HoTEE: Bridging Heterogeneous Encrypted Databases via Secure and Logarithmic-complexity Indexing (reviseV)

Anonymous Submission #xxx

Abstract

An *Encrypted Database (EDB)* utilizes either homomorphic encryption (HE) or trusted execution environment (TEE) to store, search, and process sensitive data on the public cloud. Organizations choose HE or TEE by their distinct trust, and it is highly desirable for them to share data and make queries across EDBs. However, EDBs using HE and EDBs using TEE are now isolated, because HE data does not share the same usability as TEE data where data can be securely decrypted to support general queries with logarithmic search complexity.

We propose HoTEE, the first system that can support general queries to heterogeneous EDBs on both TEE and HE data with logarithmic complexity. HoTEE features a new *Merkle rangeFilter Tree* (MFT) abstraction, which extracts *deterministic* and *order-preserving*, two essential properties for realizing general queries on ciphertexts, and embeds these properties into a secure tree-based data structure that provides logarithmic search complexity, bypassing the inherent generality and inefficiency problem caused by operation-specific encryption schemes used by prior work. Moreover, by searching on MFTs, HoTEE precisely locates and interacts with query-dependent organizations to securely transform and aggregate query results from heterogeneous EDBs. We implemented HoTEE and applied it to two popular databases: MySQL and WiredTiger. Evaluation shows that HoTEE is the only system that supports general queries on heterogeneous EDBs, while achieving up to 93.8% lower latency and up to 15.2X higher throughput compared to baselines.

1 Introduction

The prosperity of cloud computing and confidential computing fosters the deployment of *encrypted databases* (EDB) on the public cloud [3, 5, 37, 42]. An EDB instance (for short, an EDB) is increasingly *heterogeneous*: typically, it simultaneously enables both homomorphic encryption (HE) and trusted execution environment (TEE) to store, search, and process sensitive data on HE hosts (i.e., nodes) and TEE nodes. For example, a ByteDance's Jeddak database instance [1] simultaneously enables Jeddak-TEE nodes to store and analyze intra-company data and Jeddak-HE nodes to process highly sensitive inter-company data. The Canadian government has launched an EDB project using TEE nodes to store citizens' statistics and developing HE nodes to link data dispersed across public institutions [53].

The confidential benefit of EDB makes it particularly attractive on creating a principled *unified* query system, to be created in this paper, for multiple parties. Specifically, those parties who collectively analyze sensitive data and use

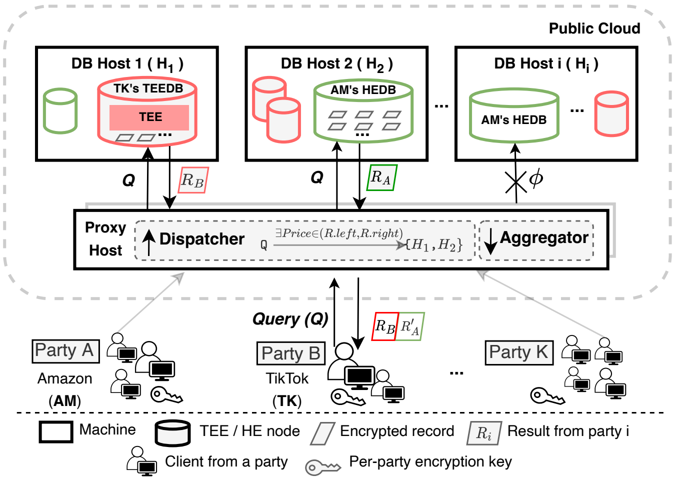


Figure 1. A principled unified query system proposed by HoTEE.

heterogeneous EDB (containing both HE and TEE nodes), as depicted in Figure 1. Such a unified query system introduces two unique benefits. First, it facilitates queries to search on heterogeneous EDB nodes simultaneously, bridging the silos between TEE and HE data. Second, it enables the aggregation of queried results from heterogeneous EDB nodes owned by multiple parties for collaborative analytics.

In order to realize the benefits, the unified system we create in this paper will require a principled index component (for short, *proxy*), which should be able to efficiently dispatch general queries (e.g., range queries) to execute in relevant EDB nodes and aggregate all queried data for the clients.

The example in Figure **??** illustrates the unique benefits of the unified query system: supposing TikTok and Amazon both have an *item* table, TikTok stores the table in TEE nodes (can be decrypted) and Amazon stores the table in HE nodes (can never be decrypted), and they share data to collectively make queries for price analytics. A TikTok salesman (i.e., client) can submit a heterogeneous query as in Figure **??**, which searches on both TikTok's data in TEE nodes and Amazon's data in HE nodes. The heterogeneous query must be dispatched by the proxy to execute in relevant EDB nodes and the searched results should be aggregated by the proxy.

Unfortunately, building such a principled unified system remains an open problem; the core difficulty lies in the principled index for dispatching queries across TEE and HE nodes. Specifically, HE data does not share the same usability as TEE data when data is flowing across different nodes, because unlike TEE that can decrypt data and then search plaintexts on a TEE-shielded B$^+$-tree [3, 42], HE data is fundamentally infeasible to be decrypted and searched.

Despite much academic and industrial effort in developing EDBs, all existing EDBs (categorized into two lines of work) are infeasible to be extended to create the principled index, as they face a fundamental *generality-efficiency* dilemma identified in our paper. The first line of work (e.g., HElibDB [5], FHEDB [31]) uses specific HE functions (e.g., FLT [5]) to evaluate only the equivalence of HE data by linearly scanning the whole dataset, which makes this line of work unsuitable for the goals of this paper, as it is neither general (lacking support for range queries) nor efficient (given massive and constantly growing amounts of data) on HE nodes. The other line of work (e.g., CryptDB [37], Monomi [45]) supports logarithmic complexity equality and range queries on HE data by using distinct encryption schemes (e.g., DET [10] and OPE [4]) to support either type of search operation. However, this line of work inherits the limitations of inefficient cryptographic computations and does not support dynamic changing or migrating data across the same HE nodes or TEE nodes.

Overall, all existing EDBs use special encryption schemes to support specific search operations, which inevitably inherits the generality and inefficiency problems of using these encryption schemes.

This paper presents HoTEE[1], the first unified query system that can support general queries on heterogeneous EDB (i.e., simultaneously enabling HE and TEE nodes) from multiple parties with logarithmic search complexity. HoTEE tackles the generality-efficiency dilemma via a new confidential index abstraction called *Merkle rangeFilter Tree* (MFT). Our principled insight is that, instead of using operation-specific encryption schemes, we extract *deterministic* and *order-preserving*, two essential properties in these schemes for realizing general queries on ciphertexts, to create a secure and searchable data structure. With MFT, we ensure ciphertexts in the index are deterministically encrypted and order-preserved, and can be searched via a tree-based data structure with logarithmic search complexity. Thus MFT bypasses the inherent generality and inefficiency problem of prior work that uses operation-specific encryption schemes.

Actually, realizing the confidential index of MFT is challenging, because searching on heterogeneous EDBs (containing HE data that cannot be decrypted, as shown in Figure **??**) without compromising confidentiality is difficult, let alone simultaneously support general queries on heterogeneous data with logarithmic complexity.

Our technical insight is that, the traditional bloom filter [11] intrinsically holds the confidential and deterministic properties that we require to create a principled index: a bloom filter deterministically checks the existence of data objects by comparing each element in a filter while hiding the plaintexts of all objects. However, a bloom filter is not searchable, thus we break a traditional *monolithic* bloom filter into *subfilters*, where each subfilter represents only one data object. By stacking subfilters into a Merkle-tree style recursive data structure, MFT executes equality queries by traversing and comparing each subfilter node logarithmically (§4.1). Furthermore, to support range queries, we propose *biased range* to securely patch the order-preserving property to subfilters, such that subfilters can preserve the same numerical orders as their corresponding plaintexts (§4.2).

We implemented HoTEE on the CryptDB codebase [37], a prevalent modular framework for evaluating distributed encrypted databases. We built two HE databases with HoTEE by augmenting prominent databases: HoTEE-MySQL and HoTEE-WiredTiger. MySQL [32] is the most widely used SQL backend for encrypted databases; WiredTiger [13] is a popular key-value store that is the default NoSQL backend for MongoDB. We run HE databases with the open-source TEE database Azure EdgelessDB [3] for heterogeneous query, and we design a secure data aggregation protocol (SDA) to cryptographically transform and aggregate searched multi-party data from heterogeneous EDB nodes (§4.3). We compared HoTEE with two famous HE databases CryptDB [37] and HElibDB [5] using typical workloads (e.g., TPC-C [27]) evaluated in relevant systems [8, 37, 52]. Evaluation shows that:

- **Generality**. HoTEE is the first unified query system that supports general queries (equality, range, join, insert, update, and delete) among heterogeneous HE and TEE nodes within the same EDB instance (§5.1), while the first line of work supports only equality queries; HoTEE achieved logarithmic search complexity on all these queries, which is proved in §4.1 and empirically evaluated in §5.2.
- **Efficiency**. Although CryptDB supports general queries, HoTEE achieved up to 28% lower latency and 1.8X higher throughput than CryptDB (§5.1). Although HElibDB supports dynamic changing data on HE nodes, which CryptDB does not support, HoTEE achieved up to 93.8% lower latency and 15.2X higher throughput than HElibDB (§5.1). Meanwhile, HoTEE is the first work to support multi-party joint query on heterogeneous EDB containing both TEE and HE nodes, compared to all baselines.
- **Extensibility**. HoTEE supports a SQL (MySQL) instance consisting of both HE nodes and TEE nodes; and it supports another NoSQL (WiredTiger) instance consisting of both HE nodes and TEE nodes.

Our main contribution is MFT, a new confidential index abstraction to enable general queries with logarithmic search complexity in a unified EDB instance, containing both HE and TEE nodes. A long-lasting open problem in the EDB area is that, existing EDBs support only limited queries on either TEE or HE, HoTEE efficiently bridges heterogeneous EDB and is the first work that supports multi-party joint query in each EDB instance. Our other contributions include

[1]HoTEE stands for Homomorphic-TEE heterogeneous query.

a secure data aggregation protocol, which is the first effective attempt to enable multi-party data aggregation in each heterogeneous EDB, making HoTEE unique to facilitate multiple parties with distinct trust within the same EDB instance to securely share data (proven in §4.4) and make joint queries in various mission-critical applications (e.g., medical diagnosis [26, 28], governmental data collection [9, 53]). HoTEE's source code is released on github.com/osdip281/HoTEE.

2 Background

2.1 Encrypted Databases on the Cloud

Organizations are rapidly moving their sensitive data to cloud databases for scalability and rich query capabilities [1, 3, 53]. However, grave security concerns are hindering wide-scale adoption, particularly the need to maintain confidentiality against adversaries (e.g., privileged insiders and attackers that compromise database services) and detect integrity breaches in queries [35, 42, 46]. Two categories of research have emerged to address these concerns, as shown in Table 1.

The first category relies on *Trusted Execution Environment* (TEE), notably Intel SGX [14], which provides a secure area that ensures data confidentiality and computation integrity in it [39]. Typically, TEE databases [3, 39, 42, 42] utilize an off-the-shelf B^+-tree to perform searches on plaintexts decrypted within the TEE, enabling both point and range queries with logarithmic search complexity.

The second category uses *homomorphic encryption* (HE) for arbitrary and recoverable computation on encrypted data, particularly through fully HE algorithms like BGV [19]. To improve the searchability of HE data, CryptDB and its variants [36, 37, 45] use operation-specific encryption algorithms to recursively encrypt a data item, allowing for point searches with DET [10] and range searches with OPE [4]. Arx [36] employs Garbled Circuit for data comparison and search. HElibDB [5] uses FLT [18] to perform direct searches on HE data, albeit only supporting point searches via linear scanning of the entire dataset. While these techniques ensure IND-CPA [2], a strong security guarantee, they inevitably inherit the deficiency and functionality issues of specific cryptographic tools and also neglect query integrity.

Despite the two lines of work, many research efforts have been put on supporting multi-party joint queries on encrypted databases. HybrTC [51] is designed to enable verifiable select-and-join queries on TEE databases owned by multiple parties. Meanwhile, Dory [16] leverages multiple trust domains to establish an encrypted file store that cryptographically conceals database access patterns. These systems target either specific database types or oblivious database access, thus have an orthogonal goal with HoTEE.

2.2 Use Cases

Our motivation is to connect diverse encrypted databases (EDB), such that a client can submit distributed queries to multiple TEE and HE databases managed by different parties. In this paradigm, each database first filters out records that do not meet the condition specified in the query, then securely joins the remaining records on common attributes. Actually, such paradigm serves as a crucial building block for processing over 70% distributed plaintext queries [29, 49, 51]. Below we show two use cases of distributed encrypted queries supported by HoTEE, all encrypted fields are shaded.

System	Data Confidentiality	Query Integrity	Multi Party	Point Search	Range Search
MySQL [32]	✗	✗	✓	$O(\log n)$	$O(\log n)$
EnclaveDB [39]	Hardware	✓	✗	$O(\log n)$	$O(\log n)$
EdgelessDB [3]	Hardware	✓	✗	$O(\log n)$	$O(\log n)$
HybrTC [51]	Hardware	✓	✓	$O(n)$	$n/a.$
CryptDB [37]	IND-CPA°	✗	✗	$O(\log n)$	$O(\log n)$
HElibDB [5]	IND-CPA	✗	✗	$O(n)$	$n/a.$
Arx [36]	IND-CPA	✗	✗	$O(\log n)$	$O(\log n)$
Dory [16]	Irreversible	✓	✓△	$O(n)$	$n/a.$
HoTEE	Irreversible	✓	✓	$O(\log n)$	$O(\log n)$

°: Depending on the applied encryption algorithm (e.g., use HE for IND-CPA).

△: Potentially support, but not explicitly handle multi-party queries.

Table 1. Characterization of representative systems. **n/a**. means not support such query type. **Data Confidentiality** column shows varying levels of confidentiality guarantees, all sufficient.

Query-1 financial statistics [53]**.** This application involves the government querying total expenses incurred by each department and joining them based on transaction IDs, using data from EDBs belonging to different departments. While the government completely trusts each department's choice of TEE or HE, the departments cannot share databases with each other for taking a credit risk. HoTEE can enable this sharing with the following query.

```
SELECT SUM(amount) AS total_expenses, department
FROM departmental_encrypted_databases
INNER JOIN edb.transaction_id
WHERE transaction_amount BETWEEN '10M' AND '100M'
AND edb.category = 'expense'
GROUP BY department;
```

Query-2 medical diagnosis [28]**.** This application involves multiple hospitals that want to study the symptoms and prevalence of a new disease. Physicians in these hospitals need to query medical data from EDBs belonging to different hospitals. While the physicians trust each hospital's choice of TEE and HE, the hospitals cannot share their patients' medical data directly due to privacy concerns. HoTEE can facilitate this sharing with the following query.

```
SELECT COUNT(*) AS num_patients, symptom
FROM hospital_encrypted_databases
INNER JOIN edb.patient_id
WHERE patient_age BETWEEN '18' AND '80'
AND patient_blood_type = 'A+'
GROUP BY symptom;
```

2.3 Preliminaries

Realizing the proposed computing paradigm requires a principled query system to overcome two key challenges. Firstly, the system must securely dispatch, execute, and verify client queries in heterogeneous EDBs. Secondly, it must securely aggregate query results from multiple parties and produce decryptable records for the client without revealing the encryption keys of those parties. To address these challenges, we identify a tool with advanced security and functionality features as particularly promising for such paradigm.

Bloom filter [11]. It is a compact data structure designed for efficient membership testing. It operates by using a single set to store the hash values of all elements. Specifically, a Bloom filter B is an m-bit array that uses k independent and uniformly distributed cryptographic hash functions $\{H_k\}$, each of which maps an element to an index in B. To insert an element x into the filter, we compute its hash using $\{H_k\}$ and set all bits at the corresponding indices in B to 1. To test if an element y is a member of the set, we again hash it using $\{H_k\}$. If any of the bits at the corresponding indices in B are 0, then y is definitely not in the set (i.e., zero false negatives); otherwise, y has a high probability of being in the set (with small configurable false positives [11]).

Notably, while prior systems have utilized Bloom filters, their use limited to specific file stores or securing particular queries on TEE databases, as noted in [16, 51]. In contrast, HoTEE is focused on building a unified query system that can support multi-party joint queries on diverse EDBs.

3 Overview

3.1 System Model

Entities. Same as conventional (plaintext) databases [32, 38], HoTEE's participants has three types: *client*, *proxy*, and *database*. Multiple clients form a single party (e.g., physicians from one hospital, §2.2). Multiple parties outsource their encrypted data to database hosts on the public cloud, providing encrypted query service to the clients through the proxy.

Data model. To exemplify our design, we employ the relational data model, followed by a discussion on extensibility to encompass various types of databases and attributes.

Same as [39, 42], in HoTEE, an encrypted table T contains n confidential columns (or attributes) and has a primary key. Clients encrypt every value before sending it to the database. Concretely, in the case of a row (i.e., record) r, the value v_i in column c_i is represented as ciphertext $E(v_i)$ using an encryption algorithm E selected by distinct parties. For HE databases, BGV [19] is the preferred encryption algorithm, and the party *cannot* divulge its HE encryption key to databases or other parties. Conversely, AES-GCM [25] is ideal for TEE databases, and the party's encryption key *can* be securely provisioned to the TEE after attestation [14].

In HoTEE, all encrypted records are *versioned*. This involves clients maintaining the latest version of their own record locally, and keeping them synchronized with corresponding EDBs. In such a case, clients are assumed to have already retrieved the latest version of a record before submitting a new update query to that record. Dory [16], a prominent encrypted file store, has a similar idea on versioning for detecting breaches on file keyword (point) queries, while HoTEE targets general queries, including range queries, on general EDBs owned by multiple parties (§4.3).

HoTEE is highly extensible, which can be attributed to two significant factors. Firstly, HoTEE is built on an unclustered architecture [42], in which the database index and underlying storage are decoupled. Consequently, HoTEE's new secure index transparently supports both structured relational data model and unstructured key-value pairs (both evaluated in §5). Secondly, HoTEE transforms real values and strings into integers with standard IEEE 754 FPNs [40] and ASCII characters [21] respectively, allowing the encoding of diverse attributes for secure indexing in a unified manner.

3.2 Threat Model and Guarantees

In line with recent cloud-based EDBs [16, 51, 52], we assume that the all databases can be compromised by malicious adversaries $\mathcal{A}$, who can arbitrarily deviate from HoTEE's protocol to learn private data. Examples of $\mathcal{A}$ include database administrators and privileged insiders. $\mathcal{A}$ can execute a range of passive and active attacks, including extracting plaintext using cipher analysis [41] and pre-image attack [47], and manipulating query results (e.g., via modifying, replaying, and dropping). Nevertheless, $\mathcal{A}$ is computaionally infeasible to reverse one-way functions like SHA-256 [33] or compromise cryptographic encryption keys.

A client CL in HoTEE is only responsible for submitting a distributed query, processing lightweight protocol messages, and retrieving final results. We assume that CL is always honest and does not engage in collusion with any database. While access control [34] and denylist [50] are alternative methods to prevent corrupted clients, they are outside the scope of this work. The proxy PR is responsible for dispatching queries, verifying and aggregating final results. Staying in line with [42, 52], we assume that the database proxy follows HoTEE's protocol honestly but can be curious about client queries and plaintext results.

Same as [16, 37], we assume the underlying storage engine is maliciously secure. This guarantees that CL can always retrieve the correct data version from the underlying storage, by detecting rollback and fork attacks, which has been thoroughly investigated in prior storage works [12, 44, 52].

Out-of-scope attacks. Same as [3, 8, 37, 42], HoTEE does not handle metadata security, such as index access pattern and ciphertext ordering. To provide such orthogonal metadata protection, HoTEE can be integrated with existing advanced oblivious algorithms (e.g., Dory [16], ORAM [15, 51]). Also, coarse statistical properties like column names, index volume, or value length are not HoTEE's focus.

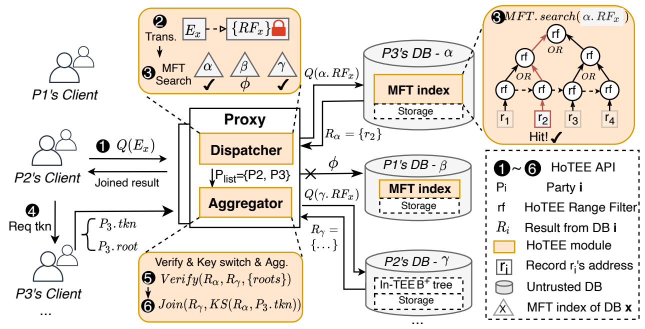


Figure 2. HoTEE's architecture and workflow.

HoTEE's guarantees. HoTEE's *confidentiality* guarantee ensures that all index keys are irreversible [33], which means that malicious adversaries are computationally infeasible to reverse the encrypted index keys and retrieve original sensitive data in plaintext. HoTEE's *integrity* guarantee ensures that any breaches on query results will be detected. These guarantees are facilitated by HoTEE's secure index, which effectively achieves high-performance encrypted queries with logarithmic complexity. Formally, we have the following theorem that captures HoTEE's security:

Theorem 1. *Given that the one-way function used in filter construction is mathematically irreversible, and assuming that ORE used in bias construction, homomorphic key switching utilized in aggregation, and MAC used in authentication are all semantically secure, and provided that the underlying storage is maliciously secure, the HoTEE protocol for multi-party joint query is one-way (i.e., mathematically irreversible) in the presence of a malicious adversary $\mathcal{A}$.*

We prove the above theorem and present corresponding complexity analysis in §4.4.

3.3 HoTEE's Workflow Overview

Figure 2 shows HoTEE's workflow with three phases.

• **Phase 1: query dispatch.** Same as plaintext distributed databases, clients send signed queries to the proxy via a TLS-enabled link (❶). The proxy will drop the connection if malformed queries are received, e.g., those with invalid signatures from unknown parties. Queries are encrypted via AES, which has been implemented efficiently in hardware.

The proxy runs a dispatching thread to determine the appropriate database to forward a query. To illustrate, in Figure 2, upon receiving a point query Q, the proxy decrypts the query parameter E_x and converts it into a secure index primitive called *rangefilter* (§4.2) in HoTEE (❷). This primitive, denoted as RF_x, acts as a cipher identifier for the original data X and is always encrypted (indicated by the red lock).

The proxy searches RF_x in pre-stored *Merkle rangeFilter Trees* (MFT) (introduced in **Phase 2**) for each database (❸) and forwards Q to database d if there exist relevant records (defined as records that match the query parameter) in d (concretely, search results in d's MFT is not ϕ). Meanwhile, the proxy generates a list of parties P_{list} to which d_i belongs and submits P_{list} to the aggregator thread (see **Phase 3**).

LibHoTEE API	Role	Description
❶ `newQry(cond)`	CL[1]	Submit a new encrypted query to HoTEE's proxy (§3.3).
❷ `transform(E(k))`	PR[2]	Transform a key k into HoTEE's searchable rangefilter (§4.2).
❸ `MFTSearch(rf)`	CL, DB[3]	R/W by searching on HoTEE's MFT index (§4.2).
❹ `reqTok(Ks)`	CL	Request new tokens from other clients for key switch (§4.3).
❺ `verify(r,root)`	PR	Run HoTEE's verification protocol to detect breaches (§4.3).
❻ `join(r,tkn)`	PR	Run HoTEE's key switch protocol, then aggregate results (§4.3).

[1] **CL**: CLient [2] **PR**: PRoxy [3] **DB**: Encrypted DataBase

Table 2. The API provided by HoTEE.

• **Phase 2: local search.** When dealing with transformed parameters in TEE databases, the workflow remains unchanged. This involves decrypting the parameters within TEE and searching plaintexts in a TEE-shielding B$^+$-tree.

However, for HE databases, a novel approach is employed where HoTEE searches RF_x (e.g., $\alpha.RF_x$ in database α) on the MFT index. Unlike previous work that relies on specific cryptographic algorithms to support designated queries with high-complexity (§2.1), MFT enhances the searchability of HE data by recursively stacking all records' rangefilters into a merkle-tree style structure, where each inner node's value is the *OR* gate computation result of its child nodes. MFT supports encrypted point and range queries on HE data, bridging the functionality gap between TEE and HE databases (§4.2).

• **Phase 3: data aggregate.** During **Phase 3**, which runs in parallel with **Phase 2**, the proxy requests *key switch* tokens from the parties listed in P_{list} (❹). Upon receiving the query results, the proxy conducts an integrity check using two rules (❺). The first rule, correctness, requires that the results match the query parameters or fall within the specified boundary, and that the re-computed root rangefilter aligns with that of the data owner party. The second rule, freshness, mandates that the results are based on the most recent version, verified by checking the MAC on rangefilters (§4.2). Once the results pass the integrity check, the proxy switches the keys of the results from different parties using the requested tokens (❻), thereby enabling the client to decrypt and utilize them. Finally, the key-switched results are joined and returned to the client (§4.3).

Overall, the highlight of HoTEE lies in its ability to provide provable guarantees of confidentiality and integrity, along with logarithmic search complexity, for general multi-party joint queries on malicious EDBs. This is achieved through two key aspects. Firstly, HoTEE employs a novel approach of constructing a new secure index that efficiently handles general queries while bypassing inherent deficiencies and limited capabilities of prior approaches. Secondly, the new secure index enables effective verification of query results in a single data structure, ensuring that HoTEE simultaneously achieves all desired security and efficiency properties.

4 Protocol Description

This section first describes a strawman index design inspired by bloom filter (§4.1) and then introduces HoTEE's new MFT index based on the strawman index (§4.2). Next we present *SDA*, a secure data aggregation protocol that locates query-dependent parties using MFT, transforms and aggregates multi-party data into querier-decryptable data (§4.3).

4.1 A Preliminary Subfilter-based Index

Preliminaries. A bloom filter [11] is a space-efficient data structure that provides membership information: all elements are added to a single set and it checks whether an element is a member of a set. It also has an intrinsic *confidential* characteristic: the elements themselves are not added to a set, instead the hash is added to the set. Formally, a bloom filter B is an array of m bits and parameterized by k independently uniform hash functions $\{H_k\}$, where $\{H_k\}$ map an element to k positions in B uniformly, and all elements are mapped to a single array in a *monolithic* manner.

Traditional monolithic bloom filter supports confidential membership checks. Specifically, to add an element x to B before checking, one uses the hash functions in $\{H_k\}$ to hash x and sets all bits at the position of hash values to 1. Then, to check whether x is in the set, one also hashes x using the k hash functions. If any bit at the position of hash values is 0, x is definitely not in this set (none false negatives); otherwise, y is in the set (with small configurable false positives).

Overall, the traditional monolithic bloom filter supports membership checks using bit representation and inherits the confidentiality as long as we make standard assumptions on the one-wayness of cryptographic hash functions [20],

Subfilter-based search index. Owing to the monolithic design, a traditional bloom filter supports only membership check by telling whether an element (hash) is in a set. Nevertheless, it cannot serve as an index for locating data in storage.

We observe that the confidential characteristic of bloom filters can be useful for encrypted search, thus in our initial design, we divide a monolithic bloom filter into *subfilters* where each subfilter stores the hash (in bits) for only *one* element. As shown in Figure 3, subfilters have a constant array size and are recursively stacked in a Merkle-tree style for encrypted search. Specifically, each leaf node in the subfilter tree stores one subfilter of an index key (e.g., "1001" for index key *2*) and is sorted in plaintext key order; for each inner node (all nodes except the last layer), its value is the element-wise *OR* gate computation result of its child nodes' subfilters: given an m length subfilter, for an inner node *inner* with k child nodes cn_k, the inner node's subfilter is:

$$inner[i] \; = \; cn_1[i] \; or \; \dots \; cn_t[i] \; or \; \dots \; cn_k[i] \; (\forall i \in [0, m]) \quad (1)$$

Same as B$^+$-tree, only the leaf nodes store values (e.g., record pointers), the inner nodes are only for indexing (tree traversal).

The design rationale behind subfilter tree is that, if a "1" bit is present in a child node's subfilter, after *OR* computation with other child nodes' subfilters, its parent node's subfilter must contain that "1" bit (e.g., if $cn_t[0]$ ="1", it always holds $inner[0]$ ="1"). This implies HoTEE's new insight of using *existence* information for searching: if one element exists in a subfilter node (i.e., hashes of the element are set to "1"), the element must exist in its parent node's subfilter. This enables HoTEE to traverse a subfilter tree in a top-down manner by verifying the existence of query parameters in subfilter nodes with confidentiality, rather than comparing the plaintext values as in traditional B$^+$-tree.

Based on the subfilter tree, to conduct an equality search, instead of searching with plaintext query parameters (for short, parameter), the client generates an encrypted subfilter of the parameter using Equation 1, then the database traverses the subfilter tree to locate the corresponding record of the parameter (Figure 3). Specifically, each inner node conducts *subfilter comparison* to verify whether every "1" bit of the parameter exists in the current node's subfilter. If so, the parameter will be passed to the child nodes iteratively until reaching the bottom layer. When reaching the leaf nodes, different from the verification in inner nodes, a leaf node must verify that every single bit in its subfilter matches the parameter as one subfilter comprises only one index key.

The equality search strategy can be extended to support range search. Given two range parameters (i.e., left and right boundaries), they are first transformed into two subfilters using Equation 1, then the subfilter tree searches two subfilters same as the equality search and locates two leaf nodes (if matched). Last, since leaf nodes are sorted during index construction, all leaf nodes between the two searched leaf nodes are returned as searched results same as B$^+$-tree.

4.2 HoTEE's Complete Confidential Index

Limitation of strawman subfilter index. Although the subfilter tree (§4.1) can support equality and range query with logarithmic search complexity, however, since the subfilter tree is driven by existence verification rather than numerical comparison as in traditional B$^+$-tree and there is no *order* among subfilters, two fundamental limitations (**L1, L2**) arise.

First (**L1**), a subfilter tree cannot support non-existent key search. Apparently, range queries can search on parameters that are non-existent in databases as clients can choose whatever search boundaries they want. This is not a problem in traditional B$^+$-tree because index nodes and query parameters are plaintexts, thus the position of non-existent keys can be trivially located by finding two adjacent plaintexts. However, since subfilter is encrypted and there are no numerical

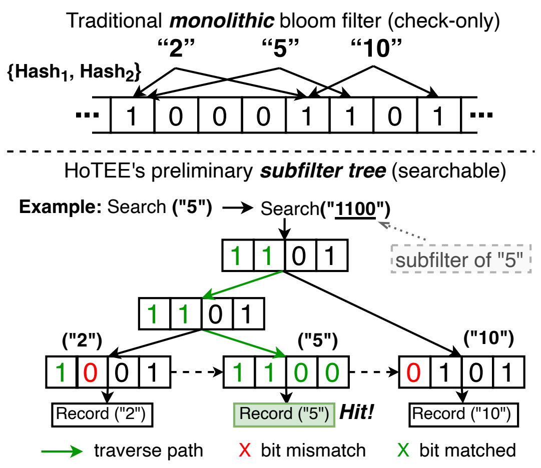


Figure 3. An initial search index built on top of subfilters.

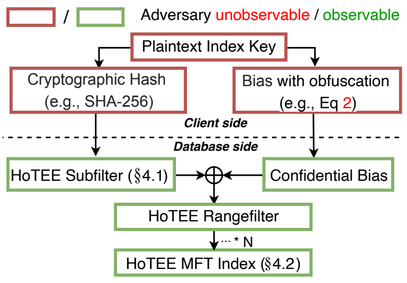


Figure 4. The routine to construct HoTEE's rangefilter primitive.

orders between subfilters, a subfilter tree cannot locate positions of non-existent keys through subfilter comparison. Second (**L2**), owing to the same reason, a subfilter tree does not support write operations (e.g., insert), because the tree cannot locate the write position among sorted leaf nodes.

Design rationale. To tackle **L1** and **L2**, our key idea is that, instead of handling specific query operations using special encryption algorithms as in previous work [5, 31, 37], HoTEE embeds the query operations into a secure data structure, and the data structure is built on traditional encryption schemes that provide with confidentiality guarantees.

Specifically, we identify *determinism* and *order*, two essential properties for equality and range search that must be retained in one searchable data structure. Recall that HoTEE's initial subfilter tree already constructs a searchable data structure with determinism by creating subfilters with deterministic plaintext-ciphertext maps, and confidentiality that directly inherits from cryptographic hash functions in bloom filters. Thus, adding orders to subfilters without sacrificing confidentiality is on the critical path of a secure index design.

Rangefilter primitive. To this end, we propose *rangefilter*, a primitive for constructing a confidential search index by tackling two limitations mentioned above. The routine to construct rangefilter is shown in Figure 4. Specifically, we generate a *confidential bias* for each subfilter with a bias function that fulfills three rules. First, the function should be monolithic, ensuring that a larger plaintext always maps to a larger bias to keep encrypted data ordered. Second, to reduce the overhead on the client side, the function should be stateless, which means that no additional states (e.g., previous biases) should be maintained at runtime, as this generation process happens on the client side. Last, the function should provide randomness such that an adversary that observes a bias should not immediately deduce the plaintext.

Definition 1 (Confidential bias function). Given two plaintexts $p_1 > p_2$, if a bias function f satisfies: (1) $f(p_1) > f(p_2)$, (2) f is independent of any previous states, and (3) f varies within a certain range of values, then f is regarded as a qualified confidential bias function for rangefilters.

We formulate that a bias function f should have the form of $f(x)=g(x)+t(x,y,z)$, where $g(x)$ is monotonically increasing on x, $t(x,y,z)$ is a parametric function which varies within a certain range and has a fixed number of values, and does not overwhelm the monotonicity of $g(x)$. Any functions that fulfill the above formulation can be applied to generate biases for subfilters. In HoTEE, we use the bias function

$$f(x) = kx + hash(x),\ hash(x) \in [0, k) \qquad (2)$$

in which x is the plaintext, k is a confidential coin, both are owned by the client. By applying bias b to a m length subfilter (i.e., m bits), a rangefilter has a *biased range* of $[b+\gamma, m+b+\gamma]$, in which γ is a random parameter generated by the client to obfuscate adversaries.

Definition 2 (HoTEE rangefilter). A rangefilter is a secure data structure that comprises a subfilter (Equation 1) and a biased range (Equation 2), providing data existence and order information for encrypted equality and range search.

To remark, we emphasize that the entire process of generating rangefilters at the client side is stateless because (1) the bias function is stateless (Equation 2), and (2) generating subfilters requires only storing parameters of a fixed set of hash functions. Overall, unlike previous work that requires clients to store stale states or even a full replica of outsourced data [36], HoTEE supports lightweight clients without paying additional storage overhead.

MFT confidential index. Based on rangefilter, we construct Merkle Rangefilter Tree (MFT), a confidential index for supporting general queries on encrypted data with logarithmic search complexity. We present the construction and search operations on MFT, and use MFT for query authentication.

• **MFT construction.** To generate rangefilters, the client takes plaintexts of index keys as input and outputs rangefilters by generating subfilters and biases respectively (**Algo** 1). Specifically, the plaintexts are sorted to ensure the encrypted data preserve the same numerical order as plaintexts (line 9). Then, subfilters are constructed using client-chosen (unknown to the database) cryptographic hash functions (e.g., SHA-256) with Equation 1 (line 10~14). Last, biases are generated using Equation 2, and are added with obfuscation

Algorithm 1: MFT Index Preparation (Client side)

```
1  Input:
2      plaintext[]                          ▷ Plaintext keys for indexing
3      chashfunc[]          ▷ Cryptographic hash functions for filters
4      k, $\gamma$                                ▷ Obfuscation parameters
5  Output:
6      rf[]          ▷ Rangefilters of plaintexts for encrypted search
7  Function Prepare() do
8      $sortedPtx[] \leftarrow$ sort(plaintext)
       ▽ Generate subfilters
9      foreach plaintext p in sortedPtx do
10         $subfilter[m] \leftarrow \{0\}$          ▷ m is subfilter's bit length
11         foreach evaluator e in chashfunc do
12             subfilter.set(e(p), 1)
13         rf.insert(subfilter)
       ▽ Generate biases in plaintext order
14     $range_{min} \leftarrow \gamma$, $range_{max} \leftarrow m + \gamma$
15     $start \leftarrow 0$
16     foreach plaintext p in sortedPtx do
17         $bias \leftarrow k * p + hash(p)$              ▷ $hash(p) \in [0, k)$
18         $p.rmin \leftarrow bias + range_{min}$
19         $p.rmax \leftarrow bias + range_{max}$
20         $rf[start++].rng \leftarrow \{p.rmin, p.rmax\}$
21     return rf
```

parameters to produce biased ranges of subfilters, which contain order information for searching (line 15~20).

With rangefilters, the client constructs MFT before uploading it index to the cloud database, as shown in **Algo** 3 in the appendix. Notably, each MFT inner node's range is constructed as the union of its child nodes' ranges for traversal (introduced in MFT search operations below).

MFT inherits two advantages of a traditional B^+-tree. First, all MFT leaf nodes are sorted in plaintext order and linked for fast data fetching. Second, only MFT leaf nodes contain values (i.e., storage pointers of records), and other inner nodes are only for indexing. Nevertheless, different from traditional B^+-tree, MFT is constructed by recursively stacking rangefilters (line 14~31), which display as an encrypted form of index keys including HoTEE's subfilters and biased ranges.

• **MFT search operations.** In MFT, the high-level idea of encrypted search is to use the biased range (for short, range) for indexing (tree traversal) among inner nodes as the biased range preserves the numerical orders of encrypted data as in plaintexts, and use the subfilter for point match among leaf nodes as it contains the existence information of plaintexts (**Algo** 2). Since most read/write queries are composed of equality and range operations [30], we describe how HoTEE's MFT index handles encrypted equality and range search below.

To handle encrypted equality search, MFT compares parameter's range with each tree node's range until the bottom layer. For each comparison, if the target's range is within the range of the tree node, it implies that the tree node is on the target's search path, because a parent node's range is the union computation result of its child nodes' ranges. When reaching the bottom layer, HoTEE compares subfilters to see if every bit in parameter's subfilter matches a leaf node's subfilter, and only when every single bit matches, the final result is searched (line 5~14). This is because, different

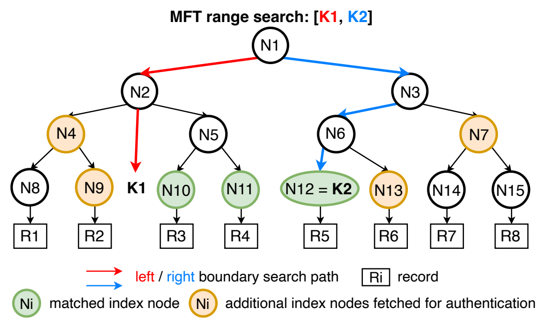

MFT range search: [K1, K2]
N1
N2
N3
N4
N5
N6
N7
N8
N9
K1
N10
N11
N12 = K2
N13
N14
N15
R1
R2
R3
R4
R5
R6
R7
R8
left / right boundary search path
Ri record
Ni matched index node
Ni additional index nodes fetched for authentication

Figure 5. A typical MFT range search example on HoTEE.

from a monolithic bloom filter design, HoTEE maintains one-to-one maps between subfilters and plaintexts (§4.1).

To handle encrypted range search, MFT conducts two equality searches on the left and right boundary parameters, and outputs all leaf nodes between two searched leaf nodes (line 25~32). Besides, MFT tackles non-existent key search for parameters as shown in Figure 5. Concretely, when the left boundary is non-existent, i.e., the range of K_1 is within the range of node N_2 but none of N_2's child nodes' ranges fully contain K_1's range (owing to the monotonicity in Equation 2), MFT searches the smallest data larger than the left boundary and confirms N_{10} as the first searched index node (line 15~19). Similarly, if the right boundary is non-existent, MFT searches the largest data smaller than the right boundary (line 20~24). In Figure 5's case, the right boundary is existent because MFT conducts subfilter comparison at leaf nodes and finds out that N_{12}'s subfilter exactly matches K_2's. Last, all leaf nodes' records in green box (R_3~R_5) are returned.

• **MFT query authentication.** As aforementioned, MFT's recursive Merkle-tree style structure enables general range queries even when query parameters are non-existent, and such structure also allows clients to efficiently authenticate queries by checking the absence and presence of data records in an outsourced database: any manipulation or omission of the query results will fail to regenerate the root subfilter, and thus be detected. To authenticate encrypted range queries, HoTEE takes the following two steps.

First (**Proof fetching**), on receiving a range query, the database searches for all records whose key falls within the search range (**Algo** 2) and constructs a proof including (1) one encrypted record to the immediate left of the lower-bound (R_l) and one encrypted record to the immediate right of the upper-bound of query result (R_r), and (2) additional subfilters to assist recompute the root's subfilter, i.e. subfilters of all left sibling nodes of MFT left traversal path and right sibling nodes of the right MFT traversal path.

Second (**Proof checking**), on receiving the proof and the query result, the client ensures no manipulation or omission of query result by checking two rules. First, R_l's range is smaller than the left boundary's range, and R_r's range is larger than the right boundary's range. Second, the client recomputes the root subfilter in a bottom-up manner based

Algorithm 2: MFT Confidential Search (Database side)

```
Input:
  root                              ▷ Entry of MFT from Algo 3
  key (i.e., rangefilter)           ▷ Search target
  opt (P | L | R)    ▷ Search option: Point search or range
   search with Left or Right boundary; default is P
Function EqualitySearch(root, key, opt=p) do
    ▽ Compare subfilters (§4.1) at leaf nodes
    if $isLeaf(root)$ then
        ▽ Every subfilter bit must match
        if $key.subfilter = root.subfilter$ then
            return root
        else
            return null
    ▽ Use biased range to traverse MFT
    if $key.rng \in root.left.rng$ then
        return EqualitySearch(root.left, key)
    if $key.rng \in root.right.rng$ then
        return EqualitySearch(root.right, key)
    ▽ If left boundary is non-existent (Fig 5)
    if $opt = l$ then
        ▽ Find smallest data larger than key
        $root \leftarrow root.right$
        while $!isLeaf(root)$ do
            $root \leftarrow root.left$
        return root
    ▽ If right boundary is non-existent
    if $opt = r$ then
        ▽ Find largest data smaller than key
        $root \leftarrow root.left$
        while $!isLeaf(root)$ do
            $root \leftarrow root.right$
        return root
Function RangeSearch(root, [key1, key2]) do
    $value[] \leftarrow \emptyset$
    $left \leftarrow$ EqualitySearch(root, key1, l)
    $right \leftarrow$ EqualitySearch(root, key2, r)
    ▽ Traverse linked leaves in order
    while $left.next\ != right$ do
        value.insert(left.next)
        $left \leftarrow left.next$
    return value
```

on all the query results and all additional sibling subfilters, and compares the computed root subfilter with the one it owned. If two subfilters are the same, the query result is authenticated successfully.

Figure 5 shows a running example. Given a range query that searches for records in $[K_1,K_2]$, it returns $\{N_{10}, N_{11}, N_{12}\}$ as the query result. For authentication, MFT returns $N_9{\sim}N_{13}$: N_9 and N_{13} are returned to prove that no records in range $[K_1, N_{10})$ and $(N_{12}, K_2]$ are omitted. Two additional MFT nodes (N_4 and N_7) are also returned to reconstruct the root subfilter and compare it to the local root subfilter for authentication.

Note that, we omit the discussion of authenticating equality queries here, because it takes the same theory of authentication and uses similar steps as range queries.

4.3 A Unified Query System Leveraging MFT

With MFT confidential index, HoTEE runs hybrid-trust multi-party joint query in three phases (Figure 2).

Phase 1: query dispatch. Before submitting queries, multiple parties first construct MFTs on their chosen index keys of data records (records are encrypted by either TEE using AES-GCM [25] or HE using BGV [19] based on independent trusts). Note that, MFT is constructed by attribute, multiple attributes can build multiple MFTs. Given the semi-honest assumption of clients (§3.2), a party cannot directly share its encryption keys for building MFTs with other parties, including the hash functions for generating subfilters (§4.1) and obfuscation parameters for generating biased ranges (§4.2). However, without knowing multiple parties' encryption keys, it is infeasible for a client (in one party) to construct query parameters (on index keys) that can be dispatched to query-dependent databases of multiple parties and then search in dispatched dependent databases.

To remedy this issue, our key observation is that even if multiple parties have distinct trusts of TEE or HE on protecting data records, query parameters are commonly less sensitive. Hence, it is secure to set up TEE enclaves (attested by all parties) at the proxy to store the encryption keys of parties, transform parameters and locate query-dependent databases without revealing multiple parties' encryption keys.

Specifically, a client sends a new query encrypted with AES to the proxy (❶), the proxy decrypts that query with AES decryption key in enclave, and transforms the parameters into rangefilters using each party's MFT encryption keys in an enclave (❷). By searching transformed rangefilters on MFTs, the proxy learns which databases contain the searched content of the query same as existing plaintext databases, while the proxy learns nothing about the plaintexts owing to the secure execution in an enclave. After searching, the proxy forms a party list P_{list} containing all related parties that the query depends on (for multi-party data aggregation), and dispatches the transformed query to dependent databases.

Phase 2: local search. Upon receiving queries from the proxy, TEE databases decrypt query parameters with pre-stored keys in enclave and search with a TEE-shielding B$^+$-tree where each tree node's value is plaintext; HE databases directly search on MFT with transformed parameters (❺).

Phase 3: data aggregate. Before sending searched query results (i.e., data records) to the client, a subtle case is that, since the records are retrieved from multiple parties, a client cannot decrypt all records because it knows only its party's decryption key. A trivial solution is to decrypt all retrieved records in the proxy enclave, and re-encrypt them with the client's pre-stored key, such that the client can decrypt data and the plaintext records are invisible to the semi-honest proxy. However, this solution does not work in a hybrid-trust scenario where records contain HE data, which cannot be decrypted in a TEE owing to the different trusts (§3.2).

Aggregating multi-party data with SDA. We propose *SDA*, a protocol that securely aggregates multi-party data without decrypting thus violating HE's confidentiality, by leveraging two key weapons: the MFT confidential index and *key* *switch* [19], a homomorphic algorithm that can switch the encryption key of HE data to a new key without decrypting it (we describe the detail of key switch syntax in Appendix A.5). At a high level, SDA locates query-dependent parties by searching on MFT confidential index and produces P_{list} at the proxy in **Phase 1**, the proxy sends P_{list} to the client and the client requests *tokens* from parties in P_{list} (❸), finally the retrieved records are transformed by running key switch with *tokens* at the proxy and sent back to the client (❹).

HoTEE's SDA protocol runs with three steps. First, HoTEE transforms query parameters and searches on MFTs at the proxy, locates databases that contain the matching data and generates a party list P_i that these databases belong to (§4.2). Second, the proxy signs P_i and returns P_i to the client, the client verifies the signature and sends i disposable encryption keys S_i to each party (❸). Upon receiving S_i, each party runs KS.Enc(S, S_i) to produce a secret token tkn_i by encrypting its encryption key with S_i, and sends tkn_i to the proxy enclave for aggregation later. Last, the proxy transforms each retrieved record from party P_i (i.e., ctx_i) with the party's token (i.e., tkn_i) by running KS.Switch(ctx_i, tkn_i) in enclave (❹), and sends transformed results to the client.

4.4 Complexity and Security Analysis

Complexity analysis. Same as a traditional B^+-tree or other binary search trees, the time complexity of traversing inner nodes is $O(\log n)$, where n is the total number of MFT nodes; the time complexity of verifying subfilters at leaf nodes is $O(m)$, where m is the length of HoTEE's subfilter. Therefore, the time complexity of basic operations that traverse inner nodes and verify leaf nodes such as equality query, range query, insert, and delete is $O(\log n)$+$O(m)$. Since $m \ll n$, the actual time complexity for all basic operations is $O(\log n)$; the time complexity of other operations that combines basic operations including join and update is $O(\log n)$.

Security analysis. We give a two-step proof sketch of HoTEE's security guarantee by analyzing two core components created by HoTEE: the MFT index and SDA protocol.

• **Step 1: MFT index's confidentiality.** Same as existing EDBs [3, 37], we analyze the index confidentiality under chosen-plaintext attacks (CPA), a well-known attack that tries to infer plaintexts by using adversary-observable variables to deduce the black-box encryption algorithm.

Specifically, we illustrate MFT's confidentiality on data records and index keys separately. First, for data records, same as existing EDBs [37, 42], their confidentiality directly inherits from TEE's or HE's confidentiality guarantee, which is secure against CPA. Second, for the index keys on the database, a CPA attacker may use the observed rangefilter including subfilter and biased range to deduce plaintexts: for subfilters, it is impossible for the attacker to infer the plaintexts as we make standard assumptions on the one-wayness of cryptographic hash functions; for the biased range, even though the attacker may deduce the bias function owing to its linearity and monotonicity (formulated in §4.2), however, the attacker cannot infer the plaintext as well because the biased range is obfuscated with a client-chosen random parameter γ.

• **Step 2: SDA protocol's confidentiality.** The security of HoTEE's SDA protocol directly inherits the cryptographic guarantee of the key switch algorithm [19], a homomorphic algorithm that was used for reducing HE data's noises.

Overall, we prove that HoTEE provides the same strong confidentiality guarantee as prior work from the aspects of two core components: HoTEE's MFT index provides the same confidentiality against CPA attacks (**Step 1**) and the SDA protocol directly inherits the used HE algorithm's cryptographic guarantee (**Step 2**).

5 Evaluation

Testbed. All experiments were done on our cluster machines, each equipped with a 2.60GHz Intel E5-2690 V3 CPU, 64GB memory, 40Gbps NIC, and 24 cores. Each node, including clients, databases, and the proxy, was executed in a docker container. The average ping latency between the nodes was set at 0.17ms with the aid of Linux traffic control [6].

Baseline. We constructed two instances, one using SQL (MySQL [32]) and the other using NoSQL (WiredTiger [13]). Both instances incorporated both HE and TEE databases. We compared HoTEE with three baselines: CryptDB [37], HElibDB [5], and vanilla insecure databases (i.e., MySQL and WiredTiger). MySQL is commonly used as the SQL backend for EDBs; WiredTiger is the default NoSQL KV backend for MongoDB. CryptDB and HElibDB are imperative HE databases. CryptDB enables general queries on HE data using deterministic encryption (DET) [10] for point queries, and order-preserving encryption (OPE) [4] for range queries with separate indexes. HElibDB relies on Fermat's Little Theorem [18] to allow directly evaluate the equivalence of HE data but requires scanning the entire dataset.

For a comparable analysis, we expanded upon baselines in three ways. Firstly, we replaced CryptDB's HE algorithm from partial HE to fully HE BGV [19], to ensure the same computational functionality as HoTEE. Secondly, all baselines were extensively implemented on both SQL and NoSQL instances. Thirdly, we integrated HoTEE's core proxy components (refer to Figure 2) in CryptDB and HElibDB to support distributed queries, as in HoTEE.

We run all baselines with the cutting-edge TEE database, Azure EdgelessDB [3], which equips an SGX-shielded B+-tree with state-of-the-art index performance. Notably, we did not compare our results to other EDBs because they

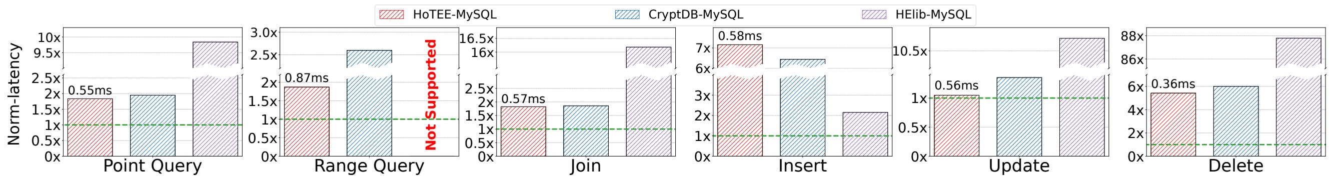


Figure 6. Normalized end-to-end latency of running TPC-C on all baselines in SQL instances. All systems' latency results are normalized to vanilla insecure MySQL (in green dashed line). Values on each red bar indicates HoTEE's averaged latency.

Distributed query latency (in milliseconds)

Systems (in SQL instance)	HoTEE		CryptDB	
Query type	PQ	RQ	PQ	RQ
P1 (*proxy*): Parameter transform	0.11	0.13	0.19	0.32
P2 (*proxy*): Dispatch search	0.12	0.18	0.06	0.1
P3 (*database*): Record search	4e-4	1e-3	2e-4	1e-3
P4 (*proxy*): Query authenticate	1e-3	1e-3	N/S	N/S
P5 (*proxy*): Aggregate & Key switch	0.17	0.21	0.12	0.24
P6 (*client*): Result decrypt	0.15	0.35	0.21	0.52
End-to-end ($\sum P_i$)	0.55	0.87	0.58	1.18

Table 3. Breakdown and comparison of query latency. **PQ** and **RQ** means point and range query respectively. **P1 (*prox.*)** means the first phase of parameter transformation occurs in the proxy, etc.

either lack support for HE queries (e.g., HybrTC [51]) or focus solely on specific file stores (e.g., Dory [16]).

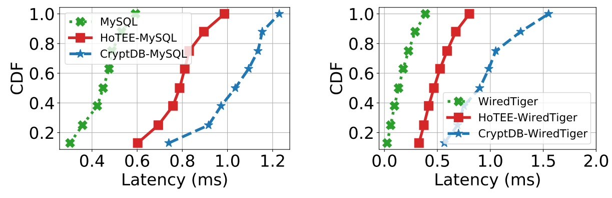
Workload and default setting. Since there is no standard benchmark for evaluating query performance on heterogeneous EDBs, for a fair comparison, we evaluated workloads from recent EDBs [3, 37, 52, 56], which includes the TPC-C benchmark [27] for relational databases and a customized workload for NoSQL key-value store.

Specifically, we followed the multi-party deployment approach [51] by first partitioning TPC-C randomly and uniformly across databases based on warehouse ID, and assigned clients to parties based on their associated warehouse IDs. For customized NoSQL workloads, we simulated financial statistics and medical diagnosis workload (as described in §2.2) by generating 2^{10} fixed-size keys (8 bytes) and values (64 bytes) in a manner similar to [54], and horizontally partitioned all key-value pairs across databases as in TPC-C.

To match the setup of existing multi-party query systems [16, 51], we equipped each party with two databases. Unless for scalability experiments, we simulated two parties: one utilized TEE databases while the other utilized HE databases. All workloads were encrypted depending on the EDB type. In the homomorphic setting, we employed HElib's BGV implementation [5], using a plaintext prime modulus of 29 and a ciphertext modulus of 1009 to compress ciphertexts with an expansion coefficient of 4.

We ran each experiment for 60 seconds and collected the result in the middle 30s (i.e., 15s~45s) to avoid the disturbance caused by system start-up and cool-down. Our evaluation focuses on the following questions:

§5.1 How efficient is HoTEE compared to baselines?
§5.2 Can HoTEE scale to more parties and larger datasets?
§5.3 How sensitive is HoTEE on heterogeneity?
§5.4 Is HoTEE robust to integrity breaches?



(a) Run in a SQL instance (b) Run in a NoSQL instance

Figure 7. Distributions of end-to-end range query latency running TPC-C and customized key-value workload on all baselines.

5.1 End-to-end Performance

We first evaluated the performance of HoTEE and baselines in a fault-free scenario where there were no breaches in the query results. The results presented in Figure 6 indicate that HoTEE demonstrated a lower latency (between 5.3x to 16.1x, excluding *insert* queries) compared to HElibDB. This is attributed to the use of MFT index by HoTEE, which provides logarithmic search complexity as opposed to HElibDB that requires linear scans to search for equivalent encrypted data, leading to reduced latency for all *read* queries.

Priority: read query latency. HElibDB achieved lower *insert* latency (between 3x to 3.3x) compared to HoTEE and CryptDB. This is attributed to the fact that HElibDB simply appends new records to its storage by trading off the linear search latency on *read* queries. In contrast, HoTEE and CryptDB both maintain sorted records in storage, leading to higher *insert* latency by first looking up a position through indexing before writing a new record to that position.

It is worth noting that in the multi-party joint query scenario of HoTEE, *read* could occur more frequently (refer to §2.2), and hence, we deem it acceptable to make such a performance trade-off, which has been a common practice in existing insecure databases [13, 32].

General functionality and efficiency. Figure 6 shows that HoTEE had 6% to 28% lower query latency than CryptDB. CryptDB builds separate tree-based indexes for different types of queries, supporting general query functionalities with logarithmic search complexity. Despite the same search complexity in theory, HoTEE still outperformed CryptDB in practice in terms of both query latency (verified in Table 3) and throughput (shown in Figure 8).

Table 3 verified the efficiency of HoTEE's MFT index over baseline. Specifically, CryptDB's major overhead was observed to be in **P1** and **P6** for cryptographic operations. In contrast, HoTEE incurred additional latency in **P2** for linear comparison of subfilters at MFT's leaf nodes, and **P4** for

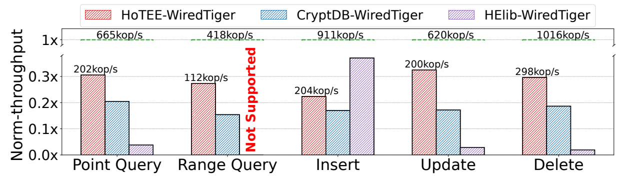
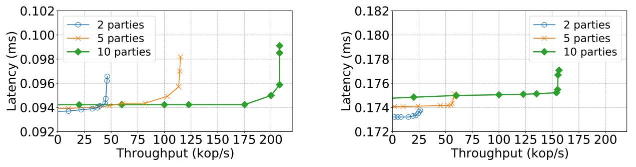
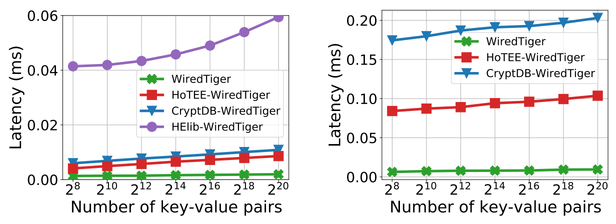


Figure 8. Normalized throughput in NoSQL instances. The green dashed line is vanilla (insecure) WiredTiger's averaged throughput. Values on red bars indicate HoTEE's averaged throughput.



(a) Point Query (b) Range Query

Figure 9. Performance of HoTEE with varying numbers of parties.



(a) Point Query (b) Range Query

Figure 10. Performance comparison with varying dataset sizes.

query authentication. Nonetheless, the overhead of CryptDB outweighed that of HoTEE.

We then evaluated the latency distribution by running range queries on all baselines in various workloads and databases. As shown in Figure 7, HoTEE significantly reduced range query latency for all workloads in comparison to CryptDB, consistent with the results in Figure 6. The latency distributions were influenced by the length of range queries, whereby longer queries resulted in more queried results, leading to higher latency of aggregation, key switch, and decryption. Notably, insecure databases do not require key switches and decryption. HElibDB was not evaluated since it does not support range queries.

High throughput on mixed queries. For workload with mixed point and range queries, as shown in Figure 8, HoTEE achieved 1.31x to 1.88x higher throughput than CryptDB on NoSQL instances. This can be attributed to the fact that CryptDB's design requires frequent shuffling between DET index and OPE index when point and range queries are mixed, whereas HoTEE's MFT directly supports both types of queries in a single secure data structure, thus avoiding CryptDB's index shuffling overhead. Besides, we further optimized the throughput by employing the batching strategy [5] on homomorphic encryption/decryption and key switching (refer to Appendix A.6 for more details).

In summary, HoTEE supports general queries with low latency and high throughput. HoTEE favors *read* query performance and tolerates moderate write (*insert*) performance downgrading. HoTEE is most suitable for diverse applications that desire high *read* performance and strong security, such as financial statistics [53] and medical diagnosis [28].

5.2 Scalability

Next we tested how HoTEE, a multi-party joint query system, scales to more parties and larger datasets.

Scale to more parties. We let each party equip one TEE node and one HE node, and we varied the number of parties up to a maximum of 10, which is considered sufficient for collaborative queries in real-world scenarios [51]. Figure 9(a) and Figure 9(b) depict HoTEE's throughput-latency performance for typical queries. With the number of parties increased, both HoTEE's throughput and tail latency increased. Note that the latency gap between different numbers of parties on range queries was larger compared to point queries because range queries typically fetch more data, leading to increased query processing time.

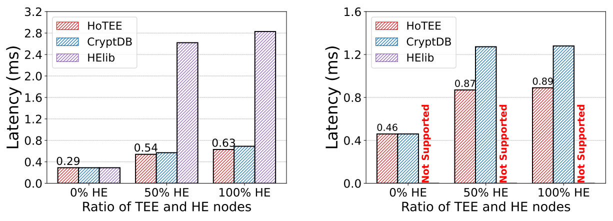
Scale to larger datasets. Then we evaluated baselines with varying sizes of the key-value datasets. Figure 10(a) and Figure 10(b) demonstrate that HoTEE's latency scales logarithmically, which confirms our complexity analysis in §4.4.

Regarding point queries, both HoTEE and CryptDB incurred moderate overhead compared to insecure WiredTiger, while HElibDB experienced significantly higher latency due to its requirement to linearly scan the whole dataset. Concerning range queries, HoTEE achieved lower latency than CryptDB (in comparison to point queries), which can be attributed to the MFT index used by HoTEE, as it takes less time to transform queries than the cryptographic approach used by CryptDB (proven in the breakdown Table 3). We did not compare HElibDB in Figure 10(b) because it does not support range queries.

5.3 Sensitivity Study on Heterogeneity

To evaluate HoTEE's sensitivity, we varied the TEE/HE node ratio and ran typical queries for each ratio. Figure 11(a) demonstrates that 100% TEE node settings resulted in significantly lower latency than other ratios. Furthermore, compared to range query performance in Figure 11(b), the latency gap between pure TEE (100% TEE nodes) settings and other ratios for point queries in Figure 11(a) is smaller due to the more efficient execution of point queries in HoTEE, as also demonstrated in Figure 9 and Figure 11.

Our sensitivity study demonstrates a significant point: although HoTEE has been much more efficient than all baselines and achieves logarithmic search complexity, there is still an efficiency gap between TEE and HE. It is noteworthy that HoTEE aims not to close the gap but to bridge it, to let mutiple parties with distinct trust on TEE/HE to collaboratively answer queries with favorable plaintext *irreversible* guarantee (Table 1, explained in §3.2).



(a) Point Query (b) Range Query

Figure 11. Performance sensitivity of running different ratios of TEE and HE nodes in SQL instances distributedly.

5.4 Robustness to Integrity Breaches

The above evaluation considers a fault-free scenario with no manipulation or omission of query results. However, ensuring the integrity of query results in an outsourced database is critical, and query authentication is necessary (§3.2). Thanks to HoTEE's MFT index that uses recursive Merkle-tree style construction with MAC-on-bias and boundary checks (§4.2), HoTEE enables such feature compared to its baselines. We ran TPC-C in SQL instances and simulated integrity breaches through random manipulation, omission, or version rollback of query messages (including both forward messages to the proxy and backward query results). HoTEE achieved a 100% detection rate of any integrity breaches.

5.5 Discussion and Limitation

Dismissing an alternative design. One may think of using solely monotonic bias with ORE for building an encrypted index, which poses two problems. Firstly, it restricts the query functionality to range queries, which is similar to CryptDB's OPE solution. Secondly, it lacks query authentication capabilities. In contrast, HoTEE enables both data confidentiality with plaintext *irreversibility* and query integrity in distributed outsourced databases.

Limitations. HoTEE has two limitations. Firstly, HoTEE does not protect the order of ciphertexts on databases, which is an inherent issue with any practical encrypted databases that aim to support range queries by not using expensive oblivious algorithms [3, 37, 39]. Secondly, HoTEE's MFT index is built on a single searchable attribute. For databases that search over multiple attributes (e.g., graph databases [7, 24]), HoTEE handles only disjunctive queries by searching on MFTs of each attribute independently. However, for conjunctive queries, MFT has to be integrated with multi-dimensional indexes (e.g., k-d tree [55]), and we leave this interesting direction for future work.

6 Conclusion

We present HoTEE, the first unified query system that can support general queries on heterogeneous EDB (i.e., simultaneously enabling HE and TEE nodes) from multiple parties with logarithmic search complexity, via a new MFT confidential index abstraction. By searching on MFTs and securely aggregating searched data, HoTEE creates a unified query system that bridges the silos between TEE and HE data, and facilitates multiple parties jointly answer queries using heterogeneous EDB nodes. Extensive results on both SQL and NoSQL instances show that HoTEE is general, extensible, and highly efficient compared to two notable EDBs. HoTEE is open-sourced and its code is released on github.com/osdip281/HoTEE.

References

[1] [n. d.]. Jeddak project. ([n. d.]). https://github.com/bytedance-jeddak/jeddak
[2] 1997. *Web resource: https://en.wikipedia.org/wiki/Ciphertext indistinguishability* (1997).
[3] Edgeless Systems GmbH. 2022. [n. d.]. *EdgelessDB Official Website. Retrieved March 1, 2022.* https://www.edgeless.systems/products/edgelessdb.
[4] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 563–574.
[5] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2022. Pantheon: Private Retrieval from Public Key-Value Store. *Proceedings of the VLDB Endowment* 16, 4 (2022), 643–656.
[6] Werner Almesberger et al. 1999. Linux network traffic control—implementation overview.
[7] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.
[8] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1511–1525.
[9] David W Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. 2018. From keys to databases—real-world applications of secure multi-party computation. *Comput. J.* 61, 12 (2018), 1749–1771.
[10] Mihir Bellare, Marc Fischlin, Adam O'Neill, and Thomas Ristenpart. 2008. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *Annual International Cryptology Conference*. Springer, 360–378.
[11] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[12] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 157–168.
[13] Rupali Chopade and Vinod Pachghare. 2020. MongoDB indexing for performance improvement. In *ICT Systems and Sustainability*. Springer, 529–539.
[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).
[15] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 655–671.
[16] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. 2020. DORY: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1101–1119.
[17] Yarkın Doröz, Gizem S Çetin, and Berk Sunar. 2016. On-the-fly homomorphic batching/unbatching. In *International Conference on Financial*

Cryptography and Data Security. Springer, 288–301.
[18] Fermat. 1997. Fermat's Last Theorem. *Web resource: https://en.wikipedia.org/wiki/Fermat27sLastTheorem* (1997).
[19] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P Smart. 2012. Ring switching in BGV-style homomorphic encryption. In *International Conference on Security and Cryptography for Networks*. Springer, 19–37.
[20] Vipul Goyal, Adam O'Neill, and Vanishree Rao. 2011. Correlated-input secure hash functions. In *Theory of Cryptography Conference*. Springer, 182–200.
[21] James L Hieronymus. 1993. ASCII phonetic symbols for the world's languages: Worldbet. *Journal of the International Phonetic Association* 23 (1993), 72.
[22] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies* 2021, 3 (2021), 246–264.
[23] Scott R. Intel. [n. d.]. SGX 2.0 (scalable SGX). ([n. d.]). https://github.com/intel/linux-sgx/issues/899
[24] Borislav Iordanov. 2010. Hypergraphdb: a generalized graph database. In *International conference on web-age information management*. Springer, 25–36.
[25] Emilia Käsper and Peter Schwabe. 2009. Faster and timing-attack resistant AES-GCM. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 1–17.
[26] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems* 34 (2021), 4961–4973.
[27] Scott T Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31.
[28] Dong Li, Xiaofeng Liao, Tao Xiang, Jiahui Wu, and Junqing Le. 2020. Privacy-preserving self-serviced medical diagnosis scheme based on secure multi-party computation. *Computers & Security* 90 (2020), 101701.
[29] Rundong Li, Mirek Riedewald, and Xinyan Deng. 2018. Submodularity of distributed join computation. In *Proceedings of the 2018 International Conference on Management of Data*. 1237–1252.
[30] Dongxi Liu and Shenlu Wang. 2012. Programmable order-preserving secure index for encrypted database query. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 502–509.
[31] Murali Mani, Kinnari Shah, and Manikanta Gunda. 2013. Enabling secure database as a service using fully homomorphic encryption: Challenges and opportunities. *arXiv preprint arXiv:1302.2654* (2013).
[32] AB MySQL. 2001. MySQL.
[33] National Institute of Standards and Technology (NIST). 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Information Processing Standards Publication 202.
[34] Seog-Chan Park and Moon-Seog Lee. 2015. A survey of access control models in database management systems. *Journal of Computing Science and Engineering* 9, 2 (2015), 57–78.
[35] Proteet Paul, Tushar Gupta, and Shamik Sural. 2022. Poster: ASQL-Attribute Based Access Control Extension for SQL. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*. 259–261.
[36] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* 2016 (2016), 591.
[37] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 85–100.
[38] Behandelt PostgreSQL. 1996. PostgreSQL. *Web resource: http://www.PostgreSQL. org/about* (1996).
[39] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.
[40] Addanki Purna Ramesh, AVN Tilak, and AM Prasad. 2013. An FPGA based high speed IEEE-754 double precision floating point multiplier using Verilog. In *2013 International Conference on Emerging Trends in VLSI, Embedded System, Nano Electronics and Telecommunication System (ICEVENT)*. IEEE, 1–5.
[41] Marc Stevens, Pierre Karpman, and Thomas Peyrin. 2017. The first collision for full SHA-1. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 570–583.
[42] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1019–1032.
[43] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. {Graphene-SGX}: A Practical Library {OS} for Unmodified Applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.
[44] Eugene Tsyrklevich and Bennet Yee. 2003. Dynamic detection and prevention of race conditions in file accesses. In *12th USENIX Security Symposium (USENIX Security 03)*.
[45] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. (2013).
[46] Harshavardhan Unnibhavi, David Cerdeira, Antonio Barbalace, Nuno Santos, and Pramod Bhatotia. 2022. Secure and Policy-Compliant Query Processing on Heterogeneous Computational Storage Architectures. In *ACM SIGMOD/PODS International Conference on Management of Data 2022*.
[47] Xiaoyun Wang and Hongbo Yu. 2005. Breaking a new version of NMAC and HMAC using preimage attacks on PB and KECCAK. *Advances in Cryptology–EUROCRYPT 2005* (2005), 1–16.
[48] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. 1993. GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review* 27, 3 (1993), 75–86.
[49] Yilei Wang and Ke Yi. 2021. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*. 1969–1981.
[50] Zhiyong Wang, Ruijie Hu, Tao Yu, and Chunyong Chen. 2019. Design and Implementation of a Database Client Blacklist Mechanism. In *2019 International Conference on Computing, Communications and Intelligence Systems (ICCCIS)*. IEEE, 71–74.
[51] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. [n. d.]. Hybrid Trust Multi-party Computation with Trusted Execution Environment. ([n. d.]).
[52] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA*. 12–17.
[53] Statistics Canada. Zachary Zanussi. [n. d.]. Privacy Preserving Technologies Part Two: Introduction to Homomorphic Encryption. ([n. d.]). https://www.statcan.gc.ca/en/about/statcan#a7
[54] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. {XRP}:{In-Kernel} Storage Functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 375–393.
[55] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–11.
[56] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.

A Appendices

A.1 Primitives for Encrypted Databases

Intel SGX. SGX [14] is a pervasively used hardware feature that provides a secure execution environment called *enclave*, where data and code execution cannot be seen or tampered with from outside. SGX has been widely adopted in recent encrypted databases [3, 39, 42] because (1) SGX provides sophisticated API including sealing and attestation, and (2) the latest SGX 2.0 greatly expands enclave memory capacity (up to 512GB/CPU [23]) to fit memory-intensive applications including databases [42].

Homomorphic BGV. To avoid trusting one TEE hardware vendor, some organizations are seeking a cryptographic root-of-trust from homomorphic encryption (HE), and are willing to pay for the deficiencies of using HE-based solutions, especially fully HE schemes like BGV [19]. BGV allows arbitrary aggregation (e.g., sum, avg) on encrypted data (with noises) without requiring a secret key for decryption, and the computation result can be fully revealed by the owner of the secret key [19]. BGV uses *key switch* algorithm to reduce ciphertext's noises, which is leveraged by HoTEE to aggregate multi-party data encrypted by distinct secret keys (§4.3).

A.2 HoTEE's Storage Model

Recall the aforementioned data model, we assume a table T with n columns $CL = \{c_0, c_1, ..., c_{n-1}\}$ and is indexed on the primary key c_0 (i.e., $PK = c_0$). A record r can be depicted as $[E(k), E(v_1), ..., E(v_{n-1})]$ where k is the value for c_0. The storage engine should provide the following API:

- EqualGet(E(k)): Retrieve the row r from T given a key k.
- RangeGet(E(k_s),E(k_e)): Retrieve all rows $\{r\}$ from T whose keys are between E(k_s) and E(k_e).
- FullGet(): Retrieve all rows $\{r\}$ from T.
- Put(r): Insert (or update) a row r to T if its key k is non-existent (or existent).
- Delete(E(k)): Delete the row r from T given a key k.

A.3 Generality of HoTEE

Is HoTEE general? We illustrate HoTEE's generality by discussing two HoTEE's core components: MFT confidential index and SDA protocol. First, MFT is general to database types including both relational and NoSQL databases, because we assume a widely-adopted unclustered architecture where index and other components (e.g., storage) are decoupled (§3.1) and MFT only substitutes the original index; MFT is also general to attribute types (integer, real value, and string): as MFT so far works on the integer attribute, we can transform real values into integers using IEEE 754 floating point number [40] and transform string values using ASCII character [21] by encoding any length-l ASCII string with integers $[0, 2^{7 \cdot l} - 1]$. Second, SDA is general as well, because SDA uses the generic MFT for searching, and aggregates data with key switch, a generic tool in common HE algorithms to reduce HE data's cumulative noises [19, 22].

A.4 Construction algorithm

Algorithm 3: MFT Index Construction (Client side)

```
Struct {
    Rangefilter myFilter                                    ▷ Input from Algo 1
    Ctx value                                               ▷ TEE or HE ciphertext
    Node pn                                                 ▷ Parent node
} MFTNode;
Function CreateLeaf() do
    ▽ Initialize bottom-layer MFTNodes
    ...
    return leaf[ ]
Function CreateInner($Node_i$, $Node_j$) do
    $(inner.left,\ inner.right) \leftarrow (Node_i,\ Node_j)$
    $(Node_i.pn,\ Node_j.pn) \leftarrow inner$
    ▽ Union of child nodes' ranges
    $inner.rng \leftarrow Node_i.rng \cup Node_j.rng$
    return inner
Function CreatMFTIndex() do
    $le[] \leftarrow$ CreateLeaf(), $in[] \leftarrow \varnothing$, $ptr \leftarrow 0$
    ▽ Pair-wise inner node construction
    while $ptr < le.length$ do
        if $ptr = le.length - 1$ then
            $in$.insert(CreateInner (le[ptr],le[ptr-1].pn))
        else
            $in$.insert(CreateInner (le[ptr],le[ptr+1]))
        $ptr \leftarrow ptr + 2$
    ▽ Bottom-up MFT construction
    while $in.length > 2$ do
        $n[] \leftarrow \varnothing$, $ptr \leftarrow 0$
        while $ptr < in.length$ do
            if $ptr = in.length - 1$ then
                $n$.insert(CreateInner (in[ptr],in[ptr-1].pn))
            else
                $n$.insert(CreateInner (in[ptr],in[ptr+1]))
            $ptr \leftarrow ptr + 2$
        $in \leftarrow n$
    ▽ Entry of MFT confidential index
    return $root \leftarrow$ CreateInner($in[0]$, $in[1]$)
```

A.5 Key switch Syntax

Specifically, key switch (KS) consists of a tuple of algorithms KS = (CtxGen, Enc, Switch) with the following syntax:

- KS.CtxGen $\longrightarrow ctx$. Generate a ciphertext ctx in the form of (a,A), in which A = $a \cdot S_A + m + te \ mod \ q$. S_A is the secret key owned by client A, m is the plaintext, a is a random parameter, other variants are modular parameters in HE.
- KS.Enc(S,S') $\longrightarrow tkn$. Enc encrypts key S using a new key S' and outputs a token tkn. tkn takes the form of (a*, A*) = $(a',\ a' \cdot S' + S + te \ mod \ q)$, a' is a random parameter.
- KS.Switch(ctx,tkn) $\longrightarrow ctx'$. By using tkn, Switch switches the encryption key of ctx from S to S' and outputs a new ciphertext $nctx = (-a \cdot a^*,\ A - a \cdot A^*)$.

A.6 Implementation Details

We implemented HoTEE with 5021 lines of C++ code on the CryptDB codebase [37], a modular framework for evaluating distributed encrypted databases. All of HoTEE's protocol messages (❶, ❸) were implemented with asynchronous RPC calls [48], and we instantiated messages using AES [25]. We modified CryptDB proxy with around 200

lines of Graphene [43] code for enclave instantiation and code wrapping (in `pr.manifest`), and spawned two enclave threads: one (`ts_thread`) for profiling, transforming queries, and searching on MFT indexes to locate query-dependent databases (❷); the other one (`ds_thread`) runs SDA protocol to generate and fetch key switch tokens (§4.3), and aggregates multi-party encrypted data with the tokens (❹).
Case study. To understand whether HoTEE can benefit real world databases, we built two HE databases on MySQL [32] and MongoDB's default KV backend WiredTiger [13], by replacing MySQL's default B^+-tree and WiredTiger's B-tree with MFT confidential index. Especially, since WiredTiger maintains an LRU cache for queries' entire B-tree traversal path including both inner and leaf nodes' pages, to comply with such caching semantics, HoTEE's search operations described in the previous section (❺) also returns all traversed MFT pages so that WiredTiger can cache them.
Optimization. First, we took a batching strategy [17] on homomorphic encryption/decryption and key switching to maximize the throughput of HE databases. Second, we fine-tuned the WiredTiger's LRU cache size to maximize cache hit ratio while keeping a large fraction of the available memory for MFT indexes for fast data retrieval. Last, we constructed MFT's rangefilter nodes by storing only the `"1"` bits and skipping the `"0"` bits to keep larger working sets of MFT indexes in memory and reduce time-consuming disk I/O.