Building Enclave-Native Storage Engines for Practical Encrypted Databases

Yuanyuan Sun, Sheng Wang, Huorong Li, Feifei Li
Alibaba Group
{yuanyuan.sun,sh.wang,huorong.lhr,lifeifei}@alibaba-inc.com

ABSTRACT
Data confidentiality is one of the biggest concerns that hinders enterprise customers from moving their workloads to the cloud. Thanks to the trusted execution environment (TEE), it is now feasible to build encrypted databases in the enclave that can process customers’ data while keeping it confidential to the cloud. Though some enclave-based encrypted databases emerge recently, there remains a large unexplored area in between about how confidentiality can be achieved in different ways and what influences are implied by them. In this paper, we first provide a broad exploration of possible design choices in building encrypted database storage engines, rendering trade-offs in security, performance and functionality. We observe that choices on different dimensions can be independent and their combination determines the overall trade-off of the entire storage. We then propose Enclave, an encrypted storage engine that makes practical trade-offs. It adopts many enclave-native designs, such as page-level encryption, reduced enclave interaction, and hierarchical memory buffer, which offer high-level security guarantee and high performance at the same time. To make better use of the limited enclave memory, we derive the optimal page size in enclave and adopt delta decryption to access large data pages with low cost. Our experiments show that Enclave outperforms the baseline, a common storage design in many encrypted databases, by over 13x in throughput and about 5x in storage savings.

PVLDB Reference Format:
doi:10.14778/3447689.3447705

1 INTRODUCTION
Due to the rapid advancement of cloud computing, many companies have moved their enterprise workloads from on-premise data centers to cloud services, who offer many attractive features, such as elasticity, high availability, and low cost. From the security perspective, the cloud tends to be less vulnerable than on-premise deployments. The service provider can employ a large team of security experts to adopt state-of-the-art protection mechanisms timely and continuously to the entire infrastructure. In this case, even huge security investments become affordable as they are amortized over all customers. However, there exposes a new dimension of attacks — this outsourced infrastructure could be compromised by insiders, such as malicious co-tenants and curious staffs, who might look into the data (e.g., in databases) and cause data breaches. In other words, anyone with privileges (or even physical access [31]) on that server can easily steal the data for his/her own interest. However, customers have no control of administrative privileges to the machines that host their data, which is completely different from on-premise deployments. Given this serious situation, it is critical to protect the confidentiality of customers’ data during the operation of cloud databases. Note that existing database security mechanisms, such as access control and data-at-rest encryption, can be easily bypassed by attackers in this context [2].

In order to tackle this problem, many research works [5, 6, 25, 47, 49, 50, 58, 61] have built encrypted databases, which prevent attackers with privileges on the database (or on the server that hosts the database) from accessing users’ data in plaintext. One line of work, e.g., CryptDB [49] and Arx [47], takes advantage of special cryptographic primitives to support direct operations over ciphertext (e.g., homomorphic encryption [28], searchable encryption [54], and garbled circuit [63]). However, they usually introduce significant overheads and only allow limited types of operations [19, 36, 46, 47, 49]. This makes them unsuitable for general-purpose cloud database infrastructures.

Instead, we follow another line of work that uses trusted execution environments (TEE), like Intel SGX and AMD SEV, to operate on confidential data in an isolated enclave. Due to the recent advancement of Intel SGX (software guard extensions), many enclave-based encrypted databases and storage systems have emerged [5, 9, 25, 40, 42, 50, 61, 64]. Although all these systems target data confidentiality, their protection strengths are sometimes either too "strong" or too "weak". Some of them make user data completely inaccessible or indistinguishable from the server. For example, EnclaveDB [50] puts all data in enclave-protected memory, and OblivDB [25] adopts oblivious data access to untrusted memory. However, such a strong protection significantly compromises either system capability or performance. On the contrary, others offer confidentiality protection as add-on features to legacy database systems. For example, Always-encrypted [5] and StealthDB [61] offer a few enclave-based functions for computation over ciphertext with marginal modifications to SQL Server and PostgreSQL. We observe that such a non-intrusive design leads to severe information leakage and performance degradation (Section 3.3). In summary, there still remains a large unexplored area between above two extreme scenarios — how confidentiality can be achieved in encrypted databases where users have more practical considerations for trade-offs among security, performance, and functionality.
In this paper, we consider the design of an encrypted storage engine, which is a fundamental building block for full-fledged encrypted databases. Instead of proposing a concrete design directly, we first provide a comprehensive exploration of possible design choices for encrypted storage engines. These choices achieve different trade-offs among security, performance and functionality, and can be further categorized into five dimensions (i.e., encryption granularity, execution logic in enclave, memory access granularity, enclave memory usage, record identity protection) as shown in Table 1. We observe that the decision on each dimension can be made independently, and their combination determines the overall trade-off of the entire storage. Moreover, not all combinations are equally useful in practice as we will discuss later. This analysis should be able to help database practitioners to well understand the effect of each decision, and guide them to find the best choices for their own needs.

After exploring the design space, we further make choices on each dimension, rendering a good trade-off for practical database usage. Under this circumstance, we propose Enclage, an enclave-native database storage engine that includes a B*-tree-like index structure and a heap-file-like table store. It allows data to be securely maintained in untrusted memory and disk by encrypting individual index and data pages. This mitigates ciphertext amplifications and prevents information leakage inside a page. To avoid unnecessary context switches between enclave and non-enclave executions, we carefully implement the main execution logic of index (i.e., entry search and update) and table (i.e., record read and write) inside the enclave. It is able to reduce the number of enclave entries per request to one in most cases, and minimize it when external interactions (e.g., I/Os) are required. We further utilize protected memory in enclave to cache frequently accessed pages, relieving the pressure of encryption/decryption overheads. However, enclave memory space is extremely limited, and we have to decide what kind of data and how much of it should be buffered to achieve the best performance. For example, we find both theoretically and experimentally that index pages of 1/2KB size outperforms those 4/8/16KB ones widely used in existing databases. For data pages, instead of buffering them in enclave, we adopt a delta decryption protocol to support fast record access with low enclave memory consumption. Note that many of our approaches and optimizations adopted in Enclage are still applicable even when other design choices are made.

Our main contributions are summarized as follows:
- To the best of our knowledge, we are the first to provide a comprehensive exploration of design choices for building enclave-based encrypted database storage. We divide the entire design space into five dimensions and discuss trade-offs implied by each individual choice. This helps database practitioners to find the best design for their own needs.
- We propose an encrypted storage engine Enclage, rendering practical design trade-offs. It adopts many enclave-native designs, such as page-level encryption, reduced enclave interaction, and hierarchical memory buffer, which offer high-level security guarantee, high performance and low storage at the same time.
- To make better use of the limited enclave memory, we propose several optimizations. A cost model is built to analyze the optimal index node size (i.e., 1-2KB), which is consistent with our experiments. A delta decryption protocol is applied to access records in data pages efficiently, having no memory contention against the index.
- We conduct extensive experiments to evaluate the efficiency of Enclage, as well as the effectiveness of individual designs adopted by it. Enclage outperforms the baseline, a common storage design in existing encrypted databases [5, 61], by over 13x in throughput and about 5x in storage savings.

2 BACKGROUND

In this section, we briefly introduce the concept of trusted execution environment and Intel SGX, and then discuss the challenges in designing SGX-based encrypted databases.

2.1 Trusted Execution Environment and SGX

A trusted execution environment (TEE) is a secure area, which guarantees the confidentiality and integrity of computation and data in it. It can be used to build secure applications in untrusted environments (e.g., on a public cloud), where the host may conduct malicious actions. Intel Software Guard Extensions (SGX) is a state-of-the-art implementation of TEE, receiving broad attention from both industry and academia. SGX is an extension of x86 instruction set architecture, and it offers protections using a concept called enclave. Readers can refer to [22, 33, 35] for more details on its implementation and features, e.g., isolation, sealing and attestation.

An enclave is an isolated virtual address space in a process, where both code and data are stored in protected memory pages called enclave page caches (EPC) that cannot be accessed by the host, i.e., the rest of the process outside the enclave. The data in EPC is encrypted by a memory encryption engine (MEE), which only decrypts data in the CPU cache. The host can only initialize the enclave by loading a specially compiled library (e.g., *.signed.so whose authenticity can be verified), and later interact with the enclave via well-defined functions (i.e., ECall/OCall). Note that the enclave can access the entire address space of the process, while the host cannot access enclave’s memory. This is an important property that can be exploited for fast data fetching, compared to co-processor TEE implementations (e.g., FPGA) where memory access has to go through PCIe. SGX provides remote attestation that allows the client to verify the authenticity of an enclave and its loaded code/data on a remote host. It facilitates the establishment of a secure channel between the client and the enclave (e.g., to pass secret keys). Note that AMD SEV [4, 38, 39] is a promising alternative to SGX to ensure the confidentiality of user data hosted by an untrusted server, but currently lacking of integrity protection [39, 53] and security-proven remote attestation [17]. The latter is a necessity for secure key provision to encrypted databases on the cloud.

2.2 Challenges for SGX-based Databases

SGX provides essential primitives for building encrypted databases. However, there remain many critical challenges to achieve a robust and performant design. They are attributed to both the nature of TEE techniques and the limitations from SGX implementation.

Limited memory space in enclave. Due to hardware limitations for securing the memory, the preserved memory region is limited to 128MB [8, 9, 45] (or 256MB in the latest implementation)
and the actual EPC capacity is even slightly lower. Furthermore, this capacity is shared by all enclaves running on that processor. To hide this limitation, SGX provides a virtual memory abstraction to evict recently unused EPC pages to the host memory. To ensure confidentiality and integrity, evicted pages are encrypted and stored in a Merkle Tree [27, 56]. An EPC fault (i.e., loading a page back) will cause significant penalty [8, 22], easily exceeding 40000 CPU cycles. In databases, many components (e.g., buffer pool) and operations (e.g., join) are inherently memory-consuming, which should be carefully re-designed. For example, a naive adaptation of indexes in enclave memory could underperform by three orders of magnitude [9].

**Huge cost from enclave interaction.** The host process can only invoke enclave via pre-defined function calls (ECall). Since an enclave runs in user mode (ring 3), it cannot execute system calls, which instead requires to temporally exit the enclave (value size resolves these issues. However, importing complex database functionality into the enclave inherits all vulnerabilities from the codebase. Hence, reducing TCB size (codebase size) makes it more robust, where an exhaustive examination of codebase becomes feasible. Note that retrieving on non-key columns are also supported. We can either conduct a FullScan to exact qualified records or build secondary indexes on them.

### 3.1.3 Index and Table Store.

We consider the storage architecture that consists of a B*-tree [24] index and a heap-file table store. Data in both the index and table store is organized as many index or data pages (e.g., of 4KB size). A new record is first appended to the heap file of the table store, and is assigned a record identifier (rid for short). The rid can be used to retrieve the corresponding record from the table store. An index stores a pair of index key (i.e., value in the indexed column) and rid for each record. In this case, indexes on both primary-key column and non-key columns are unclustered. Some sophisticated databases, such as PostgreSQL, follow similar settings. Note that rids passed between indexes and table store might be either ciphertext or plaintext, which affects the information leakage inside the database. We denote an optionally encrypted rid as \(OE(rid)\), which is either rid or \(E(rid)\). To support storage operations above, the table store \(TS\) has to support:

- \(TPut(r)\): insert a row to \(TS\).
- \(TGet(OE(rid)) \rightarrow r\): get the row from \(TS\) given a rid.
- \(TUpdate(OE(rid), r)\): update the row from \(TS\) given a rid.
- \(TDelete(OE(rid))\): delete the row from \(TS\) given a rid.
- \(TFullScan() \rightarrow \{r_1, r_2, ..., r_m\}\): retrieve all rows from \(TS\).

The primary index PI has to support:

- \(IIInsert(E(k), OE(rid))\): insert an entry with key \(k\) to \(PI\).
- \(ISearch(E(k)) \rightarrow OE(rid)\): search the entry with key \(k\).
- \(IReplace(E(k), OE(rid))\): replace the entry with key \(k\).
- \(IRemove(E(k))\): remove the entry from \(PI\) with key \(k\).
- \(IRangeSearch(E(k_1), E(k_2)) \rightarrow \{rid_1, ..., rid_j\}\): search all entries from \(PI\) whose keys are between \(k_1\) and \(k_2\).

The secondary index is similar and we omit its functions here. We follow the separation of table store and unclustered indexes, because it enables the discussion for more design choices (Section 3.4). It already covers the case for clustered indexes, i.e., we can replace rids with records in the primary index and have primary key’s ciphertext in secondary indexes.
3.2 Threat Model

Our major goal is to ensure the confidentiality of user data hosted by an untrusted database server. Similar to [5], we target a strong adversary with privileged access to OS and database, who can not only monitor the content of all server’s memory, disk and communication, but also actively tamper with it (e.g., attaching a debugger to database). However, the adversary cannot access enclaves provided by SGX (or any TEE with capabilities similar to SGX). In particular, data and computation inside an enclave are protected with respect to the confidentiality and integrity. The communication between enclave and host (e.g., ECall/OCall, direct access to host memory inside an enclave) is still exposed to the adversary. We exclude SGX side-channel attacks [44, 59] from our scope, since these vulnerabilities are implementation specific and we can adopt more secure TEEs when needed. Note that this adversary is stronger than common honest-but-curious adversary [3, 7, 10, 48, 49, 58], who only observes passively.

We do not consider the confidentiality of metadata and coarse statistical properties, such as the name of tables/columns, the volume of tables/indexes, the length of values. In addition, we do not pursue strict indistinguishability of encrypted data and storage operations, since it usually leads to impractical performance penalty [25, 42]. Instead, we aim to provide operational data confidentiality as has been discussed in prior work [5, 7, 49], where the information that the adversary learns is a function of data operations that have been performed. Since the adversary do not have the DEK, it cannot perform arbitrary data operations of its choice. The actual information leakage depends on the design choices, as listed in Table 1 Security column. For example, key comparisons in enclave leaks the ordering of individual keys, while index node accesses in enclave hides this information but still leaks parent-child node relationships.

Note that other security guarantees, such as the ciphertext amplification from denial of service, are out of our focus. In fact, the complexity of additionally protecting data integrity also depends on the design choices. Nevertheless, we will discuss how to make our design integrity-protected (Section 4.6).

3.3 Strawman: $B^*$-tree with Encrypted Keys

$B^*$-tree [24] is the most widely used index structure in relational databases. Here we introduce the design of a simple encrypted $B^*$-tree in existing work [5, 61].

3.3.1 Structure Overview. Figure 1 shows the structure, which is almost the same as a normal $B^*$-tree, except that index keys in gray blocks are ciphertext. Other fields, such as node pointers and rids, are in plaintext. Its logical semantics and representations remain unchanged, i.e. index keys are still ordered by their corresponding plaintext. To support index lookup and updates, comparisons on encrypted keys are processed by the enclave. We load the search key and compared key into enclave, which decrypts them and compares back the comparison result in plaintext (e.g., if $X < Y$ is true or false). For example, to lookup the search key E(42) shown in Figure 1, other keys encountered during the tree traversal are sent to enclave one by one (from root to the target leaf node).

The advantage of this approach is that most index processing logic, e.g., node split and merge, remains unaffected outside the enclave. Furthermore, the table store works without enclave, and uses plaintext rids to retrieve records.

3.3.2 Limitations. The above approach is easy to adopt but suffers from several severe issues:

- **Frequent enclave interaction.** Putting each individual comparison into the enclave results in frequent ECalls, affecting performance significantly (Section 2.2). For example in Figure 1, five ECalls are required for a single index lookup.

- **High ciphertext overhead on computation and storage.** Encryption schemes are unfriendly to small ciphertext due to the initialization cost. Processing many pieces of small cipher is much more expensive than processing a single piece of large cipher with respect to the same volume of plaintext. Second, each ciphertext has a metadata field, whose size is constant (32 bytes in our case) regardless of the cipher length. Hence, the smaller the encryption granularity is, the higher the ciphertext amplification will be. For example, comparing a single int (4 bytes) and a data page (4KB), their ciphertext amplifications are 8x and 1%, respectively. Moreover, a higher amplification factor means smaller node fanouts and lower index efficiency.

- **Severe information leakage.** Since this design keeps $B^*$-tree’s internal states in plaintext, even a snapshot adversary [11, 16, 26, 51, 60] can learn its structure, such as key orders and parent-child relationships. This lowers the security strength of any encryption scheme to the similar level of order-preserving encryption [14]. During an index lookup, the placement and traversal path of the search key is leaked, making it vulnerable to most leakage-abuse attacks [18, 29].

3.4 Exploration to a Broader Design Space

There are other enclave-based encrypted storage systems that provide much stronger confidentiality protection [25, 42, 50]. EnclaveDB [50] puts all data in enclave, but it requires large EPC capacity. ObliDB [25] and Oblix [42] leverage oblivious structures to make data access indistinguishable, but their costs are prohibitive in practice. In summary, there is no discussion on how confidentiality can be achieved in different ways, and what influences are implied by them.

Here we provide a comprehensive exploration of many choices for designing an encrypted storage, rendering a broader design space. Table 1 lists all design choices that we have explored, which are further categorized into five dimensions. This table also summarizes the influence of each choice in terms of security, performance and functionality. We observe that the choice made on each dimension can be independent in a sub-system (e.g., index or table...
Table 1: Possible design choices for encrypted storage categorized in five dimensions. The choices made for Enclave Index are bolded and the choices for Enclave Store are tagged with an asterisk (*).

<table>
<thead>
<tr>
<th>Design Dimension</th>
<th>Design Choice</th>
<th>Influence</th>
<th>Performance</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption Granularity</td>
<td>item-level encryption</td>
<td>leak structural information</td>
<td>high storage overhead, fast for a single read</td>
<td>can fetch data w/o enclave</td>
</tr>
<tr>
<td></td>
<td>page-level encryption</td>
<td>leak data volume only</td>
<td>low storage overhead, fast for batched small reads</td>
<td>all data access must be in enclave</td>
</tr>
<tr>
<td>Execution Logic in Enclave</td>
<td>index, key comparison</td>
<td>leak key ordering and search path</td>
<td>low performance from massive ECall(s)</td>
<td>can split or merge node w/o enclave</td>
</tr>
<tr>
<td></td>
<td>item-level access</td>
<td>high performance from a few ECall(s)</td>
<td>can fetch or scan recording w/o enclave</td>
<td></td>
</tr>
<tr>
<td>Memory Access Granularity</td>
<td>table: page-level access</td>
<td>leak page-level identity and location</td>
<td>high performance from no ECall(s)</td>
<td>can fetch or scan recording w/o enclave</td>
</tr>
<tr>
<td></td>
<td>item-level access</td>
<td>high performance from on-demand read</td>
<td>only useful in some settings</td>
<td></td>
</tr>
<tr>
<td>Enclave Memory Usage</td>
<td>item-level access</td>
<td>moderate performance from page copy</td>
<td>require small footprint in enclave</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page-level access</td>
<td>high performance from page cache</td>
<td>high EPC capacity requirement</td>
<td></td>
</tr>
<tr>
<td>Record Identity Protection</td>
<td>item-level encryption</td>
<td>low performance from active data fetching</td>
<td>no EPC capacity requirement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page-level encryption</td>
<td>high performance from data caching</td>
<td>high EPC capacity requirement</td>
<td></td>
</tr>
</tbody>
</table>

store), while the choices made by different sub-systems can also be independent. In fact, the combination of all choices determines the overall trade-off of the entire system. In the following, we elaborate on major design choices available in each dimension.

3.4.1 Encryption Granularity. To prevent the adversary from seeing plaintext, data can be encrypted at different granularities: at item level (e.g., an index key or a column value) or at page level (e.g., a data page). Overall, item-level encryption suffers from high computation and storage overhead, as well as structural information leakage (Section 3.3.2). We call this offline leakage as it is leaked even when the database is offline. Note that metadata can be optionally encrypted to mitigate this leakage, e.g., encrypting node pointers helps to hide parent-child relationships. The advantage of item-level encryption is its flexibility and portability, allowing many operations to bypass enclave, such as record access and node split. In contrast, page-level encryption has marginal offline leakage and negligible ciphertext amplification. The encryption and decryption costs are amortized to all items inside a page, especially suitable for accesses with high data locality. However, it relies on enclave for all data manipulation. Whenever a single byte needs to be accessed, the entire page must be decrypted to enclave’s memory.

3.4.2 Execution Logic in Enclave. Putting execution logic in enclave prevents the adversary from observing the online status (i.e., plaintext data and execution flow) during the processing over indexes and tables. There is always a dilemma in the choice of TCB size (Section 2.2). In our context, the minimum operations that must be processed by the enclave are key comparisons in indexes. The rest of executions, such as table store management, are feasible to be entirely out of the TCB. For the index, key comparison in enclave leaks entry-level search path of each lookup and requires massive ECalls. Its advantage is that complex logic, such as node split and merge, remains unaffected in the host. Index node access in enclave hides the sequence of key comparisons in the same node via a single ECall, reducing search path leakage to the node level. For the table store, none in enclave is fast, but leaks record-level data location from each request, i.e., identities of returned records are exposed among issued requests. Data page access in enclave can reduce above identity leakage to the page level, but relies on the enclave to achieve this protection.

3.4.3 Memory Access Granularity. Recall that the enclave can access the entire address space of the process, including unprotected memory in the host (Section 2.1). However, such memory access, as well as OCalls, can be monitored by the adversary. It leaks access patterns when the enclave execution involves access to host memory. Item-level access reads each item into the enclave on demand, e.g., read the index key to compare with. This approach leaks item-level access patterns, but is fast. Page-level access loads the entire page into the enclave whenever a single byte is needed. It hides the exact item used by the enclave at the cost of a page copy. Note that oblivious structures [21, 25, 30, 52, 55] are designed to protect such patterns by introducing additional accesses. They are rather orthogonal and can be further adopted.

3.4.4 Enclave Memory Usage. Recall that the EPC capacity is extremely limited (Section 2.1). The performance of the enclave execution heavily depends on its local EPC usage and the global EPC usage from all concurrent enclaves running on that processor. When unused EPC space is available, it could be used to cache frequently accessed data (either items or pages), which avoids expensive decryption costs from repeated accesses. Moreover, the access to cached data is protected, i.e., the adversary cannot tell whether and how many times an item has been recently accessed. One can choose to either reserve a fixed size of EPC to each storage or make the reserved size proportional to the storage volume. In an extreme case, pre-loading all data in unlimited EPC can completely eliminate the decryption cost and access pattern leakage of subsequent accesses.

3.4.5 Record Identity Protection. For two Get requests initiated by upper-layer database components (e.g., query executor) at the untrusted server, their equality will be exposed if the adversary can observe that they retrieve the same record. To avoid such query pattern leakage, the record identity must be hidden during processing. In particular, it involves the protection on both rids obtained from the index and records retrieved from the table store. When the rid is in plaintext, its record identity is already exposed. Hence, rid encryption is necessary to hide the identity, but it is still far from enough. If the returned ciphertext of the same rid (or record) is consistently identical, the identity is still exposed. The dynamic ciphertext re-encryption can solve this problem by returning a distinct ciphertext each time for both rids and records. However, the complete record identity protection still requires the assistance from choices in other dimensions. For example, key comparison in enclave already leaks entry identity, where re-encryption on neither rid nor record will be useful then.
4 DESIGN OF ENCLAGE

After exploring the entire design space, we introduce Enclave, an enclave-native encrypted storage engine for practical usage. It consists of two major components: Enclave Index, a B*-tree-like index; and Enclave Store, a heap-file-like table store.

4.1 Design Choices Made in Enclave

Different design choices render trade-offs among security, performance and functionality. Whether a choice is practical highly depends on the application scenario, e.g., leakages that must be prevented and functions that must be supported. Here we explain design considerations behind Enclave for practical usage. Enclave makes different choices for its index and table store, as marked in Table 1. In a nutshell, Enclave applies page-level encryption for both components to benefit from its security advantage and storage efficiency. However, due to the limited EPC capacity and costly encryption overhead, we find that the page size must be carefully tuned to achieve the best performance (Section 4.3), and hence set different sizes for index nodes and data pages. Except for the encryption granularity, Enclave Index and Enclave Store make distinct choices on other design dimensions. For Enclave Index (Section 4.2), it puts index node access in enclave to minimize ECall1s and follows page-level access to load demanded nodes in host memory. Since the index efficiency is critical to the entire storage and few effective data locality exists in heap-file-based table store (especially for range queries), we reserve most EPC space to cache Enclave Index nodes with fixed EPC usage. For Enclave Store (Section 4.4), it puts data page access in enclave to leverage data-page encryption. We observe that, in most cases, data locality in the heap file is much lower than that in the index. Hence, Enclave Store prefers to not cache data pages in enclave, i.e., with minimal EPC usage, conceding most EPC memory to Enclave Index. Instead, it adopts a delta decryption protocol to enable item-level access to target record, which significantly lowers encryption cost. Note that rid encryption and ciphertext re-encryption is inherently effective and efficient in Enclave, and we enable them by default. Enclave hence achieves indistinguishability for entry/record access inside a page.

4.2 Enclave Index

4.2.1 Overview. Figure 2 shows the overall design of Enclave Index. It adopts a three-tier storage hierarchy, including a EBuffer tier, a MBuffer tier and an external storage tier. As Enclave Index resembles a B*-tree, index nodes are organized as pages and are placed at any of these tiers. Unlike in conventional B*-tree where each page contains a single node, a page in Enclave Index might hold several nodes to reduce encryption cost. We construct a buffer manager (called EBuffer) in the enclave to manage page transfers between unprotected host memory and protected enclave memory. Each EBuffer page contains a single index node and is not encrypted. Another buffer manager (called MBuffer) is constructed to manage page transfers between host memory and external storage. Each MBuffer page contains multiple encrypted EBuffer pages. To make the presentation more concise, we call a EBuffer page as a node and an MBuffer page as a page in following sections.

EBuffer and MBuffer are different in terms of the encryption status, placement and capacity. For example, EBuffer manages plaintext nodes in limited enclave memory, while MBuffer manages ciphertext pages in large host memory. Apart from these, both buffer managers have the same structure. Each buffer manager comprises three components, namely, a hash map, a buffer descriptor, and a buffer pool (detailed in Section 5.2.1). Moreover, the address translation is needed between any two adjacent tiers in the memory hierarchy, in order to convert upper layer representations to lower layer representations (e.g., nodeID to a pair of pageID and offset, and pageID to physical address).

We carefully implement the main execution logic of index, EBuffer and MBuffer inside the enclave. When executing an ISearch operation (Section 3.1.2), the tree is traversed starting from the root to leaf node. When a node does not exist in EBuffer, it has to be fetched from MBuffer or even external storage through OCall1. A node might be evicted from EBuffer to make room for other nodes. Moreover, node modifications in EBuffer will not be written through to MBuffer immediately. If the evicted node has been modified, it will be encrypted and written back to the MBuffer. More details about the working process are discussed in Section 5.2.3.

4.2.2 Optimizations. Though the direct porting of B*-trees shall work with our three-tier hierarchy, there remain several critical issues that significantly affect the performance. Enclave Index addresses following issues with its own optimization:

Reduction of EPC page swapping overhead. Recall that built-in EPC page swapping is prohibitive (Section 2.2). To resolve this issue, we need to ensure that the EPC usage from EBuffer and all other components will not exceed the limit. In Enclave, we leave most of EPC quota to EBuffer and minimize EPC usage from other components, such as Enclave Store. In this case, the EBuffer acts as a faster EPC swapping protocol dedicated for index nodes, since it only ensures confidentiality but ignores integrity. More details of EBuffer page swapping are discussed in Section 5.2.3.
Mitigation of encryption and decryption cost. Due to the node-level encryption, the node size significantly affects the EBuffer hit ratio, which further affects the frequency of encryption and decryption. We observe that this decryption cost dominates the overall performance. Hence, we build a cost model to analyze the influence of different node sizes to the overall Enclave Index, in order to derive the optimal node size. We observe that the optimal node size for EBuffer is 1KB, which is different from those in existing databases that only consider I/Os (e.g., 4/8/16KB). More details are provided in Section 4.3.

Avoidance of unnecessary OCall. MBuffer maintains encrypted pages and the simplest way is to build it as an independent module outside the enclave. Whenever a target index node is not in EBuffer, we exit the enclave via OCall and let the MBuffer prepare the page from either its buffer pool or external storage. However, this inherently results in frequent OCalls. Since an enclave has direct access to host memory, we can get a page from MBuffer without OCall, as long as that page is already in host memory and its placement is known to the enclave. Hence, we implement part of MBuffer’s execution logic in enclave to avoid unnecessary OCall. Only when the page is on external storage, will an OCall is invoked. More details are provided in Section 5.2.3.

4.3 Cost Model for Optimal Node Size

The choice of node size affects many aspects of Enclave Index, such as the encryption/decryption cost, node fanout, and EBuffer hit ratio. Here we build a cost model to derive the optimal node size to improve overall performance. According to the following analysis, the optimal performance is achieved when the node size is between 1KB and 2KB (in common settings). This is consistent with our experiment results in Section 6.2.4.

4.3.1 Assumptions. To simplify the model, we make several assumptions: 1) Query distribution - we assume that all keys are uniformly queried. 2) EBuffer size - we assume that only the bottom two levels, i.e., leaf nodes and their parent nodes, will be evicted from EBuffer to MBuffer. We observe that 10MB is enough to achieve this goal in our experiments. 3) MBuffer size - we assume that MBuffer is large enough to accommodate all nodes, i.e., no I/O occurs during index lookup. This allows us to focus on overheads introduced by Enclave Index. 4) Write-back probability - we assume that when a node is evicted from EBuffer, the write-back probability is a constant \(r_{\text{back}}\), i.e., the chance it has been modified while in EBuffer. This is mainly affected by the write ratio in the workload. Above assumptions will not affect the generality of our analysis, and it can be extended accordingly when other assumptions are made.

4.3.2 Notations. Table 2 summarizes major notations used in the analysis. Note that node size \(p\) is the only variable to be determined and all capital-letter notations are environment-specific constants. The rest notations can be derived from them. For example, the maximum number of entry slots in EBuffer is \(n_e = \frac{E_{\text{s}}}{p}\).

Table 2: Major notations used in cost model.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Notation</th>
<th>Description (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h)</td>
<td>tree height</td>
<td>(p)</td>
<td>node size</td>
</tr>
<tr>
<td>(N)</td>
<td>total # of indexed keys</td>
<td>(l_{\text{meta}})</td>
<td>node metadata length</td>
</tr>
<tr>
<td>(n_e)</td>
<td>total # of entries in EBuffer</td>
<td>(L_{\text{key}})</td>
<td>key length</td>
</tr>
<tr>
<td>(n_{\text{s}})</td>
<td>total # of entries in MBuffer</td>
<td>(L_{\text{id}})</td>
<td>nodeID length</td>
</tr>
<tr>
<td>(f_{\text{oj}})</td>
<td>max fanout of leaf node</td>
<td>(L_{\text{oid}})</td>
<td>rid length</td>
</tr>
<tr>
<td>(f_{\text{oi}})</td>
<td>max fanout of internal node</td>
<td>(S_{\text{E}})</td>
<td>EBuffer size</td>
</tr>
</tbody>
</table>

In a B⁺-tree, each node has at most \(f_{\text{oj}}\) and at least \(\lfloor f_{\text{oj}}/2 \rfloor\) entries. Assume that the tree height is \(h\) and each node contains \(\lceil \frac{f_{\text{oj}}}{2} \rceil\) keys, which is the worst case. We say that the root is at 1-st level, and leaves at \(h\)-th level. For \(i\)-th level, the number of nodes \(n_{\text{num}}\) is

\[
n_{\text{num}} = \left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor^{i-1} (1 \leq i \leq h). \tag{2}
\]

The number of nodes from the root to the \((h-2)\)-th level is then:

\[
n_{\text{num}}_{[1,h-2]} = 1 + \left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor + \cdots + \left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor^{h-3} = \frac{\left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor^{h-2} - 1}{\left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor - 1}, \tag{3}
\]

which are all stored in EBuffer. The nodes in last two levels is

\[
n_{\text{num}}_{[h-1,h]} = \left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor^{h-2} + \left\lfloor \frac{f_{\text{oj}}}{2} \right\rfloor^{h-1}. \tag{4}
\]

Since each leaf node contains \(\lceil \frac{f_{\text{oj}}}{2} \rceil\) entries, we have \(\lceil \frac{f_{\text{oj}}}{2} \rceil^{h} \geq N\), and hence \(h = \lceil \log_{\frac{f_{\text{oj}}}{2}} N \rceil\).

4.3.3 Cost Model. Based on above notations, we model different types of cost as follows.

**EBuffer miss ratio.** In EBuffer, the number of entry slots reserved for nodes at last two levels is \(n_e = n_{\text{num}}_{[1,h-2]}\). With uniform accesses, the probability of an EBuffer miss \(r_{\text{miss}}\) is

\[
r_{\text{miss}} \approx 1 - \frac{n_e - n_{\text{num}}_{[1,h-2]}}{n_{\text{num}}_{[h-1,h]}}. \tag{5}
\]

For each index lookup, the expected number of EBuffer misses is \(E_{\text{miss}} = 2 \cdot r_{\text{miss}}\) (from nodes in last two levels).

**EBuffer lookup cost.** During an index lookup, a sequence of nodes from root to leaf should be accessed in EBuffer. To locate a node in EBuffer, a hash is calculated to locate the target bucket, costing \(T_{\text{hash}}\). Then, a chain of nodes are checked until the target node is found, whose cost is proportional to the total number of entries in EBuffer, costing \(T_{\text{chain}} \cdot n_e\). The EBuffer lookup cost per node is hence \(C_{\text{EBuffer}} = T_{\text{hash}} \cdot n_e + T_{\text{hash}}\), where both \(T_s\)s are implementation-specific constants. Moreover, EBuffer is accessed \(h\) times per query.

**MBuffer lookup cost.** When an EBuffer miss occurs, the node must be first fetched from MBuffer. Similar to EBuffer, the MBuffer access cost depends on counterparts \(T_{\text{Mchain}}\) and \(T_{\text{Mhash}}\), i.e., \(C_{\text{MBuffer}} = T_{\text{Mchain}} \cdot n_m + T_{\text{Mhash}}\). For a query, the expected number of MBuffer access is equal to EBuffer misses \(E_{\text{miss}}\).

**Node decryption cost.** The number of nodes to be decrypted per query is \(E_{\text{miss}}\), equaling to the number of EBuffer misses. To decrypt a node, the preparation cost is \(T_{\text{init}}\) and the decryption cost is proportional to the node size, i.e., \(T_{\text{dec}} \cdot p\). Hence, the node decryption cost is \(C_{\text{dec}} = T_{\text{dec}} \cdot p + T_{\text{init}}\), where \(T_{\text{dec}}\) and \(T_{\text{init}}\) are scheme-specific constants.

**Node encryption cost.** Similarly, the node encryption cost is \(C_{\text{enc}} = T_{\text{enc}} \cdot p + T_{\text{init}}\), where \(T_{\text{enc}}\) and \(T_{\text{init}}\) are scheme-specific constants. However, the number of encryption is additionally affected by the write-back probability, i.e., \(E_{\text{miss}} \cdot r_{\text{back}}\).

**Key comparison cost.** Many keys in a node are compared with the search key before the target one can be located. The cost of a
single key comparison is $T_{\text{cmp}}$. With binary search, the cost of key comparisons per node is $C_{\text{cmp}} = T_{\text{cmp}} \cdot \log\left(\frac{L}{e}\right)$.

**Total cost of Enclage Index lookup.** The index lookup cost in Enclave can be derived from accumulating the cost from individual steps discussed above:

$$C_{\text{total}} = h \cdot (C_{\text{EBuffer}} + C_{\text{cmp}}) + C_{\text{miss}} \cdot (C_{\text{MBuffer}} + C_{\text{back}} + C_{\text{enc}} + C_{\text{dec}})$$

In our setting, the constant values are: $S_r = 80MB, N = 10M, L_{\text{key}} = 8B, L_{\text{meta}} = 24B,$ and $L_{\text{id}} = 8B$. We observe that when the node size changes from 0.5KB to 16KB, the total cost first decreases and then increases, and reaches the minimum cost when $p \in [1KB, 2KB]$. Since the node size has to be a power of 2, the best choice should be 1KB (slightly better in our experiment) or 2KB. In addition, we found that the key length $L_{\text{key}}$ is the most dominant factor that affects the choice of $p$. For example, when $L_{\text{key}}$ changes from 8B to 4B, the optimal $p$ shifts to $[2KB, 4KB]$.

### 4.4 Enclage Store and Delta decryption

#### 4.4.1 Overview

Enclage Store is a heap-file-like table store, in which each new record is appended to a heap file and get a unique $r_{\text{id}}$. After that, a $<\text{key}>, r_{\text{id}>}$ pair is inserted to Enclage Index. This $r_{\text{id}}$ contains the record location (i.e., page identifier, offset and record length), and can be used to retrieve, update and delete the corresponding record later. Unlike in conventional heap file where a record can be simply appended, Enclage Store has to encrypt each individual data page. To achieve this, we maintain an active data page in enclave memory to hold recently arrived records. When the page is full, it will be encrypted and flushed to MBuffer. Recall that arrived records are value-wise encrypted (Section 3.1.2). To reduce their memory footprint, we first safely decrypt them in enclave before appending them to the active page.

Due to the nature of append-only strategy, data locality in Enclave Store is much lower than that in Enclage Index. We choose to not cache data pages (except the active page) in EBuffer. Hence, most EPC memory can be reserved to accelerate Enclage Index lookup. Instead of retrieving a record, we have to load the entire page into enclave, decrypt it and extract the target record. To reduce the record exact cost, we adopt a delta decryption protocol (Section 4.4.2) that enables direct access to the target record without encrypting the entire page. As a result, different from Enclage Index whose optimal node size is 1KB, the data page size in Enclage Store can be aligned with sophisticated choices (e.g., 4/8/16KB) without affecting the performance. In our implementation, MBuffer manages both index and data pages of 4KB size, while EBuffer manages only index nodes of 1KB size.

#### 4.4.2 Delta Decryption Protocol

The delta decryption protocol is built on the AES counter mode (AES-CTR), which allows a small block within a large cipher be solely decrypted. AES-CTR utilizes a monotonous counter, whose value is different for each block, to initialize the decryption. As shown in Figure 3, though the page is encrypted as a whole using AES-CTR, the IV of each individual record can be derived by concatenating a 12B page-wise nonce and a 4B counter. Each data page keeps its nonce in the metadata, while the counter is calculated based on the offset of the target record. To be compatible with CTR, this offset must be aligned to 16B. Hence, the counter of record $r_j$ becomes $\lfloor \text{offset}_r / 16 \rfloor$. For example, the counter for $O f f s e t_3$ in Figure 3 is 64/16 = 4. Note that if the record boundaries are not aligned to 16B, additional bytes need be decrypted. For example, to extract $r_2$ in Figure 3, part of $r_1$ shall be decrypted as well.

When executing a TGet operation (Section 3.1.2), we first locate the page in MBuffer via its pageID and load it to enclave (for hiding record-level access patterns). We then calculate the counter for the record, and construct the IV using the counter and the page-wise nonce. Now we are able to decrypt the target record with low cost. Note that the extracted record needs be value-wise encrypted before it leaves the enclave.

### 4.5 Scalability

The SGX supports multi-threading and provides thread synchronization primitives [34]. The concurrency control mechanisms of in-enclave data structures can be implemented almost the same as its counterpart in untrusted world, sustaining similar scalability capacity. Hence, the scalability issue is almost orthogonal to Enclage’s design considerations, which focus on resolving performance and security issues from enclave-specific limitations. Those scalable B-tree variants and lock-free data structures [12, 15, 23, 32, 37, 43] can be further adopted to achieve better scalability.

### 4.6 Integrity Protection

Apart from confidential protection, it is feasible to further enhance Enclage with integrity protection if needed. For Enclage Index, we can replace the $B^*$-tree structure with a Merkle B-tree [41] that provides resiliency to tampering and replay attacks [27, 56]. In a Merkle B-tree, each index node additionally contains the digest of its child nodes, where updating a node requires cascaded digest re-calculation back to the root. Fortunately, with the help of enclave, we can delay the digest re-calculation of an MBuffercached node until it is to be evicted. This significantly reduces computation cost and improves concurrency. In our implementation, these digests are stored in a separate file, so that it can be made optional and the Enclage Index structure remains unchanged. For Enclage Store, heap pages can be protected by Merkle-tree similarly. Alternatively, if we can make heap pages immutable, authenticated encryption schemes (e.g., AES GCM) are more than enough.

### 5 SYSTEM IMPLEMENTATION

In this section, we provide our implementation details, including the internal data structures and the buffer manager.

#### 5.1 Structure Format

We briefly introduce the format of major data structures implemented in Enclage. 1) Cipher. The metadata of a Cipher includes: IV and MAC needed in decryption, and the length of ciphertext, which could be either an index node or a data page in our context. 2) Index
node. The metadata of an index node includes: its unique identifier (nodeID), nodeIDs of adjacent nodes (i.e., parent and two siblings), and its type (leaf or internal). Each entry contains a rid in a leaf node or a child nodeID in an internal node. 3) MBuffer entry. Its metadata includes: the unique identifier in its data file (pageID), the dirty bit (dirty), and the number of components accessing the page (refcount). Recall that each MBuffer page could contain multiple index nodes, there is a exists field to indicate whether each node is valid. 4) EBuffer entry. Its format is similar to that of a MBuffer entry. The only difference is that the stored node is plaintext.

5.2 Buffer Management

5.2.1 Buffer Manager Structure. A buffer manager has three major components. A buffer pool contains an array of entry slots, each of which can store an index node (for EBuffer) or a page (for MBuffer). Each entry can be directly located by its index (i.e., buffer slot id). A buffer descriptor contains an array of entry metadata descriptors, each of which holds the metadata (Section 5.1) of the page stored in corresponding buffer pool slot. A hash map is built to fast lookup the node or page in the buffer pool given its nodeID or pageID, respectively. We adopt external chaining to resolve hash collisions. The hash function used is MurmurHash [1], which is a practical approach used in many systems.

5.2.2 Address Translation. Recall that the choice of node size is critical for enclave-based indexes (Section 4.3). Hence, we decouple the sizes of index node and disk page, and this additionally needs an address translation function to fill in the gap among three layers in the memory hierarchy. In particular, for EBuffer and MBuffer, we have nodeID = pageID + X + Y, where the X is the number of index nodes per page, and Y ∈ [0, X) is the node ranking in that page. For MBuffer and external storage, we have physicalAddress = pageID + pageSize + of fset.

5.2.3 Working Procedure. Here we explain the procedures of buffer-related operations. When a node is accessed, there are three cases: 1) It is already in EBuffer. In this case, we simply lookup the hash map to obtain the EBuffer entry containing that node; 2) It is not in EBuffer, but in MBuffer. We need to allocate a buffer entry in EBuffer, and then calculate its pageID and offset. With this information, we can find the node in MBuffer and decrypt it to the allocated entry in EBuffer. Meanwhile, we need to increase that MBuffer page’s refcount by one; 3) It is not in EBuffer and MBuffer, but in the disk. In this case, we have to allocate a buffer entry in MBuffer, and load the corresponding page from disk before loading it to EBuffer. Note that this is the only occasion that an OCall is invoked, and hence we minimize the enclave interaction.

Write back a victim from EBuffer. When EBuffer is full, a victim node has to be selected through LRU strategy before we can load a new node. If the victim has not been modified, it is simply dropped and the refcount of corresponding page in MBuffer is decreased by one. When it is dirty (i.e., has been modified or is a new node), we need to write it back to corresponding page in MBuffer. When the node is an existing node, the page should be pinned in MBuffer (i.e., refcount > 0). In this case, we simply encrypt and write it back, then set that page’s dirty bit; When the node is a new node, there are two different cases. If the page is in MBuffer, we encrypt and write it back similar as an existing node. If the page is not in MBuffer, we compare the pageID with a diskMaxPageID in order to know whether the page already exists. If so, it has to be first reloaded to avoid losing other nodes on that page.

6 EVALUATION

In this section, we evaluate the performance of Enclave against several baseline approaches. We show that Enclave outperforms baselines on both efficiency and storage savings, and it brings in acceptable overhead compared with vanilla plaintext storage.

6.1 Experimental Setup

Environment. We run experiments on a machine consisting of an Intel(R) Xeon(R) CPU E3-1270 v6 with 3.80GHz processor, which has 4 physical cores (8 logical threads in total). The machine has 64GB RAM capacity, and runs Red Hat 6.4.0 with Linux 4.135 kernel. The processor has 32KB data and instruction caches, 256KB L2 cache, and 8MB L3 cache. Moreover, our implementation is based on Intel SGX SDK 2.6.

Workloads. We use YCSB [20] to generate six core workloads with a zipfian distribution of skewness 0.99, each of which contains 10 million key-value pairs (i.e., records in the table store). For each record, its entire length is 128B and its index key length is 8B.

Baselines. For Enclave Index, we implement two variants of item-based encryption indexes as baselines. The first one (called Baseline) is the implementation of a common design used by Azure Always-encrypted [5] and StealthDB [61] (Section 3.3). The other one (called ItemEnc) puts the execution of Baseline entirely in enclave. In other words, the index structure is still maintained in host memory, but costly enclave interactions are eliminated. For Enclave Store, we implement two baselines with different encryption granularities (Section 3.4.1). The first one (called Item-level) manages an unmodified heap file containing encrypted records as used in [5, 61]. The second one (called Page-level) manages a heap file containing encrypted pages. Our Enclave Store adopts delta decryption on top of Page-level and is called Delta-enc.

Default parameters. There are a number of parameters that are set to default values if not otherwise specified. The default settings are: ItemEnc size is 80MB, ItemEnc size is 350MB for Enclave Index and 2GB for Enclave Store (i.e., large enough to hold the entire file in host memory), node size is 1KB, page size is 4KB, and the rid length is 8B. Note that all experiments are performed in a single-thread setting, and the scalability issue is discussed in Section 4.5.

6.2 Performance Evaluation for Indexes

We first evaluate the performance of Enclave Index, and demonstrate how its efficiency is affected by different factors.

6.2.1 Overall Performance. There exist two phases (i.e., load and run) in the YCSB benchmark. During the load phase, there are only Put operations to construct the storage. We conduct the test with ECalls in both switch and switchless mode.

Switch mode. Different operations are executed according to corresponding workloads in the run phase. As shown in Figure 4(a), Enclave Index achieves about 100 Kops/s, and outperforms Baseline (20.04×) and ItemEnc (5.34×). It is because that Enclave Index only needs one ECall during each operation, and each accessed node is encrypted at most once only when it is not in EBuffer. In contrast, Baseline invokes an ECall for each key comparison, accompanied by decrypting two keys that need to be operated on. This results in a huge performance degradation. Similarly, frequent key decryption
is the main reason for the poor performance in ItemEnc. Moreover, item-grained encryption leads to fewer keys per node, and then stretches the tree height.

**Switchless mode.** Compared to Baseline in the switchless mode, Enclage Index also achieves better performance in throughput under all workloads. As shown in Figure 4(b), Enclage Index achieves more than 182 Kops/s, and outperforms Baseline (13.19x) and ItemEnc (9.69x). The more frequent the EC all is invoked, the greater the performance gain from the switchless mode. Baseline executes massive Ecalls for key comparisons, and hence yields greater performance improvement (3.18x) than Enclage Index (2.2x) and ItemEnc (1.1x). We also evaluate the performance on uniform workloads in Figure 4(c). Enclage Index sustains superior performance compared to Baseline (10.85x) and ItemEnc (8.32x), but the performance gap slightly narrows compared to zipfian distribution in Figure 4(b). Furthermore, we compare Enclage Index with ShieldStore [40], which is a SGX-based in-memory key-value store. Note that ShieldStore is hash-based scheme and therefore cannot support range queries. As can be seen, the throughput of Enclage Index is 7-12x higher than that of ShieldStore. It is because ShieldStore has to decrypt entries in the target hash bucket one by one during index lookup, requiring costly decryption on many small items.

**Skewness.** We evaluate the impact of workload skewness in Figure 4(d). We generate Workload A following a zipfian distribution with varying skewness (from 0.5 to 0.99). As can be seen, the performance of Baseline and ItemEnc are almost unaffected, as they have to decrypt data for each access. In contrast, Enclage Index performs better with higher skewness, due to its EBuffer design.

**Latency.** Table 3 shows the response time of Enclage Index on Workload A with both switch and switchless modes. With the help of switchless mode, the average latency is improved by 51.8%, while the 99-percentile latency is improved by 37.5%.

6.2.2 EBuffer Size. Recall that when the EPC usage exceeds the limit, its page swapping will cause significant performance penalty. Figure 5(a) shows the overall throughput with increasing EBuffer size, which achieves the best performance at the size of 80MB. When the size keeps growing, the throughput degrades rapidly, due to huge page swapping overheads. When the size reaches 100MB, the throughput of Enclage Index is about 2.42x worse than that at 80MB. Hence, we set 80MB as our default EBuffer size.

6.2.3 EBuffer Hit Ratio. Figure 5(b) shows the EBuffer hit ratio, which increases steadily from 0.8 to 0.9 as the EBuffer size increases.
Table 4: Space consumption of indexes.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (GB)</td>
<td>1.34</td>
<td>1.34</td>
<td>1.34</td>
<td>1.41</td>
<td>1.41</td>
<td>1.34</td>
</tr>
<tr>
<td>Enclave Index (GB)</td>
<td>0.23</td>
<td>0.23</td>
<td>0.23</td>
<td>0.24</td>
<td>0.24</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 5: Performance penalty from integrity protection.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>W/O integrity</td>
<td>206.92</td>
<td>225.11</td>
<td>231.72</td>
<td>253.88</td>
<td>182.99</td>
<td>252.83</td>
</tr>
<tr>
<td>With integrity</td>
<td>192.83</td>
<td>214.67</td>
<td>218.84</td>
<td>241.46</td>
<td>176.67</td>
<td>241.37</td>
</tr>
</tbody>
</table>

SGX-disabled called Encrypted B⁺-tree and a vanilla B⁺-tree. As can be seen, there is about 33% degradation from encryption and 30% from SGX. Note that when the caller (e.g., query execution engine) of Enclave is also in enclave, the dominating OCall cost from SGX can be eliminated. In this case, Enclave Index retains 67% of B⁺-tree’s performance, while effectively ensures the confidentiality of sensitive data hosted on an untrusted database server.

6.2.10 Overhead of Integrity Protection. We also evaluate the overhead of the optional integrity protection in Enclave Index. The corresponding performance degradation is shown in Table 5. As can be seen, the integrity protection comes with an average performance loss of 5%. It is mainly from two processes: verifying the digest when a node is loaded to EBuffer; re-calculating the digest when a node is written back to MBuffer.

6.2.11 Scalability. Figure 9 presents the multi-core scalability of Enclave Index in switch and switchless modes with uniform YCSB workloads. In particular, our SGX-supported machines have only 4 physical cores (i.e., 8 logical cores) in total. In the switch mode as shown in Figure 9(a), the throughput improves almost linearly from 1 to 4 threads, then slows down from 4 to 8 threads due to resource contention, and eventually gets saturated. In the switchless mode, we assign 1 untrusted worker (i.e., worker for 0Cal1 thread and 4 trusted worker (i.e., twoker for ECal1 threads). This setting gets the best performance among all combinations in our test. As shown in Figure 9(b), the throughput reaches its peak when there are 4 caller threads (with full 800% CPU utilization). After that, the performance degrades because more caller threads will busy wait for enclave responses, consuming extra CPU resources.

6.3 Performance Evaluation for Table Store

We then evaluate the overall performance of the complete Enclave, i.e., plugging-in Enclave Store beneath Enclave Index. Considering security and performance tradeoffs, we adopt the delta decryption protocol in Enclave Store and call this version of Enclave as Delta-enc. In addition, we implement two baselines (i.e., Item-level and Page-level) as explained in Section 6.1.

6.3.1 EBuffer Size for Table Store. According to our result in Section 6.2.2, the optimal size of EBuffer is 80MB. It is only true when
there is only one enclave and its memory is exclusively used by the index. Here we investigate how the enclave memory should be assigned when both index and table store exist. First of all, we assume that there exist 4 enclaves at the same time, each with 20MB of available memory. We evaluate the overall performance by re-sizing the $E_{Buffer}$ size for index, while fixing the enclave memory usage (i.e., 20MB in total). When the $E_{Buffer}$ is 20MB, the memory footprint for table store is only two pages: one active page called $W_{Page}$ for $T_{Put}$; and another page called $R_{Page}$ for $T_{Get}$. In this case, each $T_{Get}$ requires to fetch the data page from $M_{Buffer}$. When $E_{Buffer}$ shrinks, more data pages can be cached in enclave. As shown in Figure 10(a), the optimal system performance is achieved when the $E_{Buffer}$ is 20MB in all workloads. It is because that the data locality in data pages is lower than that in index nodes. Hence, to achieve high system performance, it is better to allocate most of the enclave memory to Enclave Index, and it is enough to leave two pages of enclave memory to Enclave Store.

6.3.2 Overall Performance. Figure 10(b) shows the overall throughput of Enclave with three Enclave Store variants. Page-level performs the worst. It is because every time a access miss (i.e., the desired page is not in the enclave) occurs, Page-level has to load and decrypt the desired page, which is inherently costly. In contrast, the throughput of Delta-enc is about 1.40× that of Page-level. It is because when a access miss occurs, Delta-enc can extract the desired record without decrypting the entire page. As can be seen, Item-level achieves the best performance, which is about 1.57× that of Page-level. It can directly extract the encrypted record from $M_{Buffer}$, where no encryption and decryption are needed. However, it has much larger storage footprint and leaks record identities to the host.

6.3.3 Space Consumption. Different Enclave Store variants bring in different degrees of storage overhead, which also depends on the record size. Table 6 shows their storage consumption with different record sizes under Workload A. Since Delta-enc has the same physical layout as Page-level with no difference in storage consumption, we omit it in the table. Due to the cipher amplification, Item-level introduces significant storage overhead. For example, its space consumption is 2.99× that of Page-level when the record size is 16B. As the record size increases, the space amplification of Item-level is gradually alleviated. When the record size reaches 256B, the space overhead of it is only 1.07× that of Page-level.

### 7 RELATED WORK

Existing encrypted databases in the literature are mainly built on top of either cryptographic primitives or TEE implementations. **Crypto-based encrypted databases.** CryptDB [49] utilizes special cryptographic primitives to support direct operations over ciphertext, e.g., homomorphic encryption [28] for arithmetic operation, searchable encryption [54] for keyword search, and order-preserving encryption [14] for comparison. Arx [47] encrypts data with only provable encryption schemes and utilizes garbled circuit [63] to allow better protection. MONOMI [58] introduces client-server-split query execution protocol to efficiently process analytical queries over encrypted data. However, these systems introduce prohibitive overheads and only support a limited set of operations, which are unsuitable for general-purpose database infrastructures. **TEE-based encrypted databases.** Since TEEs, such as Intel SGX and AMD SEV, are able to protect both confidentiality of data and execution inside the enclave, it drives the emergence of many TEE-based encrypted databases [5, 9, 25, 40, 42, 50, 61, 64]. Some of them provide strong protection to confidentiality. EnclaveDB [50] makes user data completely inaccessible by implementing an immemory storage and query engine inside the enclave, which only allows pre-compiled queries and assumes that all data can fit in the memory. ObliDB [25] hides the access pattern via oblivious query processing for data in both B*-trees and linear arrays. Oblix [42] relies on a combination of novel doubly-oblivious data structures and enclave to construct a search index for encrypted data. These designs usually lead to limited capacity or prohibitive overheads, making them less practical. In contrast, other systems provide simple-but-weak protection to confidentiality. Speicher [9] and ShieldStore [40] are key-value stores that place data outside enclave with record-grained encryption and integrity checking. Always-encrypted [5] and StealthDB [61], provide confidentiality as add-on features to legacy database systems, using a few enclave-based small functions (e.g., <, >, and =) for computation over ciphertext. However, such non-intrusive design leads to severe information leakage and performance degradation. In summary, there still remains a large unexplored area between above two extreme scenarios, i.e., how confidentiality can be achieved in different ways with practical trade-offs among security, performance and functionality.

### 8 CONCLUSION

Though trusted execution environments provide a powerful building block to construct encrypted databases, practical designs of TEE-based encrypted databases have not been well explored. We provide a comprehensive exploration of possible design choices for building an enclave-based encrypted database storage, and discuss how these choices affect security, performance and functionality. We then propose Enclave, an enclave-native storage engine that makes practical trade-offs under this design space. It contains a B*-tree-like index structure and a heap-file-like table store, and leverages many enclave-friendly designs to offer both high-level security guarantee and good performance. Many designs adopted in Enclave are generally applicable for encrypted storage even when other design choices are made. From extensive experimental evaluation, we observe that Enclave significantly outperforms state-of-the-art approaches in real-world encrypted databases. It improves the throughput by 13× and the storage efficiency by 5×.


