

# VCRYPT: Leveraging Vectorized and Compressed Execution for Client-side Encryption

Charlotte Felius  
CWI  
Amsterdam, Netherlands  
felius@cwi.nl

Peter Boncz  
CWI & MotherDuck  
Amsterdam, Netherlands  
boncz@cwi.nl

## ABSTRACT

VCRYPT is a novel extension on DuckDB that enables fine-grained client-side [en/de]ryption in a performance- and storage-efficient manner, by exploiting columnar compression as well as vectorized and compressed execution. We designed VCRYPT such that in analytical queries, typically (i) data can be encrypted and decrypted batch-at-a-time instead of value-at-a-time, and (ii) the extra storage for cryptographic *nonces* gets compressed away. We also demonstrate the use of VCRYPT inside MotherDuck, leveraging its *hybrid* processing model that evaluates SQL queries partly on a client DuckDB and partly on a cloud DuckDB, to achieve *secure hybrid execution*. This provides security even if the cloud server is untrusted, by forcing the [en/de]ryption of sensitive data to happen only client-side, while still allowing useful cloud-side work like filters and joins.

## 1 INTRODUCTION

Vcrypt, short for *Vectorized Cryptography*, is a novel extension on DuckDB [8], a widely used online analytical processing database management system, that enables user-level encryption, per value, but optimized for performance. The extension consists of scalar functions for the encryption and decryption of single values. It represents encrypted scalar types as a struct type, whose members include a field that holds the encrypted value, and the other fields holding encryption meta-data. These latter fields are designed such that (i) columnar compressed storage will typically reduce them to negligible space; and such that (ii) vectorized execution will typically be able to [en/de]crypt a batch of values in a single call to crypto primitives, greatly enhancing computational efficiency and (iii) *compressed execution* keeps in-flight encrypted data small and avoids duplicate work being performed. Our approach shines in DuckDB, as it leverages some of its specific data compression methods as well as its ability for compressed execution using its special vector representations. However, the properties of Vcrypt provide this efficiency potential for *any* compressed columnar store (including Parquet files) and vectorized query engine.

Until now DuckDB is lacking any encryption functionality except for Parquet encryption [4]. To distinguish encrypted values from regular ones, Vcrypt introduces in DuckDB novel user-defined types, representing encrypted values for each normal (plaintext) type. It further introduces simple scalar UDF functions to create such new encrypted values, i.e. `val_enc = encrypt(val, key_id)`; and transform the resulting encrypted values back into plaintext, i.e. `val = decrypt(enc_val, key_id)`. These functions can be used explicitly in query expressions; but we also envision them to be used in VIEW definitions on tables with encrypted

columns, where the VIEW definition wraps the encrypted column in a `decrypt()` call in its topmost SELECT list in order to make end-users seamlessly query encrypted data. In addition, we implemented a key management mechanism which utilizes hierarchical encryption, and uses the DuckDB secrets manager to redact encryption keys. Hence the `key_ids` are not the encryption keys themselves, but identifiers for the key in a key management system for which the user will have to authenticate. While VCRYPT theoretically encrypts one value-at-a-time, it is designed to minimize storage and computational overheads of encryption by leveraging *columnar encryption*, *vectorized processing* and *compressed execution* [1], often present in modern analytical systems such as Velox [7] and DuckDB [8].

**Hybrid Query Processing.** As proposed in [5], to enhance security we argue that sensitive data could exclusively be encrypted and decrypted client-side, similar to Monomi [9]. Ideally, the part of the query that does not contain sensitive data should be computed at the server, while the part of the query that includes sensitive data is exclusively processed client-side. More specifically, from the point that a query plan involves the processing of sensitive data, the resulting part of the plan should be executed client-side and thereafter no further processing should happen on the server. To achieve such a hybrid form of query execution, we make use of MotherDuck [3] to enable *secure hybrid query processing*. MotherDuck provides a cloud-computing platform for DuckDB, by enabling hybrid query processing (i.e. *dual execution*). We extend MotherDuck's *hybrid optimizer* to implement a novel constraint to force the encryption and decryption of sensitive data client-side, herewith improving security by giving only the end-user control of the sensitive data.

**Demo.** We will demonstrate DuckDB SECRET management and encryption or decryption of table data in a stock DuckDB on a laptop. We will show the speed and storage-size advantage of vectorized encryption compared over tuple-at-a-time, by [en/dis]abling our optimizations. Then, we will connect this DuckDB to the MotherDuck service and demonstrate how encrypted data can be stored in the cloud. We then showcase *secure hybrid query processing*, where complex queries are executed on cloud-data, with only client-side decryption. We will profile query performance with EXPLAIN ANALYZE to demonstrate and explain the efficiency of our approach.

## 2 VCRYPT OVERVIEW

DuckDB has a rich and easy-to-use extensibility mechanism, and provides an infrastructure to add extensions as C++ modules, that consist of e.g. parser extensions, custom data-types, new scalar functions and optimizer rules. We implemented VCRYPT as such a DuckDB extension. Database encryption, especially when done at the level of individual values, can lead to severe performance bottlenecks, increasing its cost and reducing its utility for analytical workloads currently. These bottlenecks are

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-89318-099-8 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

(i) inflated storage, since a large nonce ("number used only once") needs to be stored alongside every value to hide the effects of deterministic encryption<sup>1</sup>, and (ii) encryption/decryption needs to operate on at least 0.5-4KB of data in order to reach good throughput, as shown in Figure 1, hence individual values are at least 100x too small. We port existing cryptographic primitives from the HTTPFS extension to the VCRYPT extension to avoid an inter-extension dependency. For *randomized* encryption we use AES Countermode (CTR) to avoid a large storage overhead of a 16-byte tag (used for verification) per value.

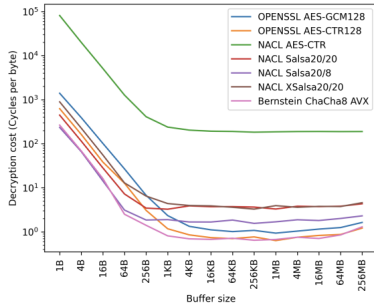


Figure 1: Decryption overhead in cycles (y-axis) per byte (x-axis), image from [2].

## 2.1 Representing Encrypted Data

In VCRYPT, an encrypted value is self-contained, i.e. a value can always be encrypted or decrypted individually. However, to be self-contained, each encrypted value needs to store corresponding metadata to be able to correctly decrypt. If only a single scalar value is encrypted, its encrypted size will be at least 16 bytes. For a 32-bits integer, an AES standard 12-byte nonce plus 16-byte encrypted value represents a storage overhead of 7x, and the lack of compressibility of encrypted data increases that to typically >20x. Also, Figure 1 shows that encryption/decryption on 16 bytes at-a-time leads to two orders of magnitude lower throughput than achievable on 4KB buffers. We do note that these inefficiencies are much lessened if data would be encrypted on the granularity of data pages, rather than values, and this could be especially efficient in compressed column stores [2, 4]. In-database encryption, however, requires users to trust the database server. In contrast, by managing encryption in a user-level extension, outside the control of the main database system, users do not need to trust the database server, and such an approach also enables purely client-side decryption, as we will describe later.

A **User-defined Type** is used to distinguish encrypted types from regular types in VCRYPT. This is similar to the GEOMETRY type from the DuckDB Spatial extension. We identify ENCRYPTED types by prefixing any existing DuckDB type with "E\_", e.g. E\_INTEGER will hold encrypted integers. The ENCRYPTED type is necessary to indicate which types are actually encrypted in a BLOB. We use the DuckDB support for STRUCT types, to store all data and meta-data sub-fields together, as described below.

**Nonce Compression.** A useful property of AES-CTR is that a nonce has to be *unique*, but not *random*. This uniqueness property is crucial since re-use of the same nonce with the same key leaks information when similar data is encrypted. We therefore

<sup>1</sup>The same data would otherwise produce the same encrypted value, leaking information.

represent an encrypted value as a DuckDB STRUCT type, containing an 8-byte higher part of the 12-byte *number used once* (i.e. nonce), a 4-byte lower part of the nonce, plus a 4-byte *counter*. The higher and lower part of the nonce together with the counter construct the *Initialization Vector (IV)*, which is used to randomize encrypted values. In the DuckDB STRUCT we also store an encrypted *cipher* value (typically 1 byte) and an encrypted BLOB, holding the actual encrypted values.

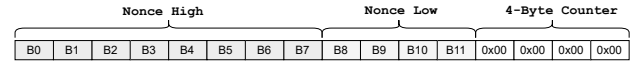


Figure 2: Splitting makes the nonce highly compressible: high and low will be RLE-encoded, and counter DELTA-encoded.

**Encrypted BLOB.** AES-CTR operates on a granularity of 16 bytes, and we call such units *tiles*. In VCRYPT, encrypted values are stored in the encrypted BLOB, which holds a *batch* of at least 128 co-encrypted values. The first byte of any VCRYPT encrypted BLOB is used for versioning, as we anticipate evolving it in the future, e.g. with data compression. A BLOB can contain more values for thin data types, as the second constraint is that the encrypted BLOB consists of at least 32 tiles (32x16=512 bytes). The nonce used for subsequent tiles in one BLOB is automatically incremented by OpenSSL; so we store the nonce and counter (together IV) for the first tile in a batch. The reason to split the IV into three separate values, where  $IV = High \ll 64 + Low \ll 32 + Counter$  (see figure 2), is to make this representation highly compressible in column stores. When bulk-inserting data, large stretches of adjacent values can then have identical High, Low values; leading to high compressibility of these columns. DuckDB supports e.g. constant compression, when all column values in a row-group are equal. Further, each batch of tuples will have the same Counter and encrypted BLOB value, making those RLE-compressible. The 4-byte Counter is increased after each batch of tuples with the amount of tiles in the batch, providing a unique nonce for each tile. We decided to split the nonce to make use of mature support in databases for 64- and 32-bits types as opposed to 128-bits types. Rather than one 64-bits Low we also split off Counter, because Parquet (1.0) does not support delta-compression, hence augmenting an (in that case, 64-bits) Low in subsequent values would remove the opportunity for constant compression.

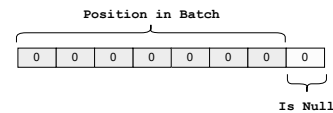


Figure 3: Breakdown of 1-byte *cipher* that represents both the encrypted position of the value in a *batch* as well as the encrypted nullability.

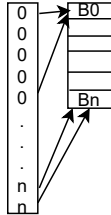
**Cipher** is a small integer value (Figure 3) that holds the array index of the value in the BLOB, which can be seen as an array holding all values in the batch. We use one additional bit (the lowest bit) to store whether the value is NULL. The cipher field is encrypted using a XOR with a (simple) hash of the first plaintext value in the batch, so position and nullability cannot be inferred before decryption. The simple hash uses fast prime multiplication; but to avoid exposing the cipher in case the plaintext is simply 0, we calculate the cipher with formula 1.

$$Cpr = (P_1 \oplus T) * P_2 \quad (1)$$

$Cpr$  corresponds to the cipher value,  $P_1$  is the first prime,  $P_2$  is the second prime and  $T$  is the 64-bit plaintext. Given a batch size of 128, we need 7 bits for the array index, plus one bit for nullability, so cipher typically fits one byte.

**Variable Sized Data.** The above defines how all DuckDB scalar types are encrypted by `VCRYPT`. For variable-sized data types we need to keep extra information for decrypting a value, such as a byte position and a length. We also concatenate 128 subsequently inserted variable-sized data type values in a batch into one BLOB that gets encrypted and is shared by all values in that batch. To be able to decrypt all individual values, we concatenate 128 byte offsets and store them at the beginning of the encryption buffer. When encrypting data, we then also encrypt these byte offsets, and thus do not leak any information about the individual size of the encrypted data. With storing the byte offset from each individual value, the length  $l$  of an encrypted value is the difference between two subsequent byte offsets  $b$ , such that  $l = \Delta b$ . In other words, we can calculate  $\Delta b$  by subtracting the byte position at a certain index from the byte position at the subsequent index, i.e.  $\Delta b = b_{i+1} - b_i$ . Each value in the cipher column of a `VCRYPT` struct now corresponds to an index in the byte offset array. In our approach, we hence do not leak individual string lengths, not even approximately, as systems often pad strings to the closest multiple of 16 bytes. `VCRYPT` only leaks the approximate length of a concatenated batch of 128 strings.

| rowid | High   | Low    | Ctr | Cpr | BLOB          |
|-------|--------|--------|-----|-----|---------------|
| 0     | $hi_0$ | $lo_0$ | 0   | 52  | 0x7F9A...42F0 |
| 1     | $hi_0$ | $lo_0$ | 0   | 90  | 0x7F9A...42F0 |
| 2     | $hi_0$ | $lo_0$ | 0   | 143 | 0x7F9A...42F0 |
| 3     | $hi_0$ | $lo_0$ | 0   | 233 | 0x7F9A...42F0 |
| 4     | $hi_0$ | $lo_0$ | 0   | 6   | 0x7F9A...42F0 |
| ⋮     | ⋮      | ⋮      | ⋮   | ⋮   | ⋮             |
| 127   | $hi_0$ | $lo_0$ | 0   | 194 | 0x7F9A...42F0 |
| 128   | $hi_0$ | $lo_0$ | 1   | 11  | 0x46E3...17AA |
| ⋮     | ⋮      | ⋮      | ⋮   | ⋮   | ⋮             |
| 255   | $hi_0$ | $lo_0$ | 1   | 83  | 0x46E3...17AA |



**Figure 4:** Left: example table representation of encrypted 64-bits BIGINTs. In compressed form, the High and Low columns are constant, the Counter (Ctr) gets bitpacked as well as the random cipher (Cpr), which varies between [0,8] bits. The counter increments every batch, which often contains 128 values, since we [en/de]crypt per batch. Note that the counter internally increases *within* a batch. The same BLOB repeats for 128 consecutive encrypted values. Right: Dictionary Vector in-flight during query processing. In a 2048-element DuckDB vector, there are  $n=16$  unique encrypted BLOBs in a dictionary.

## 2.2 Compressed Execution

The gist of our approach is that multiple subsequently encrypted (and supposedly stored) values will share the same encrypted BLOB (Figure 4); which in columnar storage will trigger the duplicates to be eliminated with RLE compression. Furthermore, DuckDB uses vectorized execution, with support for *compressed execution*, i.e. query processing on still (partially) compressed data. For this purpose, beyond PLAIN vectors that store data uncompressed in an array, DuckDB supports various compressed subclasses of their Vector class. A Dictionary Vector stores two

arrays: one with codes (indexes) that point in a separate array with values (the dictionary). Also, a Constant Vector is a vector that stores only one value, and represents a vector where all entries have that value. When reading our RLE-encoded encrypted BLOB, the DuckDB scan will represent them either as a Dictionary Vector or a Constant Vector. Therefore, the values from RLE will be stored only once in the compressed vector that is used for query execution. The vectorized UDF methods for decryption and encryption in `VCRYPT` take advantage of this, as they have support for Constant and Dictionary Vectors for their parameters. This means that encryption and decryption will be performed only once for each value, and on buffers of at least 512 bytes, which as we mentioned before (see Figure 1) improves AES-CTR throughput by two orders of magnitude.

## 2.3 Key Management

In databases, keys or passwords are often exposed in the write-ahead-log (WAL) or in logs, e.g. used for debugging. To minimize the exposure of this sensitive data, `VCRYPT` authenticates the user only once and automatically uses available encryption keys. To enable this mechanism withing DuckDB, we make use of the DuckDB Secrets Manager using the `CREATE SECRETS` syntax, as shown in listing 1. The keyword `VCRYPT` implies that the DuckDB SECRET is intended for columnar encryption in the `VCRYPT` extension. The user is then responsible to name their secret (in the example 'my\_secret'), and insert a user-defined key after `TOKEN`. Moreover, the user can define the desired length of the columnar key, which can either be 16, 24 or 32 bytes.

```

1 CREATE SECRET my_secret (
2   TYPE VCRYPT,
3   TOKEN 'secret_key',
4   LENGTH 16);
5
6 encrypt(value, "my_secret")

```

**Listing 1:** Defining a DuckDB SECRET for an encrypted column in `VCrypt`.

To further strengthen its security, `VCRYPT` computes an internal encryption key using `TOKEN` when the UDFs `encrypt()` or `decrypt()` are called. This internal key is computed by using a deterministic sha-256 HMAC function from OpenSSL that calculates a HMAC. Since we support various key sizes, we truncate the computed HMAC to the prospective key length defined by the user. For better security, the user needs to store the DuckDB secret either locally or in a third-party authentication system. We aim to make the management of encryption keys compatible with Key Vault or any Key Management System (KMS).

## 3 INTEGRATION WITH MOTHERDUCK

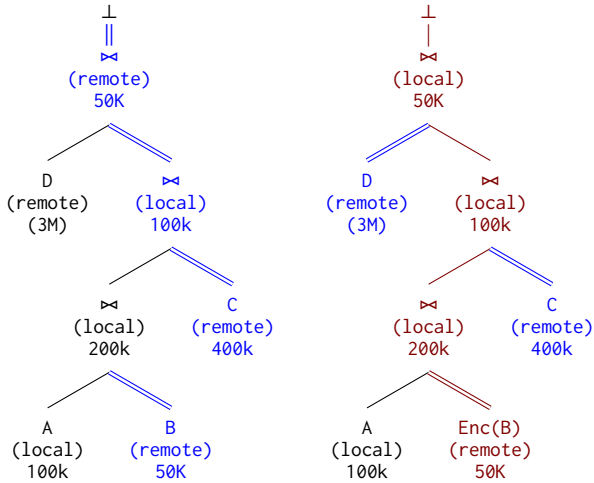
MotherDuck is a serverless analytics cloud-service using DuckDB, with a twist: its client libraries have an embedded DuckDB as well and queries can execute partially on the client and partially in the cloud. This hybrid architecture brings also novel advantages for security; e.g. sensitive data can be largely processed locally, at the client, if the server is not completely trusted.

### 3.1 Hybrid Query Optimizer

MotherDuck is like `VCRYPT` a DuckDB extension and introduces *bridge operators* that can transfer data from remote (server) to local (client) and vice versa. It also introduces new DuckDB optimizer rules that perform hybrid query optimization and inserts



bridge operators as needed. The task of the hybrid optimizer in MotherDuck is to determine where the operators from a query plan should be executed, i.e. remote (server-side) or locally (client-side). As its goal is to run queries as efficiently as possible, it takes into account where queried tables are located (either locally or remotely) and then tries to minimize data transfer cost using cardinality estimates [6].



(a) Data Transfer Cost-Based Query Plan in MotherDuck (b) Transformed Hybrid Query Plan for VCRYPT

**Figure 5: Hybrid Query Plans for MotherDuck and VCRYPT.** The non-sensitive hybrid query plan from MotherDuck (a) gets transformed into (b) when remote column B contains encrypted data Enc(B). The bridge operator  $\parallel$  indicates a transfer to a different site (e.g. remote to local). The blue color indicates such a transfer with non-sensitive data, and the dark red color indicates when sensitive data is included. Figure adapted from [6].

To ensure that sensitive data is only processed client-side, we modified the hybrid optimizer rule to force that all decryption operations in a query plan, and all parent operators in that plan through which decrypted data flows, are executed locally. In addition to modifying the MotherDuck client-side extension, we plan in the future to add a VCRYPT optimizer rule. This rule is relevant both with and without MotherDuck, and aims to optimize DuckDB query plans that handle VCRYPT encrypted-data. An important consideration is that operations that inhibit data to stay represented as Dictionary Vectors will cause encrypted data to become much bigger, and queries slower, as the compressing effect of Dictionary Vectors will be removed. This specifically holds for encrypted data flowing into build-sides of joins, but also for sorting and aggregation. For the latter-two operations, it is important to push decryption below them. This could also be a strategy for joins, but an alternative is to flip probe- and build-sides. A novel designed cost model should decide what approach to take.

## 4 DEMONSTRATION

We split the demonstration up in two parts: (i) *Performance and Storage* of VCRYPT, where we give an overview of the performance and storage overhead. The second part consist of *Secure Hybrid Execution* where VCRYPT is integrated with MotherDuck to enable client-side decryption.

**(i) Performance and Storage.** For the performance and storage overview, the participant will first define a DuckDB SECRET used for encryption (listing 1). We will pre-load TPC-H with SF1 to DuckDB, and let the participant decide which columns to encrypt. For the decryption the participant is encouraged to define a VIEW in which the encrypted columns are wrapped. After encrypting the specified columns, the participant can perform any arbitrary query on the encrypted data. To benchmark the performance and storage usage, we use the TPC-H benchmark suite and show the execution speed as well as storage overhead in the DuckDB CLI. We will compare storage and execution speed of VCRYPT using various batch sizes in multiples of the the default 128 values. In addition, we compare the VCRYPT *vectorized* implementation against a naive *per-value* implementation, which is two orders of magnitude slower and requires one order of magnitude more space. An example of an adapted TPC-H query using VCRYPT is given in listing 2, assuming that l\_shipdate is encrypted.

```

1 SELECT
2     SUM(l_extendedprice * l_discount) AS revenue
3 FROM lineitem WHERE
4     decrypt(l_shipdate, 'my_secret') >= DATE
5     1994-01-01
6     AND decrypt(l_shipdate, 'my_secret') < DATE
7     1995-01-01
8     AND l_discount BETWEEN 0.05 AND 0.07
9     AND l_quantity < 24;

```

**Listing 2: Example (adapted) TPC-H Q6 including decrypt() scalar UDF to decrypt the l\_shipdate column.**

**(ii) Secure Hybrid Execution with MotherDuck.** In the second part of the demo the participant can inspect remote tables through the MotherDuck UI. We will provide the user with predefined VIEWS that can be executed client-side, to mimic TPC-H queries involving encrypted data, and ensure that this data is decrypted only client-side. On top of this, the user can perform any arbitrary query that involves encrypted data. In short, the user can thus (i) execute arbitrary queries in the MotherDuck UI, including decrypting sensitive data on the locally, (ii) inspect hybrid query plans containing encrypted data with e.g. EXPLAIN ANALYZE and (iii) inspect end-to-end query execution of TPC-H queries involving encrypted data, to get an idea of the encryption overhead while using secure hybrid execution.

## REFERENCES

- [1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.
- [2] Sam Ansmink. 2021. Encrypted Query Processing in DuckDB. *Master's thesis. Vrije Universiteit Amsterdam, The Netherlands* (2021).
- [3] RJ Atwal, Peter A Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, et al. 2024. MotherDuck: DuckDB in the cloud and in the client. In *CIDR*.
- [4] Ilaria Battiston, Lotte Feliuss, Sam Ansmink, Laurens Kuiper, and Peter Boncz. 2024. DuckDB-SGX2: The Good, The Bad and The Ugly within Confidential Analytical Query Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–5.
- [5] Lotte Feliuss. 2024. Enhancing Security for Columnar Storage and Data Lakes. *Proceedings of the VLDB Endowment*. ISSN 2150 (2024), 8097.
- [6] Jeewon Heo. 2024. Cost-Based Hybrid Query Optimization in MotherDuck. *Master's thesis. Vrije Universiteit Amsterdam, The Netherlands* (2024).
- [7] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.
- [8] Mark Raasveldt and Hannes Mühlhausen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
- [9] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. (2013).