (108) was used to measure EPC paging. The results for this experiment are shown in Figure 6.12(a) and 6.12(b). The figures show that without EPC paging, relative runtime is 5.33x on average. However when the EPC is full and EPC paging starts around a join table size of 1.05 million tuples, the slowdown increases to over 2 orders of magnitude. For GA-EDuckDB, the same experiment is run for the numerical TPC-H queries. The results are shown in Figure 6.13(a) and 6.13(b). In this graph a similar result is visible, with queries scaling linearly until EPC paging begins, after which the overhead increases massively, with some queries reaching the 30s timeout limit of the benchmark runner at only scale factor 1. At scale factor 8, only 4/10 queries still scale linearly and 5/10 queries reach the timeout of 30s. Finally, in Figure 6.14, a graph is shown of the 4 TPC-H queries that GA-EDuckDB supports without overflowing the EPC at scale factor 8 . These results demonstrate the overheads that are to be expected for GA-EDuckDB if more EPC is available in newer versions of SGX.



(a) Runtime (b) EPC Paging

Figure 6.12: Impact of EPC paging on join microbenchmark performance



(a) Runtime (b) EPC Paging

Figure 6.13: Impact of EPC Paging on TPC-H performance

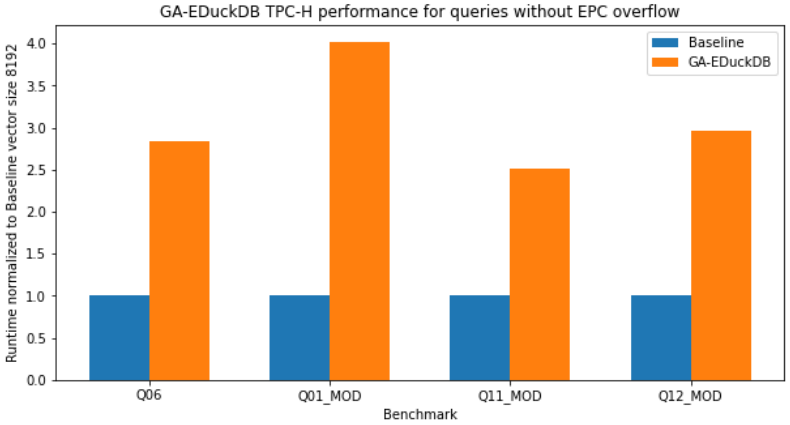


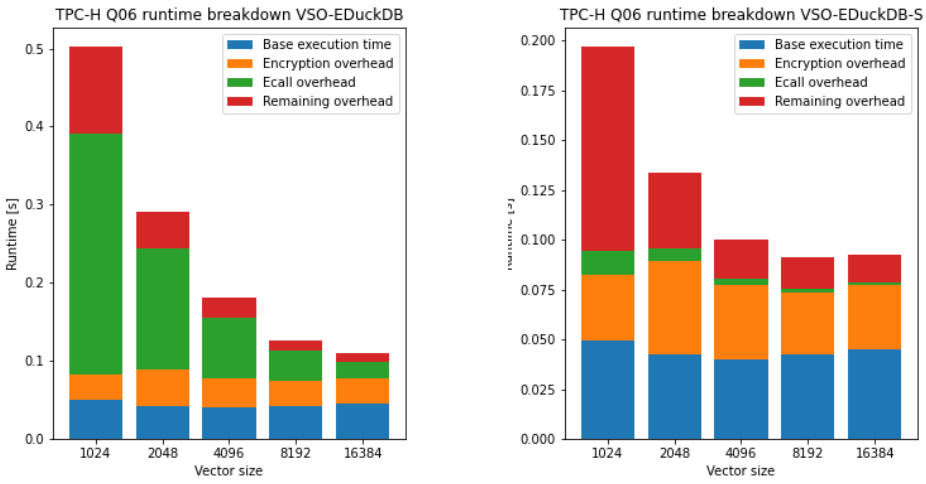
Figure 6.14: GA-EDuckDB performance for queries with no EPC overflow at larger scale factors

6.5.5 VSO-EDuckDB overhead breakdown

To better understand what causes the additional performance overheads from VSO-EDuckDB, the vector size experiment from section 6.5.2 was run with instrumentation code to count the number of ECalls made. With the total ECall counts and the cost of an ECall determined by Tian et. al. (67), an estimated breakdown was made for the VSO-EDuckDB and VSO-EDuckDB-S implementations using the results for TPC-H Q06 from section 6.5.1. The results are shown in Figure 6.15(a) and 6.15(b). In the results the impact of ECalls of VSO-EDuckDB performance is shown clearly. For lower vector sizes of VSO-EduckDB, over half the execution time is spent performing ECalls. Even at a vector size of 16K, VSO-EDuckDB still has a significant overhead from ECall cost, suggesting higher vector sizes may be worth considering. For VSO-EDuckDB-S, where the switchless optimization is enabled, we can clearly see the 10x reduction of ECall cost. When sufficiently large vector sizes are used, VSO-EDuckDB-S adds only a small overhead on top of the baseline encryption.

6.5.6 Performance comparison to existing literature

Comparing EDuckDB to existing EDBMS literature directly is difficult, due to it being the first EDBMS of this type. We will briefly go over the most relevant related work to see globally how EDuckDB compares. A PHE+DET based EDBMS that focusses on OLAP workloads is Symmetria. As discussed in section 3.2.1.5, the authors evaluate their EDBMS using TPC-H SF100 running on Apache Spark across 10 machines. In their results, which can be seen in Figure 3.5, the average runtime is 5.25x baseline Apache



(a) VSO-EDuckDB

(b) VSO-EDuckDB-S

Figure 6.15: Detailed TPC-H Q06 breakdown of performance overhead for VSO-EDuckDB and VSO-EDuckDB-S

Spark, but with several massive outliers, most notable Q06 at >35x and Q01 at >15x. When comparing these results to those of GA-EDuckDB, we find a similar pattern. GA-EDuckDB has large outliers when EPC is exceeded, but run times of <4x otherwise. Interestingly, the queries that perform the worst in Symmetria, Q01 and Q06 are among the queries that GA-EDuckDB can run reasonably efficiently at 3x and 4x. On these results an important side-note should be made: the experiments are run on very different systems at very different database sizes. The comparison of these results is therefore of limited value. In the TH-EDBMS literature, options for direct comparison are limited. Many systems either focus diffrent and incomparable workloads (8)(7), use incomparable security requirements (72)(21), and/or do not experimentally evaluate their system in a real-world manner (6)(7). One system that does use the full-dbms-split model in SGX and focusses on OLAP workloads in their evaluation is CryptSQLite. The authors evaluate their system using TPC-H SF1. In their results, no precise overheads are specified. From their graphs, overheads of around one to two orders of magnitude can be seen. Combined with the fact that baseline SQLite TPC-H performance is multiple orders of magnitude worse than that of baseline DuckDB on our experiment machine, we find that for Q06 both VSO-EDuckDB and GA-EDuckDB outperform CryptSQLITE by roughly 4 orders of magnitude.

6.5.7 Security analysis

To conclude our analysis we analyse the resulting security properties of the two designs. First we verify that the intended minimum of leakage requirements are met, then we discuss what would be required to improve security.

Leakage Starting with information leakage, VSO-EDuckDB achieves class 1 leakage, similar to StealthDB or Arx. This means that an attacker will learn detailed information on what types of queries are run, what query plan is chosen by the optimizer, and which tables are touched. Furthermore, attackers that are persistent, will also be able to see which tuples are included in a query result or intermediate result. The reason for this information leaking is that the vectorized operators in DuckDB use selection vectors to mark which tuples in a vector are included in a result. As these selection vectors are not encrypted in VSO-EDuckDB, an attacker can learn for example which tuples are returned from a filter scan operation. A possible improvement that could be made to VSO-EDuckDB in an attempt to improve the leakage profile is the encryption of these selection vectors. However, whether doing so will actually increase security by a considerable margin is questionable as the operators currently also leak the amount of matched tuples in a vectorized filter operation directly. GA-EDuckDB in its current implementation, has better leakage patterns than VSO-EDuckDB. As all execution happens from within the enclave, attackers will not directly be able to see what queries are run, what query plan is chosen, or which tables or columns are involved. Indirectly however, the attacker can see which parts the columns are scanned for each query, as these buffers are stored in unsecure memory. GA-EduckDB performs similar to CryptSQLITE in this respect, with the main difference that CryptSQLite leaks which pages are accessed and GA-EDuckDB leaks which vectors are accessed. For both VSO-EDuckDB and GA-EDuckDB, improving their leakage to oblivious query execution where only course-grained access patterns are leaked are possible for both implementations. To achieve this, oblivious query operators will need to be implemented as is demonstrated by Cipherbase, Opaque, and ObliDB.

Threat model Both GA-EDuckDB and VSO-EDuckDB currently only provide protection against passive attackers. For a class 1 leakage database like VSO-EDuckDB, active attacks are not included in the threat model due to recent attacks such as that by Grubss et. al. (69). For GA-EDuckDB, enabling authenticated encryption would provide security against some active attacks, but fully securing against an active threat would require

preventing the attacker from being able to swap two ciphertexts. Even though it would require some significant code rewriting, adding this protection to GA-EDuckDB is relatively straightforward. Each encrypted vector would need to be encrypted with authenticated encryption combined with a special unique identifier pinning a vector to a specific set of rows, column and table. This unique identifier would need to be verified by each operator that needs to decrypt the vector. As shown in Figure 5.6, authenticated encryption adds a reasonable amount of overhead to the total runtime especially relative to the total GA-DuckDB runtime. And as checking one identifier per vectorized operation would likely not incur a large overhead, it is expected that protection against active attackers could be added to GA-EDuckDB at a reasonable performance overhead.

6.5.8 Conclusion

From the evaluation of the two designs VSO-EDuckDB and GA-EDuckDB, we can draw the following conclusions. Firstly, both designs can result in reasonable overheads for TPC-H Q06 when using an appropriate vector size and an efficient compression schemes, as shown in Figure 6.10. Furthermore, the difference between running DuckDB inside an enclave entirely and running as little code as possible inside the enclave are surprisingly small. However, GA-EDuckDB was shown to suffer significantly from the limitation of 128MB EPC. Even at scale factor 1 of TPC-H with a database size of around 1GB, EPC paging causes large overheads for many queries. At scale factor 8 only 4/10 TPC-H queries tested were able to run without overflowing the EPC. While VSO-EDuckDB was not evaluated on its scaling properties, we predict that while its performance will be better than GA-EDuckDB due to it requiring less secure memory, it will run into the same EPC limitations as GA-EDuckDB for many TPC-H queries at similar scale factors.

7

Conclusion

In this thesis we have explored building an OLAP oriented EDBMS using DuckDB and SGX. We have done this by first exploring the use cases and corresponding threat models of EDuckDB. Secondly, different encryption strategies were analyzed to optimize decryption efficiency. Finally, the integration with SGX was designed, implemented and evaluated.

The main contributions of this work are the insights gained on efficient, SGX-based, OLAP EDBMS design. We have demonstrated encryption strategies that allow efficient querying of encrypted data. Additionally, we have presented two viable approaches for designing an OLAP EDBMS using the current generation of Intel SGX. With the implemented prototypes following these approaches, we have shown that reasonable performance can be expected when query operations using little memory are used. For query operations requiring larger amounts of memory for storing intermediates, large overheads are currently to be expected. With these findings we can draw the following conclusion: To support efficient OLAP workloads in an SGX-based EDBMS, one of two things are required. Either the EPC needs to be increased significantly to allow secure and efficient storing of intermediate query results, or new query operation and optimization techniques need to be developed to efficiently operate within the limited amount of secure memory currently offered by SGX. For example, one of the more memory intensive operators, hash aggregation, could be made efficient by using a partitioned hash join to reduce the hash table size. Together with the partitioned hash join optimized for the EPC size, Query Optimizer rules would be designed to use the special join operator.

Another contribution of this work is the insight that the modern OLAP query processing techniques compressed execution and vectorized execution, can significantly improve the performance of an SGX-based EDBMS. Using compressed execution will reduce the overall amount of data needing encryption/decryption as well as reducing the EPC usage, while

vectorized execution allows intuitive processing of values grouped into encrypted buffers and also helps reduce overheads in both GA-EDuckDB and VSO-EDuckDB.

7.1 Research questions

1. What are the use cases for encryption enabled DuckDB and what are the corresponding trust and threat models? For this thesis, three use cases of Encrypted DuckDB were analyzed: an analyst running a local instance of DuckDB, an IoT edge node, and a cloud database. For this first use case, it was found that no significant improvement to security was possible over existing common encryption techniques such as FDE. For the latter two cases, it was found that a DuckDB-based EDBMS can provide security against threats that are currently difficult to protect against. For the detailed analysis we refer to section 4.2.

2. How to implement encryption in DuckDB at a negligible performance overhead? For OLAP workloads, scanning and processing large amounts of tuples is very common. For this reason, encryption needs to be implemented carefully as doing so naively will result in large overheads. Even with a well-chosen encryption granularity and industry standard encryption algorithms TPC-H overheads can surpass 300% as shown in Figure 5.6. However, when fast encryption schemes are chosen such as hardware-accelerated AES-CTR, average performance overhead for a subset of TPC-H SF1 was found to be 22% as shown in Figure 5.6. By using compressed execution, these overheads can be further reduced by up to 2.12× for AES-CTR, as discussed in section 5.3.2.3.

(a) What is the optimal granularity to encrypt the data? As discussed in section 5.1, choosing the right encryption granularity is crucial for OLAP workloads. In our analysis, we conclude that for our requirements, per-value encryption is not viable due to high initialization overhead of encryption algorithms resulting in poor encryption/decryption performance when using small buffers. The more suitable approach for OLAP workloads is to group values into buffers and encrypting the buffers as a whole. For EDuckDB, two viable approaches for integrating this into the execution model are shown: Block Encryption and Vector Encryption, discussed in sections 5.2.2 and 5.2.3. The performance of these two approaches is comparable as is shown in Figure 5.5. Besides performance, choosing the encryption granularity also impacts which component performs the encryption/decryption at which point in the query processing. For SGX-based EDuckDB, Vector Encryption

was chosen as it was suitable for both GA-EDuckDB and VSO-EDuckDB as described in sections 6.3.4 and 6.4.

(b) What encryption scheme is most suitable? As discussed in section 4.3, the class of cryptographic schemes used in cryptography based EDBMSs such as PHE and PPE schemes, generally do not perform well for analytical workloads. This is one of the reasons why for EDuckDB, symmetric encryption schemes are chosen in combination with the TH-EDBMS model to allow computation over the ciphertexts. Two main types of encryption schemes were found to be suitable for EDuckDB, depending on the required threat model. For the passive attack model, the stream cipher AES-CTR and the family of stream ciphers Salsa/ChaCha were found to be most suitable due to having high performance and being well researched by the scientific community. On the moderately new hardware used in our experiments, AES-CTR performed best with decryption costs of under 1 cycle per byte. On more recent Intel and AMD hardware, both AES-CTR and ChaCha decryption costs are roughly halved and get as low as 0.4 cycle per byte. When the requirements assume an active attacker, an authenticated encryption algorithm is necessary. AES GCM was found to be suitable as its performance is only marginally worse than AES-CTR.

3. What functionality can we support when only a negligible performance overhead is allowed? As discussed in the answer of research question 2b and section 4.3, CRYPTO-EDBMS suffer from significant overheads for a most OLAP functionality. The functionality that can be supported efficiently, such as equality and range comparison through PPE, comes with prohibitive leakage properties. For TH-EDBMS generally lower overheads and/or better leakage characteristics can be accomplished when using a modern TH such as SGX. However, with SGX, even for queries that require little memory, overheads of 150% to 300% were seen in our results as is shown in Figure 6.14. Since operations involving large intermediates are very common in OLAP workloads, we can conclude that most common DBMS functionality can currently not be supported at negligible performance overhead when assuming our security requirements.

4. How to integrate a trusted hardware solution into our encrypted DuckDB implementation

(a) Which solution is most suitable? In current EDBMS literature, Intel SGX is by far the most common TH technology used. For reasons discussed in section 6.1.1, SGX was the most suitable TH solution for our research. However, SGX is far from perfect as is proven by its long track record of security issues and its EPC limitation. On paper, one of the competitor technologies, AMD SEV-SNP has the potential to outperform SGX significantly for some use cases. In practice however, AMD is suffering from several critical security issue as well that have yet to be solved.

(b) How to integrate a trusted hardware solution into encryption enabled DuckDB to improve privacy at a negligible performance overhead? As stated in the answer to research question 3, currently no available TH or TEE is able to provide security through encryption for our defined use cases at a negligible overhead. However, in our research we have demonstrated two viable approaches that have reasonable performance overheads which are explained in sections 6.4 and 6.3.4. The implementations of these approaches have TPC-H run times range from 2.5x to >40x. When using compressed executions with a compression ratio of 3x to reduce the amount of data to be decrypted, overheads as low as 66% were shown. The main bottleneck in these implementations is the availability of EPC in SGX. For a performant, scalable, OLAP SGX-based EDBMSs, either EPC needs to be increased, or efficient query operators and optimization techniques are required to work around the limited EPC.

7.2 Future work

As the research in this thesis has been relatively broad and several parts of a fully functional EDBMS have been considered out of scope due to time constraints, this work has multiple obvious directions for future work.

Implementing with newest SGX versions One of the most interesting research directions is to repeat the experiments with newer versions of SGX. The version of SGX used for this research has only 128MB of EPC available, but the 10th generation of Intel CPUs has double that amount. Even more recently and concurrently to this research, Intel has launched their newest line of server CPUs offering up to 1TB of EPC. With these amounts of secure memory, different architectures can be considered and reasonable performance may even be achieved by running unmodified DBMS in SGX through Graphene.

Using DuckDB compression With this research having shown the effectiveness of compression in reducing overheads through emulated compressed execution. An interesting research direction is to rerun the experiments when DuckDB has implemented compressed execution as is expected later in 2021. Especially combined with the latest SGX version supporting up to 1TB of memory, a DuckDB-based EDBMS seems to be a viable candidate for a production-ready OLAP EDBMS.

Missing DBMS functionality For this research, important DBMS functionality was left out of scope, such as string segments, updates and deletes. This functionality is evidently crucial to building a fully functional EDBMS and would therefore be an interesting subject for follow-up research. For query processing over encrypted strings, integration with techniques such as EncDBDB (9) could be considered. For updates/deletes, relevant techniques have been developed for ObliDB (72).

Oblivious query processing The EDBMS designs in this paper assume access pattern leakage to be out of scope due to their complexity and high overhead. However, oblivious query processing techniques are crucial to provide security when very strict leakage patterns are required. In recent EDBMS literature (6)(21)(72), several techniques and operators have been developed to hide access patterns from query processing to various degrees. These techniques are applicable to both GA-EDuckDB and VSO-EDuckDB and are therefore interesting direction for future research.

Other TEE technology AMD SEV-SNP has the potential to offer significant performance improvements over SGX. When the existing problems with it will most likely be an interesting to analyse its usability in EDBMS design. While currently, x86 is still the leading architecture in cloud infrastructure, ARM may become more and more common as is proven by Amazon working on their own line of ARM processors for their data centers that promise significantly better cost efficiency. With this shift, other TEE technology such as ARM TrustZone or the newly released ARM CCA become more relevant to EDBMS design. When more information becomes available on ARM CCA, analyzing its applicability to EDBMS design will be a promising research direction.

References

[1] **Block cipher mode of operation**. `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`, 2021. vii, 12

[2] Eyad Saleh, Ahmad Alsa'deh, Ahmad Kayed, and Christoph Meinel. **Processing over encrypted data: between theory and practice**. *ACM SIGMOD Record*, **45**(3):5–16, 2016. vii, 17

[3] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. **CryptDB: protecting confidentiality with encrypted query processing**. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011. vii, ix, 2, 15, 19, 20

[4] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. **Processing analytical queries over encrypted data**. In *Proceedings of the VLDB Endowment*, **6**, pages 289–300. VLDB Endowment, 2013. vii, 2, 23, 69

[5] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. **Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets**. *Proceedings of the VLDB Endowment*, **13**(10):1290–1303, 2020. vii, 27, 28, 41, 47, 48, 50, 69

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. **Orthogonal Security with Cipherbase.** In *CIDR*, 2013. vii, 1, 2, 30, 32, 103, 111

[7] Christian Priebe, Kapil Vaswani, and Manuel Costa. **Enclavedb: A secure database using sgx**. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018. vii, 1, 33, 34, 48, 88, 103

[8] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. **Stealthdb: a scalable encrypted database with full sql query support**. *arXiv preprint arXiv:1711.02279*, 2017. vii, viii, 1, 34, 35, 48, 50, 83, 84, 86, 87, 103

[9] Benny Fuhry, Florian Kerschbaum, et al. **Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves**. *arXiv preprint arXiv:2002.05097*, 2020. vii, 36, 37, 88, 111

[10] Chia-Che Tsai, Donald E Porter, and Mona Vij. **Graphene-SGX: A Practical Library {OS} for Unmodified Applications on {SGX}**. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017. viii, 86, 90, 91

[11] Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. **Guidelines for Using the CryptDB System Securely.** *IACR Cryptol. ePrint Arch.*, **2015**:979, 2015. ix, 20, 22

[12] Daniel J. Bernstein and Tanja Lange. **eBACS: ECRYPT Benchmarking of Cryptographic Systems.** (accessed 31 May 2021). ix, 60

[13] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. **Eleos: ExitLess OS services for SGX enclaves**. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017. ix, 82, 83

[14] **Cloud Database and DBaaS - Global Market Trajectory & Analytics**. `https://www.researchandmarkets.com/reports/4804281`, 2021. 1

[15] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. **Executing SQL over encrypted data in the database-service-provider model**. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002. 1, 15, 16, 22

[16] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. **Azure SQL Database Always Encrypted**. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1511–1525, 2020. 1

[17] Peter A Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *Cidr*, **5**, pages 225–237, 2005. 2, 8

[18] Alfons Kemper and Thomas Neumann. **HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots**. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011. 2

[19] Yongzhi Wang, Lingtong Liu, Cuicui Su, Jiawen Ma, Lei Wang, Yibo Yang, Yulong Shen, Guangxia Li, Tao Zhang, and Xuewen Dong. **CryptSQLite: Protecting data confidentiality of SQLite with Intel SGX**. In *2017 International Conference on Networking and Network Applications (NaNA)*, pages 303–308. IEEE, 2017. 2, 65, 69, 90

[20] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. **Big data analytics over encrypted datasets with seabed**. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 587–602, 2016. 2, 25, 27, 41, 47

[21] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. **Opaque: An oblivious and encrypted distributed analytics platform**. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 283–298, 2017. 2, 48, 103, 111

[22] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. **Database architecture optimized for the new bottleneck: Memory access**. In *VLDB*, **99**, pages 54–65, 1999. 2, 7, 8

[23] Thomas Neumann. **Efficiently compiling efficient query plans for modern hardware**. *Proceedings of the VLDB Endowment*, **4**(9):539–550, 2011. 2, 9

[24] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019. 2, 3, 8

[25] Edgar Frank Codd. **A relational model of data for large shared data banks**. *Communications of the ACM*, **26**(1):64–69, 1983. 5, 6

[26] Thomas M Connolly and Carolyn E Begg. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005. 6

[27] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. **C-store: a column-oriented DBMS**. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018. 7

[28] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006. 7

[29] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. **Super-scalar RAM-CPU cache compression**. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006. 7, 9, 68, 70

[30] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. **The seattle report on database research**. *ACM SIGMOD Record*, **48**(4):44–53, 2020. 7

[31] Goetz Graefe. **Volcano - an extensible and parallel query evaluation system**. *IEEE Transactions on Knowledge and Data Engineering*, **6**(1):120–135, 1994. 7

[32] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. **Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016. 9

[33] Muhammad Naveed, Seny Kamara, and Charles V Wright. **Inference attacks on property-preserving encrypted databases**. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655, 2015. 12, 21, 25, 29

[34] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. **Practical order-revealing encryption with limited leakage**. In *International conference on fast software encryption*, pages 474–493. Springer, 2016. 12

[35] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. **Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation**. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015. 12

[36] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. **On data banks and privacy homomorphisms**. *Foundations of secure computation*, **4**(11):169–180, 1978. 13

[37] Craig Gentry. **Fully homomorphic encryption using ideal lattices**. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009. 13

[38] **Microsoft SEAL (release 3.4)**. `https://github.com/Microsoft/SEAL`, October 2019. Microsoft Research, Redmond, WA. 13

[39] **HElib**. `https://github.com/homenc/HElib`, 2021. 13

[40] Sumeet Bajaj and Radu Sion. **TrustedDB: a trusted hardware based outsourced database engine**. *Proceedings of the VLDB Endowment*, **4**(12):1359–1362, 2011. 15, 30, 31, 88

[41] Jens Köhler, Konrad Jünemann, and Hannes Hartenstein. **Confidential database-as-a-service approaches: taxonomy and survey**. *Journal of Cloud Computing*, **4**(1):1, 2015. 16

[42] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. **Sok: Cryptographically protected database search**. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191. IEEE, 2017. 17

[43] **TPC-H Documentation**. `http://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp`. Accessed: 2021-04-03. 19, 69

[44] Pascal Paillier. **Public-key cryptosystems based on composite degree residuosity classes**. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999. 20

[45] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. **Practical techniques for searches on encrypted data**. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55. IEEE, 2000. 20

[46] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. **Order-preserving symmetric encryption**. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 224–241. Springer, 2009. 20

[47] Hakan Hacigümüş, Bala Iyer, and Sharad Mehrotra. **Efficient execution of aggregation queries over encrypted relational databases**. In *International Conference on Database Systems for Advanced Applications*, pages 125–136. Springer, 2004. 22

[48] Hakan Hacigümüş, Bala Iyer, and Sharad Mehrotra. **Query optimization in encrypted database systems**. In *International Conference on Database Systems for Advanced Applications*, pages 43–55. Springer, 2005. 22

[49] Tingjian Ge and Stan Zdonik. **Answering aggregation queries in a secure system model**. In *Proceedings of the 33rd international conference on Very large data bases*, pages 519–530, 2007. 23, 24, 41

[50] Mihir Bellare, Phillip Rogaway, and Terence Spies. **The FFX mode of operation for format-preserving encryption**. *NIST submission*, **20**:19, 2010. 24

[51] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. **Apache spark: a unified engine for big data processing**. *Communications of the ACM*, **59**(11):56–65, 2016. 25

[52] Savvas Savvides, Julian James Stephen, Masoud Saeida Ardekani, Vinaitheerthan Sundaram, and Patrick Eugster. **Secure data types: a simple abstraction for confidentiality-preserving data analytics**. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 479–492, 2017. 26, 69

[53] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. **Searchable symmetric encryption: improved definitions and efficient constructions**. *Journal of Computer Security*, **19**(5):895–934, 2011. 28

[54] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords*. Citeseer, 1997. 28

[55] Andrew Chi-Chih Yao. **Protocols for secure computations extended abstract**. In *23rd FOCS*. 28

[56] Oded Goldreich. **Towards a theory of software protection and simulation by oblivious RAMs**. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, 1987. 28

[57] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. **Blind seer: A scalable private dbms**. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE, 2014. 28

[58] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. **Arx: an encrypted database using semantically secure encryption**. *Proceedings of the VLDB Endowment*, **12**(11):1664–1678, 2019. 28, 48

[59] Andrew Chi-Chih Yao. **How to generate and exchange secrets**. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986. 29

[60] Vladimir Kolesnikov and Abdullatif Shikfa. **On the limits of privacy provided by order-preserving encryption**. *Bell Labs Technical Journal*, **17**(3):135–146, 2012. 29

[61] Silvio Micali, Oded Goldreich, and Avi Wigderson. **How to play any mental game**. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229. ACM, 1987. 29

[62] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. **Black-box garbled RAM**. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 210–229. IEEE, 2015. 29

[63] Nicolas Anciaux, Mehdi Benzine, Luc Bouganim, Philippe Pucheral, and Dennis Shasha. **GhostDB: querying visible and hidden data without leaks**. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 677–688, 2007. 30

[64] Einar Mykletun and Gene Tsudik. **Incorporating a secure coprocessor in the database-as-a-service model**. In *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05)*, pages 7–pp. IEEE, 2005. 30

[65] Sean W. Smith and David Safford. **Practical server privacy with secure coprocessors**. *IBM Systems Journal*, **40**(3):683–695, 2001. 30

[66] Murat Kantarcioğlu and Chris Clifton. **Security issues in querying encrypted data**. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 325–337. Springer, 2005. 30

[67] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. **Switchless calls made practical in intel SGX**. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, pages 22–27, 2018. 35, 81, 95, 102

[68] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. **Leakage-abuse attacks against order-revealing encryption**. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672. IEEE, 2017. 37, 39, 41, 52

[69] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. **Breaking web applications built on top of encrypted data**. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1353–1364, 2016. 37, 48, 104

[70] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. **Access pattern disclosure on searchable encryption: ramification, attack and mitigation.** In *Ndss*, **20**, page 12. Citeseer, 2012. 37

[71] Michael T Goodrich. **Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data**. In *Proceedings of the*

twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 379–388, 2011. 37

[72] Saba Eskandarian and Matei Zaharia. **Oblidb: Oblivious query processing for secure databases**. *arXiv preprint arXiv:1710.00458*, 2017. 38, 41, 48, 88, 103, 111

[73] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. **T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.** In *NDSS*, 2017. 38

[74] Victor Costan, Ilia Lebedev, and Srinivas Devadas. **Sanctum: Minimal hardware extensions for strong software isolation**. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016. 38, 41, 78

[75] **BigQuery Title**. `https://github.com/google/encrypted-bigquery-client`, 2013. 39

[76] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. **Joins over encrypted data with fine granular security**. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 674–685. IEEE, 2019. 40

[77] **Always Encrypted**. 40

[78] **Virtualization-based Security (VBS)**. 40

[79] Anselme Tueno and Florian Kerschbaum. **Efficient Secure Computation of Order-Preserving Encryption**. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 193–207, 2020. 41

[80] Zheli Liu, Jin Li, Siyi Lv, Yanyu Huang, Liang Guo, Yali Yuan, and Changyu Dong. **EncodeORE: reducing leakage and preserving practicality in order-revealing encryption**. *IEEE Transactions on Dependable and Secure Computing*, 2020. 41

[81] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. **LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection**. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020. 41

[82] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. **CacheOut: Leaking Data on Intel CPUs via Cache Evictions**, 2020. 41

[83] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. **Keystone: An open framework for architecting tees**. *arXiv preprint arXiv:1907.10119*, 2019. 41

[84] **Apple Support - FileVault disk encryption**. `https://support.apple.com/nl-nl/HT204837`, 2021. 45

[85] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. **Lest we remember: cold-boot attacks on encryption keys**. *Communications of the ACM*, **52**(5):91–98, 2009. 46

[86] Patrick Stewin and Iurii Bystrov. **Understanding DMA malware**. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012. 46

[87] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. **A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications**. *IEEE Internet of Things Journal*, **4**(5):1125–1142, 2017. 46

[88] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. **VoltPillager: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface**. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021. 47, 49

[89] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. **Building web applications on top of encrypted data using Mylar**. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 157–172, 2014. 48

[90] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. **Order-preserving encryption revisited: Improved security analysis and alternative solutions**. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011. 52

[91] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. **Why your encrypted database is not secure**. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 162–168, 2017. 53

[92] Morris J Dworkin. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac.* National Institute of Standards & Technology, 2007. 61, 62, 63

[93] Daniel J Bernstein et al. **ChaCha, a variant of Salsa20**. In *Workshop record of SASC*, **8**, pages 3–5, 2008. 62

[94] Gael Hofemeier and Robert Chesebrough. **Introduction to intel AES-NI and intel secure key instructions**. *Intel, White Paper*, 2012. 62

[95] Daniel J Bernstein. **Cryptography in nacl**. *Networking and Cryptography library*, **3**:385, 2009. 62

[96] R. Dolbeau and D.J. Bernstein. **ChaCha8 amd64 AVX implementation**. `https://github.com/floodyberry/supercop/tree/master/crypto_stream/chacha8/dolbeau/amd64-avx2`, 2016. 63

[97] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. **Performance analysis of scientific computing workloads on trusted execution environments**. *arXiv preprint arXiv:2010.13216*, 2020. 78

[98] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. **Insecure until proven updated: analyzing AMD SEV's remote attestation**. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1087–1099, 2019. 78

[99] Sandro Pinto and Nuno Santos. **Demystifying arm trustzone: A comprehensive survey**. *ACM Computing Surveys (CSUR)*, **51**(6):1–36, 2019. 78

[100] Francis Akowuah and Amit Ahlawat. **Protecting sensitive data in android sqlite databases using TrustZone**. In *2018 International Conference on Security & Management*, **2018**, 2018. 78

[101] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. **AEGIS: Architecture for tamper-evident and tamper-resistant processing**. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003. 78

[102] Victor Costan and Srinivas Devadas. **Intel SGX Explained.** *IACR Cryptology ePrint Archive*, **2016**(086):1–118, 2016. 79

[103] Andrew Baumann, Marcus Peinado, and Galen Hunt. **Shielding applications from an untrusted cloud with haven**. *ACM Transactions on Computer Systems (TOCS)*, **33**(3):1–26, 2015. 90

[104] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {**SCONE**}**: Secure Linux Containers with Intel** {**SGX**}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016. 90

[105] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. **Panoply: Low-TCB Linux Applications With SGX Enclaves.** In *NDSS*, 2017. 90

[106] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. **Occlum: Secure and efficient multitasking inside a single enclave of intel sgx**. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020. 90

[107] **Graphene-SGX**. `https://github.com/oscarlab/graphene`, 2021. 90

[108] **SGX monitoring tool**. `https://github.com/fortanix/sgxtop`, 2021. 100

Appendices

A Benchmark machine specifications

Hardware

1. Intel(R) Core(TM) i5-7260U CPU @ 2.20GHz (Turboboost enabled)
2. Intel SGXv1 extension (max 128MB EPC) enabled in BIOS
3. 32GB RAM DDR4 2133 MT/s
4. 500GB NVME SSD

Software

1. Distro: Ubuntu 18.04.4 LTS (gcc 7.5.0, libc 2.27)
2. Linux kernel: 5.4.69 with FSGSBASE patches (necessary for Graphene, see `https://graphene.readthedocs.io/en/latest/building.html`)
3. Intel SGX SDK/PSW: version 2.9.100.2 (built from commit a2b90e326d4e)
4. Intel SGX Driver: `https://github.com/fortanix/linux-sgx-driver` (built from commit a2b90e326d4e)
5. Graphene-SGX: `https://github.com/oscarlab/graphene` (built from commit ab4f14df33a7)

B Graphene-SGX Manifest file

```
# Secure DuckDB manifest file example
#
# This manifest was prepared and tested on Ubuntu 16.04.
```

```
################################## RUNNING ####################################

# Executable to load into Graphene and run. Note that Graphene tries its best
# to find the corresponding manifest file (by appending ".manifest" or
# ".manifest.sgx") based on the executable name, and vice versa. Still, it is
# required to have the explicit name of the executable here.
loader.exec = file:benchmark_runner

################################## GRAPHENE ###################################

# LibOS layer library of Graphene. There is currently only one implementation,
# so it is always set to libsysdb.so. Note that GRAPHENEDIR macro is expanded
# to relative path to Graphene repository in the Makefile as part of the
# build process.
loader.preload = file:$(GRAPHENEDIR)/Runtime/libsysdb.so

# Show/hide debug log of Graphene ('inline' or 'none' respectively). Note that
# GRAPHENEDEBUG macro is expanded to inline/none in the Makefile as part of the
# build process.
loader.debug_type = $(GRAPHENEDEBUG)

################################## ARGUMENTS ##################################

# Read application arguments directly from the command line. Don't use this on
    production!
loader.insecure__use_cmdline_argv = 1

################################## ENV VARS ###################################

# Specify paths to search for libraries. The usual LD_LIBRARY_PATH syntax
# applies. Paths must be in-Graphene visible paths, not host-OS paths (i.e.,
# paths must be taken from fs.mount.xxx.path, not fs.mount.xxx.uri).
#
# In case of Redis:
# - /lib is searched for Glibc libraries (ld, libc, libpthread)
# - $(ARCH_LIBDIR) is searched for Name Service Switch (NSS) libraries
loader.env.LD_LIBRARY_PATH = /lib:$(ARCH_LIBDIR):/lib/x86_64-linux-gnu:/usr/lib
    /x86_64-linux-gnu:/home/sam/Documents/main-experiment-runner/build/release/
    src/

################################## MOUNT FS  ##################################

# General notes:
# - There is only one supported type of mount points: 'chroot'.
# - Directory names are (somewhat confusingly) prepended by 'file:'.
# - Names of mount entries (lib, lib2, lib3) are irrelevant but must be unique.
# - In-Graphene visible path names may be arbitrary but we reuse host-OS URIs
```

```
#   for simplicity (except for the first 'lib' case).

# Mount host-OS directory to Graphene glibc/runtime libraries (in 'uri') into
# in-Graphene visible directory /lib (in 'path'). Note that GRAPHENEDIR macro
# is expanded to relative path to Graphene repository in the Makefile as part
# of the build process.
fs.mount.lib.type = chroot
fs.mount.lib.path = /lib
fs.mount.lib.uri = file:$(GRAPHENEDIR)/Runtime

# Mount host-OS directory to Name Service Switch (NSS) libraries (in 'uri')
# into in-Graphene visible directory /lib/x86_64-linux-gnu (in 'path').
fs.mount.lib2.type = chroot
fs.mount.lib2.path = $(ARCH_LIBDIR)
fs.mount.lib2.uri = file:$(ARCH_LIBDIR)

# Mount host-OS directory to NSS files required by Glibc + NSS libs (in 'uri')
# into in-Graphene visible directory /etc (in 'path').
fs.mount.etc.type = chroot
fs.mount.etc.path = /etc
fs.mount.etc.uri = file:/etc

fs.mount.usr.type = chroot
fs.mount.usr.path = /usr
fs.mount.usr.uri = file:/usr

# This is ugly but it allows to use the duckdb shared lib
fs.mount.duckdb.type = chroot
fs.mount.duckdb.path = /home/sam/Documents/main-experiment-runner/build/release
    /src
fs.mount.duckdb.uri = file:/home/sam/Documents/main-experiment-runner/build/
    release/src

############################### SGX: GENERAL ###################################

# Set enclave size (somewhat arbitrarily) to 1024MB. Recall that SGX v1
    requires
# to specify enclave size at enclave creation time. If Redis exhausts these
# 1024MB then it will start failing with random errors. Greater enclave sizes
# result in longer startup times, smaller enclave sizes are not enough for
# typical Redis workloads.
sgx.enclave_size = 16384M
#sgx.enclave_size = 1024M
#sgx.enclave_size = 2048M
#sgx.enclave_size = 2048M

# Set maximum number of in-enclave threads (somewhat arbitrarily) to 8. Recall
```

```
# that SGX v1 requires to specify the maximum number of simulteneous threads at
# enclave creation time.
#
# Note that internally Graphene may spawn two additional threads, one for IPC
# and one for asynchronous events/alarms. Redis is technically single-threaded
# but spawns couple additional threads to do background bookkeeping. Therefore,
# specifying '8' allows to run a maximum of 6 Redis threads which is enough.
sgx.thread_num = 8
sgx.rpc_thread_num = 0

############################## SGX: TRUSTED LIBS ##############################

# Specify all libraries used by Redis and its dependencies (including all
# libraries which can be loaded at runtime via dlopen). The paths to libraries
# are host-OS paths. These libraries will be searched for in in-Graphene
    visible
# paths according to mount points above.
#
# As part of the build process, Graphene-SGX script (`pal-sgx-sign`) finds each
# specified library, measures its hash, and outputs the hash in auto-generated
# entry 'sgx.trusted_checksum.xxx' in auto-generated redis-server.manifest.sgx.
# Note that this happens on the developer machine.
#
# At runtime, during loading of this library, Graphene-SGX measures its hash
# and compares with the one specified in 'sgx.trusted_checksum.xxx'. If hashes
# match, this library is trusted and allowed to be loaded and used. Note that
# this happens on the client machine.

# Glibc libraries. ld, libc, libm, libdl, librt provide common functionality;
# pthread is needed because Redis spawns helper threads for bookkeeping.
sgx.trusted_files.ld = file:$(GRAPHENEDIR)/Runtime/ld-linux-x86-64.so.2
sgx.trusted_files.libc = file:$(GRAPHENEDIR)/Runtime/libc.so.6
sgx.trusted_files.libm = file:$(GRAPHENEDIR)/Runtime/libm.so.6
sgx.trusted_files.libdl = file:$(GRAPHENEDIR)/Runtime/libdl.so.2
# sgx.trusted_files.librt = file:$(GRAPHENEDIR)/Runtime/librt.so.1
sgx.trusted_files.libos = file:$(GRAPHENEDIR)/Runtime/liblibos.so.1
sgx.trusted_files.libpthread = file:$(GRAPHENEDIR)/Runtime/libpthread.so.0
sgx.trusted_files.duck_db = file:/home/sam/Documents/main-experiment-runner/
    build/release/src/libduckdb.so
sgx.trusted_files.libgcc = file:/lib/x86_64-linux-gnu/libgcc_s.so.1
sgx.trusted_files.libstdcpp = file:/usr/lib/x86_64-linux-gnu/libstdc++.so.6
sgx.trusted_files.libcrypto = file:/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1

############################## SGX: ALLOWED FILES
    ##############################

# Specify all non-static files used by app. These files may be accessed by
```

```
# Graphene-SGX but their integrity is not verified (Graphene-SGX does not
# measure their hashes). This may pose a security risk!

sgx.allowed_files.duckdb_tpch_list = file:duckdb_benchmark_data/tpch.list
sgx.allowed_files.duckdb_tpch_sql = file:duckdb_benchmark_data/tpch.sql
sgx.allowed_files.duckdb_tpch_customer = file:duckdb_benchmark_data/
    tpch_customer.csv
sgx.allowed_files.duckdb_tpch_lineitem = file:duckdb_benchmark_data/
    tpch_lineitem.csv
sgx.allowed_files.duckdb_tpch_nation = file:duckdb_benchmark_data/tpch_nation.
    csv
sgx.allowed_files.duckdb_tpch_orders = file:duckdb_benchmark_data/tpch_orders.
    csv
sgx.allowed_files.duckdb_tpch_part = file:duckdb_benchmark_data/tpch_part.csv
sgx.allowed_files.duckdb_tpch_partsupp = file:duckdb_benchmark_data/
    tpch_partsupp.csv
sgx.allowed_files.duckdb_tpch_region = file:duckdb_benchmark_data/tpch_region.
    csv
sgx.allowed_files.duckdb_tpch_supplier = file:duckdb_benchmark_data/
    tpch_supplier.csv
sgx.allowed_files.sgx_stats = file:sgx_stats
sgx.allowed_files.log_file = file:benchmark_log.txt
sgx.allowed_files.out_file = file:benchmark_output.txt
sgx.allowed_files.duckdb_database = file:duckdb_benchmark_db.db
sgx.allowed_files.duckdb_database_wal = file:duckdb_benchmark_db.db.wal
sgx.allowed_files.duckdb_database_tmp = file:duckdb_benchmark_db.db.tmp
sgx.allowed_files.custom_query = file:custom_query.sql
```