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 *Enclage*, 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 *Enclage* outperforms the baseline, a common storage design in many encrypted databases, by over 13× in throughput and about 5× in storage savings.

PVLDB Reference Format:
Yuanyuan Sun, Sheng Wang, Huorong Li, Feifei Li. Building Enclave-Native Storage Engines for Practical Encrypted Databases. PVLDB, 14(6): 1019-1032, 2021.
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 environment*s (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 ObliDB [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.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447705

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 13× in throughput and about 5× in storage savings.

2 BACKGROUND

In this section, we brief 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 (`OCall`). The cost of `ECall`/`OCall` is expensive, *i.e.,* about 8000 cycles as previously reported [45, 62]. Due to recent patches on SGX driver and microcode, we observe that this cost reaches 15000 cycles. SGX now supports a switchless mode for relatively efficient (asynchronous) calls [34, 57]. Nevertheless, frequent context switches between the enclave and host must be prevented. This raises a challenge to existing database kernels that are unaware of this issue.

Dilemma of TCB size. The size of trusted computing base (TCB) is always a issue in TEEs. In the context of SGX, it includes the processor, microcode and loaded library. Any vulnerabilities in the loaded code will compromise the security of the entire system. Hence, reducing TCB size (*i.e.,* codebase size) makes it more robust, where an exhaustive examination of codebase becomes feasible. However, this inherently leads to frequent context switches between enclave and host (degrading performance as discussed above), and leaves most of the execution logic unprotected (suffering from leakage-abuse attacks [18, 29]). Enlarging the TCB size resolves these issues. However, importing complex database functionality into the enclave inherits all vulnerabilities from the original codebase. It results in a cumbersome patching process and makes TCB less reusable by different database implementations. Hence, the choice of TCB size is extremely critical.

3 ENCLAVE-BASED ENCRYPTED STORAGE

In this section, we first define our data model and threat model, and then introduce a strawman design [5, 61] as a stepping stone to the subsequent design space exploration. At last, we discuss different design choices categorized into five dimensions.

3.1 Data Model and Abstraction

3.1.1 Data model. We consider a relational data model, where a table T contains n columns (*i.e.,* attributes) $COL = \{C_0, C_1, ..., C_{n-1}\}$ and has a primary key $PK \in COL$. Data in all columns are considered confidential, such that each value must be encrypted using a *data encryption key* (DEK) before the value can be sent to the database. This DEK will be later provisioned to the enclave when computation over plaintext is needed. For a record (*i.e.,* a row) r, its value v_i stored in column C_i is the ciphertext $\mathtt{E}(v_i)$, where `E(x)` represents the encryption scheme used by the data owner. To ensure high-level security guarantee, encryption schemes that provide indistinguishability under chosen-plaintext attack (IND-CPA) are preferred, *e.g.,* AES CBC/GCM modes. Columns can be encrypted by different DEKs to further strengthen the security. Note that it is feasible to have plaintext columns, but this may cause unexpected information leakage from cross-column correlations [6, 13].

3.1.2 Encrypted Storage Abstraction. Without the loss of generality, we use a simple case to demonstrate functions that should be supported by an encrypted storage engine. Assume there is a single table T with n columns $\{C_0, C_1, ..., C_{n-1}\}$, where $PK = C_0$ and is indexed. We denote a record r as $[\mathtt{E}(k), \mathtt{E}(v_1), ..., \mathtt{E}(v_{n-1})]$, where k, v_i, are values for C_0, C_i, respectively. To ensure confidentiality, data involved in all storage operations must be ciphertext:

- $\mathtt{Put}(r)$: insert a row to T if its key k does not exist.
- $\mathtt{Get(E}(k)) \rightarrow r$: get the row from T given a key k.
- $\mathtt{Update}(r)$: update the row from T given a key k.
- $\mathtt{Delete(E}(k))$: delete the row from T given a key k.
- $\mathtt{RangeScan(E}(k_1)\mathtt{,E}(k_2)) \rightarrow \{r_i, ..., r_j\}$: retrieve all rows from T whose keys are between k_1 and k_2.
- $\mathtt{FullScan()} \rightarrow \{r_1, r_2, ..., r_m\}$: retrieve all rows from T.

Note that retrievals 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 *rid*s 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 $\mathtt{OE}(rid)$, which is either *rid* or $\mathtt{E}(rid)$. To support storage operations above, the table store TS has to support:

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

The primary index PI has to support:

- $\mathtt{IInsert(E}(k)\mathtt{,OE}(rid))$: insert an entry with key k to PI.
- $\mathtt{ISearch(E}(k)) \rightarrow \mathtt{OE}(rid)$: search the entry with key k.
- $\mathtt{IReplace(E}(k)\mathtt{,OE}(rid))$: replace the entry with key k.
- $\mathtt{IRemove(E}(k))$: remove the entry from PI with key k.
- $\mathtt{IRangeSearch(E}(k_1)\mathtt{,E}(k_2)) \rightarrow \{rid_i, ..., rid_j\}$: 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 of more design choices (Section 3.4). It already covers the case for clustered indexes, *i.e.,* we can replace *rid*s 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 data integrity and protection 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 *rid*s, are in plaintext. Its logical semantics and representations remain unchanged, *i.e.,* index keys are still ordered by their corresponding plaintext. To support index lookups 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 passes 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 *rid*s to retrieve records.

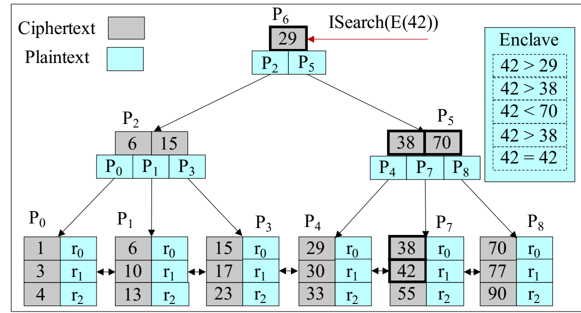


Figure 1: B$^+$-tree with encrypted keys. Node ids (p_0-p_8) and record ids (r_0-r_2) are plaintext.

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 8× 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 *Enclage Index* are bolded and the choices for *Enclage Store* are tagged with an asterisk mark (*).

Design Dimension	Design Choice	Influence		
		Security (Information Leakage)	Performance	Functionality
Encryption Granularity	item-level encryption	leak structural information	high storage overhead; fast for a single read	can fetch data w/o enclave
	page-level encryption *	leak data volume only	low storage overhead; fast for batched small reads	all data access must be in enclave
Execution Logic in Enclave	index: key comparison	leak key ordering and search path	low performance from massive `ECalls`	can split or merge node w/o enclave
	index: index node access	leak node-level search path	high performance from a few `ECalls`	all index access must be in enclave
	table: none	leak record-level identity and location	high performance from no `ECall`	can fetch or scan record(s) w/o enclave
	table: data page access *	leak page-level identity and location	medium performance from a few `ECalls`	all record access must be in enclave
Memory Access Granularity	item-level access *	leak item-level access pattern	high performance from on-demand read	require small footprint in enclave
	page-level access	leak page-level access pattern	moderate performance from page copy	require large footprint in enclave
Enclave Memory Usage	minimum usage *	no additional access protection	low performance from active data fetching	no EPC capacity requirement
	fixed usage	hide a few frequently accessed items	medium performance from data caching	low EPC capacity requirement
	proportional usage	hide many frequently accessed items	medium performance from data caching	high EPC capacity requirement
	unlimited usage	hide access to all items	high performance from data caching	high EPC capacity requirement
Record Identity Protection	no action	leak record identity among queries	no influence	no influence
	***rid* encryption** *	hide linkage between *rid* and record	little influence	only useful in some settings
	ciphertext re-encryption *	hide cipher identity among queries	little influence	only useful in some settings

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 *rid*s 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 *rid*s 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.

3.4.6 Discussion of Combinations' Effectiveness and Information Leakage. We have discussed many choices on five design dimensions, but not all their combinations are valid or equally useful in practice. For example, a table store with *none in enclave* cannot work when *page-level encryption* is chosen. From the perspective of information leakage, the overall leakage depends on the weakest aspects in all dimensions, as listed in Table 1. For example, *ciphertext re-encryption* becomes useless if *key comparison in enclave* is chosen, as the index lookup process already leaks *rid* identity. How to exploit leaked information to conduct leakage abuse attacks has been well studied [13, 18, 29, 49]. We leave the evaluation of exhaustive combinations to the database practitioners, who can assemble choices that best fit for their own needs.

4 DESIGN OF *ENCLAGE*

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

4.1 Design Choices Made in *Enclage*

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 *Enclage* for practical usage. *Enclage* makes different choices for its index and table store, as marked in Table 1. In a nutshell, *Enclage* 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, *Enclage Index* and *Enclage Store* make distinct choices on other design dimensions. For *Enclage Index* (Section 4.2), it puts *index node access* in enclave to minimize `ECalls` 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 *Enclage Index* nodes with *fixed EPC usage.* For *Enclage 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, *Enclage Store* prefers to not cache data pages in enclave, *i.e.,* with *minimal EPC usage*, conceding most EPC memory to *Enclage 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 *Enclage*, and we enable them by default. *Enclage* hence achieves indistinguishability for entry/record access inside a page.

4.2 *Enclage Index*

4.2.1 Overview. Figure 2 shows the overall design of *Enclage Index*. It adopts a three-tier storage hierarchy, including a *EBuffer* tier, a *MBuffer* tier and an external storage tier. As *Enclage 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

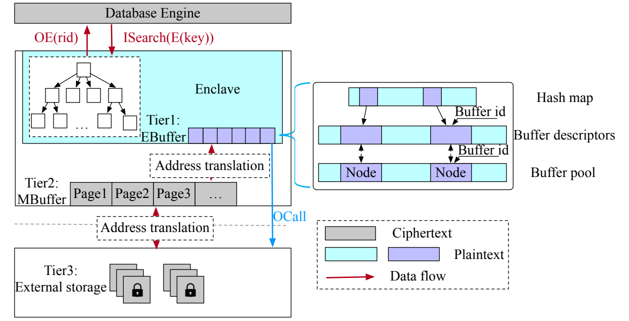


Figure 2: Three-tier architecture of *Enclage Index*.

contains a single node, a page in *Enclage 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 `OCall`. 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. *Enclage 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 *Enclage*, we leave most of EPC quota to *EBuffer* and minimize EPC usage from other components, such as *Enclage 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.

Table 2: Major notations used in cost model.

Notation	Description	Notation	Description (in bytes)
h	tree height	p	node size
N	total # of indexed keys	L_{meta}	node metadata length
n_e	total # of entries in *EBuffer*	L_{key}	key length
n_m	total # of entries in *MBuffer*	L_{nid}	nodeID length
fo_l	max fanout of leaf node	L_{rid}	*rid* length
fo_i	max fanout of internal node	S_e	*EBuffer* size

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 *Enclage 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 `OCalls`. *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 `OCalls`. 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 index node size affects many aspects of *Enclage 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 *Enclage Index*. 4) *Write-back probability* - we assume that when a node is evicted from *EBuffer*, the write-back probability is a constant r_{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 $n_e = \frac{S_e}{p}$. We assume that *rid* and nodeID (*nid*) have the same length $L_{rid} = L_{nid}$ (unified as L_{id}), resulting in $fo_i = fo_l$ (unified as fo). In addition, since L_{meta} is usually much less than p, we have

$$fo \approx \frac{p}{L_{key} + L_{id}}. \quad (1)$$

In a B$^+$-tree, each node has at most fo and at least $\lceil \frac{fo}{2} \rceil$ entries. Assume that the tree height is h and each node contains $\lceil \frac{fo}{2} \rceil$ keys, which is the worst case. We say that the root is at 1-st level, and leaves are at h-th level. For i-th level, the number of nodes num_i is

$$num_i = \lceil \frac{fo}{2} \rceil^{i-1} (1 < i \leq h). \quad (2)$$

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

$$num_{[1,h-2]} = 1 + \lceil \frac{fo}{2} \rceil + \cdots + \lceil \frac{fo}{2} \rceil^{h-3} = \frac{\lceil \frac{fo}{2} \rceil^{h-2} - 1}{\lceil \frac{fo}{2} \rceil - 1}, \quad (3)$$

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

$$num_{[h-1,h]} = \lceil \frac{fo}{2} \rceil^{h-2} + \lceil \frac{fo}{2} \rceil^{h-1}. \quad (4)$$

Since each leaf node contains $\lceil \frac{fo}{2} \rceil$ entries, we have $\lceil \frac{fo}{2} \rceil^{h} \geq N$, and hence $h = \lceil log_{\frac{fo}{2}} N \rceil$.

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 $n_e - num_{[1,h-2]}$. With uniform accesses, the probability of an *EBuffer* miss r_{miss} is

$$r_{miss} \approx 1 - \frac{n_e - num_{[1,h-2]}}{num_{[h-1,h]}}. \quad (5)$$

For each index lookup, the expected number of *EBuffer* misses is $E_{miss} = 2 \cdot r_{miss}$ (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_{Ehash}. 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 $T_{Echain} \cdot n_e$. The *EBuffer* lookup cost per node is hence $C_{EBuffer} = T_{Echain} \cdot n_e + T_{Ehash}$, where two Ts 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_{Mchain} and T_{Mhash}, *i.e.,* $C_{MBuffer} = T_{Mchain} \cdot n_m + T_{Mhash}$. For a query, the expected number of *MBuffer* access is equal to *EBuffer* misses E_{miss}.

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

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

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

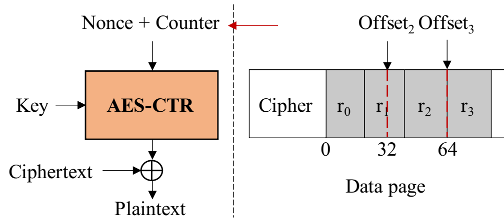


Figure 3: The delta encryption protocol.

single key comparison is T_{cmp}. With binary search, the cost of key comparisons per node is $C_{cmp} = T_{cmp} \cdot \lg(\lceil \frac{fo}{2} \rceil)$.

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

$$C_{total} = h \cdot (C_{EBuffer} + C_{cmp}) + E_{miss} \cdot (C_{MBuffer} + r_{back} \cdot C_{enc} + C_{dec}) \quad (6)$$

In our setting, the constant values are: $S_e = 80MB$, $N = 10M$, $L_{key} = 8B$, $L_{meta} = 24B$, and $L_{id} = 8B$. 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_{key} is the most dominant factor that affects the choice of p. For example, when L_{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 *rid*. After that, a <key, *rid*> pair is inserted to *Enclage Index*. This *rid* 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 *Enclage 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, to retrieve a record, we have to load the entire page into enclave, decrypt it and extract the target record. To reduce the record exaction 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 top of 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_i becomes $\lfloor offset_i/16 \rfloor$. For example, the counter for $Offset_3$ 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-wisely 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 *EBuffer*-cached 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 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 $Y \in [0, X)$ is the node ranking in that page. For *MBuffer* and external storage, we have $physicalAddress = pageID * pageSize + offset$.

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 *Enclage* against several baseline approaches. We show that *Enclage* 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 4.9.135 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 *Enclage 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 *Enclage 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 *Enclage Store* adopts delta decryption on top of `Page-level` and is called `Delta-enc`.

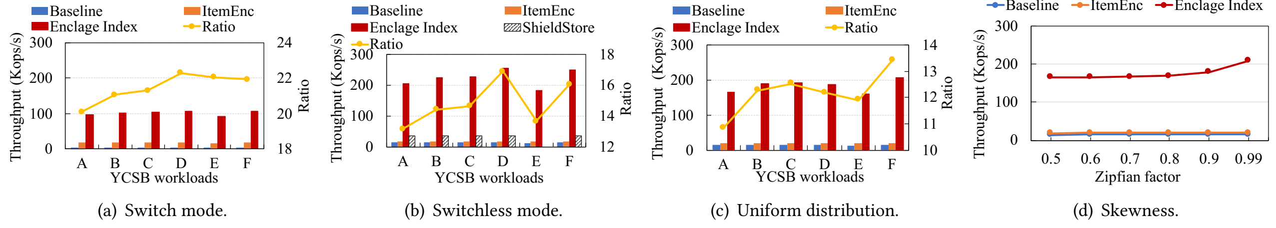
Default parameters. There are a number of parameters that are set to default values if not otherwise specified. The default settings are: *EBuffer* size is 80MB, *MBuffer* size is 350MB for *Enclage Index* and 2GB for *Enclage Store* (*i.e.,* large enough to hold the entire file in host memory), node 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 *Enclage 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), *Enclage Index* achieves about 100 Kops/s, and outperforms `Baseline` (20.04×) and `ItemEnc` (5.34×). It is because that *Enclage 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



(a) Switch mode. (b) Switchless mode. (c) Uniform distribution. (d) Skewness.

Figure 4: Overall throughput in different modes.

Table 3: The response time in different modes.

Enclage Index (μs)	Avg latency	P50 latency	P95 latency	P99 latency
Switch mode	10.44	10	14	16
Switchless mode	5.03	4	9	10

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.

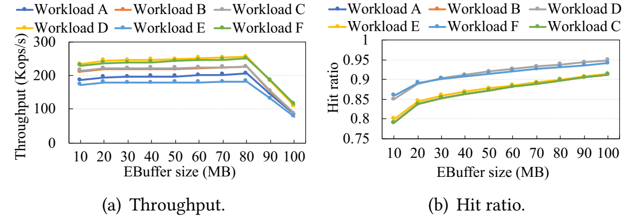
Switchless mode. Compared to `Baseline` in the *switchless mode*, *Enclage Index* also achieves better performance in throughput under all workloads. As shown in Figure 4(b), *Enclage Index* achieves more than 182 Kops/s, and outperforms `Baseline` (13.19×) and `ItemEnc` (9.69×). The more frequent the `ECall` is invoked, the greater the performance gain from the *switchless mode.* `Baseline` executes massive `ECalls` for key comparisons, and hence yields greater performance improvement (3.18×) than *Enclage Index* (2.2×) and `ItemEnc` (1.1×). We also evaluate the performance on uniform workloads in Figure 4(c). *Enclage Index* sustains superior performance compared to `Baseline` (10.85×) and `ItemEnc` (8.32×), 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 7-12× 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%.

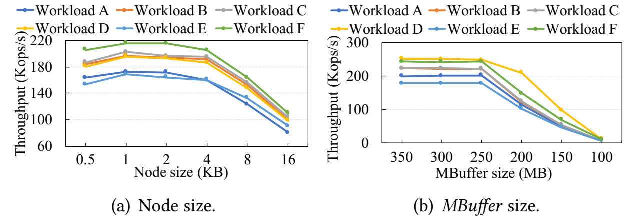
6.2.2 EBuffer Size. Recall that when the EPC usage exceeds the limit, its page swapping will cause significant performance penalty. Figure 5(a) shows the overall throughput with increasing *EBuffer* size, which achieves the best performance at the size of 80MB. When the size keeps growing, the throughput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



(a) Throughput. (b) Hit ratio.

Figure 5: The impact of *EBuffer* size.



(a) Node size. (b) *MBuffer* size.

Figure 6: The impact of node and *MBuffer* sizes.

The initial hit ratio is already quite high. This is because 10MB of enclave memory is more than enough to accommodate most internal nodes in *EBuffer*. We observe that at most two node replacements are needed during an index operation, and hence we take it as one assumption of our theoretical analysis in Section 4.3.1.

6.2.4 Node Size. Figure 6(a) shows how the node size affects the throughput of the entire index. In this experiment, YCSB workloads follow a uniform distribution, which is consistent with the assumption in Section 4.3.1, and the fixed page size is 16KB. As can be seen, the throughput of *Enclage Index* first improves as node size grows, and then begins to decline after reaching its peak at 1KB. It is because that the node size has complex impact on many aspects of the index, such as encryption/decryption cost and *EBuffer* hit ratio. Intuitively, as node size increases, the node decryption cost grows as well, but its amortized cost on each item can either decrease (when many items are accessed together) or increase (when only a few items are accessed). Hence, the optimal choice is determined by the combination of many factors, and 1KB is consistent with the findings from our cost model built in Section 4.3.3.

6.2.5 MBuffer Size. It directly reflects the impact of disk I/Os. As shown in Figure 6(b), when the size is still larger than 250MB, the entire index file is cached in memory. As the *MBuffer* size keeps reducing, the I/O cost rapidly dominates the overall performance, due to frequent page loading from the external storage. That is to say, for a I/O heavy workload, any overhead of in-memory computation (*e.g.,* enclave execution in our case) becomes marginal.

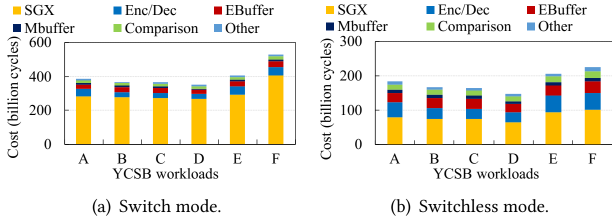
6.2.6 Benefits of Optimization. Figure 8(a) demonstrates the effectiveness of *Enclage Index* optimization introduced in Section 4.2.2.

Table 4: Space consumption of indexes.

	A	B	C	D	E	F
Baseline (GB)	1.34	1.34	1.34	1.41	1.41	1.34
Enclage Index (GB)	0.23	0.23	0.23	0.24	0.24	0.23

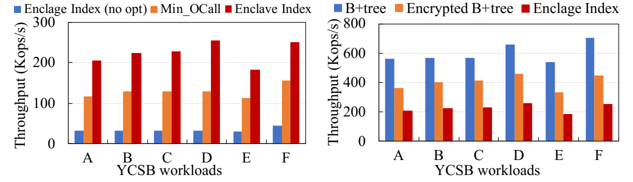
Table 5: Performance penalty from integrity protection.

(Kops/s)	A	B	C	D	E	F
W/O integrity	206.92	225.11	231.72	253.88	182.99	252.83
With integrity	192.83	214.67	218.44	241.46	176.67	241.37
Degradation ratio	7%	5%	6%	5%	3%	5%



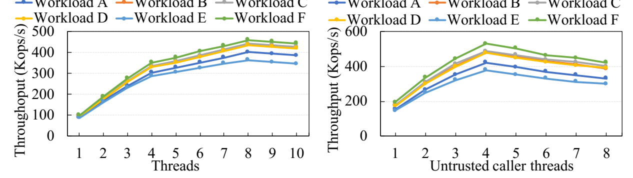
(a) Switch mode. (b) Switchless mode.

Figure 7: Break-down cost of index operations.



(a) Optimization benefit. (b) Performance penalty.

Figure 8: Optimization benefits and performance penalty.



(a) Switch mode. (b) Switchless mode.

Figure 9: Scalability to multiple cores.

In particular, we take advantage of enclave memory to speed up index lookup while carefully avoiding EPC swaps. We also eliminate unnecessary `OCalls` when accessing *MBuffer*. In this test, the minimum *EBuffer* size is set to 5KB (*i.e.,* 5 nodes). In the figure, `Enclage Index (no opt)` indicates the vanilla *Enclage Index* without any optimizations; `Min_OCall` is the variant that avoids unnecessary `OCalls`; *Enclage Index* is the default variant that additionally utilizes large enclave memory (*i.e.,* 80MB) on top of `Min_OCall`. As can be seen, `Min_OCall` achieves a 3.74× improvement in throughput and *Enclage Index* gains an additional 1.74× improvement.

6.2.7 Break-down Cost. Figure 7 shows the break-down cost of major functions in *Enclage Index*. We accumulate all CPU cycles consumed by corresponding functions during the workload execution. *SGX* is the cost of the enclave environment, mainly from enclave interactions (*i.e.,* `ECall/OCall`); *EBuffer* is the cost to maintain the buffer manager of *EBuffer*; *MBuffer* is the cost to maintain the buffer manager of *MBuffer*; *Enc/Dec* is the total cost of encryption and decryption, which includes the cost to encrypt nodes when writing them back to *MBuffer*, to decrypt nodes when loading them into *EBuffer* from *MBuffer*, and to decrypt input keys before executing corresponding operations; *Comparison* is the cost to execute key comparisons over search paths from the root to the target leaf node; and *Other* is all other miscellaneous cost, such as LRU maintenance, data copy, pageID assignment. As can be observed, the cost under switch mode is mainly dominated by *SGX*, which is more than 72%. This cost is reduced to about 43% under switchless mode.

6.2.8 Space Consumption. We measure the size of index files to compare their space consumption. As shown in Table 4, we observe that the file size of *Enclage Index* is about 0.23GB and the one `Baseline` is 1.40GB, achieving a 5.76× reduction. It is because that `Baseline` has to maintain cipher metadata for each key or *rid*, which causes huge space amplification. In contrast, *Enclage Index* only needs per-node metadata, which becomes negligible. Note that there are encryption schemes that has no space amplification, such as AES ECB mode, but they are insecure for practical usage.

6.2.9 Performance Penalty. Figure 8(b) evaluates the performance penalty from both SGX and encryption. We implement two *Enclage Index* variants without SGX environment, *i.e., Enclage Index* with 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 *Enclage* is also in enclave, the dominating `OCall` cost from SGX can be eliminated. In this case, *Enclage 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 *Enclage 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 *Enclage 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.,* uworker for `OCall`) thread and 4 trusted worker (*i.e.,* tworker for `ECall`) 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.

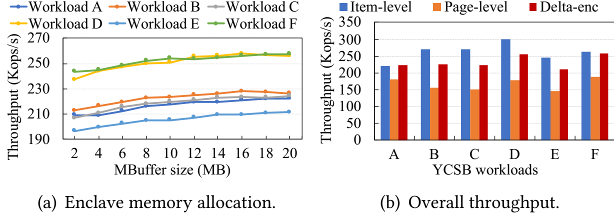
6.3 Performance Evaluation for Table Store

We then evaluate the overall performance of the complete *Enclage*, *i.e.,* plugging-in *Enclage Store* beneath *Enclage Index*. Considering security and performance tradeoffs, we adopt the delta decryption protocol in *Enclage Store* and call this version of *Enclage* 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

Table 6: Space overhead with different encryption protocols.

	16B	32B	64B	128B	256B
`Item-level` (GB)	0.67	0.89	1.36	2.29	4.09
`Page-level` (GB)	0.23	0.45	0.91	1.85	3.81
Ratio	2.99	1.98	1.50	1.24	1.07



(a) Enclave memory allocation. (b) Overall throughput.

Figure 10: The impact of encryption protocols.

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 *EBuffer* size for index, while fixing the enclave memory usage (*i.e.,* 20MB in total). When the *EBuffer* is 20MB, the memory footprint for table store is only two pages: one active page called *WPage* for `TPut`; and another page called *RPage* for `TGet`. In this case, each `TGet` requires to fetch the data page from *MBuffer*. When *EBuffer* shrinks, more data pages can be cached in enclave. As shown in Figure 10(a), the optimal system performance is achieved when the *EBuffer* 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 *Enclage Index*, and it is enough to leave two pages of enclave memory to *Enclage Store*.

6.3.2 Overall Performance. Figure 10(b) shows the overall throughput of *Enclage* with three *Enclage 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 *MBuffer*, 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 *Enclage 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 in-memory 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 *Enclage*, 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 *Enclage* are generally applicable for encrypted storage even when other design choices are made. From extensive experimental evaluation, we observe that *Enclage* significantly outperforms state-of-the-art approaches in real-world encrypted databases. It improves the throughput by 13× and the storage efficiency by 5×.

REFERENCES

[1] 2008. *MurmurHash*. https://sites.google.com/site/murmurhash/
[2] 2010. *Google fired engineer for privacy breach*. https://www.cnet.com/news/google-fired-engineer-for-privacy-breach/
[3] 2019. *Google Encrypted BigQuery client*. https://github.com/google/encrypted-bigquery-client
[4] 2020. *SEV Secure Encrypted Virtualization API Version 0.24*. https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf
[5] 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.
[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase.. In *CIDR*.
[7] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction Processing on Confidential Data using Cipherbase. In *Proceedings of the IEEE 31st International Conference on Data Engineering (ICDE'15)*. IEEE, 435–446.
[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 689–703.
[9] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, 173–190.
[10] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2013), 752–765.
[11] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. 2012. A Security Analysis of Amazon's Elastic Compute Cloud Service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*. 1427–1434.
[12] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Bradley C Kuszmaul. 2005. Concurrent Cache-Oblivious B-Trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 228–237.
[13] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2018. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1715–1728.
[14] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Proceedings of the Annual Cryptology Conference*. Springer, 578–595.
[15] Anastasia Braginsky and Erez Petrank. 2012. A Lock-Free B+Tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 58–67.
[16] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. 2011. AmazonIA: When Elasticity Snaps Back. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 389–400.
[17] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. 2019. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1087–1099.
[18] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 668–679.
[19] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Proceedings of the Annual Cryptology Conference*. Springer, 353–373.
[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*. 143–154.
[21] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. 2017. The Pyramid Scheme: Oblivious RAM for Trusted Processors. *arXiv preprint arXiv:1712.07882* (2017).
[22] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 86 (2016), 1–118.
[23] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 131–140.
[24] Ramez Elmasri. 2008. *Fundamentals of Database Systems*. Pearson Education India.
[25] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.
[26] Tal Garfinkel and Mendel Rosenblum. 2005. When Virtual Is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HotOS*.
[27] Blaise Gassend, G Edward Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE, 295–306.
[28] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.
[29] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. 2017. Practical Passive Leakage-abuse Attacks Against Symmetric Searchable Encryption. In *Proceedings of the 14th International Conference on Security and Cryptography SECRYPT 2017*. 200–211.
[30] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.
[31] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (2009), 91–98.
[32] Shane V Howley and Jeremy Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 161–171.
[33] Intel. 2014. *Intel(R) Software Guard Extensions Programming Reference*. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
[34] Intel. 2018. *Intel(R) Software Guard Extensions SDK for Linux* OS*. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf
[35] Intel. June 2015. *Intel(R) Software Guard Extensions (Intel SGX)*. https://software.intel.com/sites/default/files/332680-002.pdf
[36] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private Large-Scale Databases with Distributed Searchable Symmetric Encryption. In *Proceedings of the Cryptographers' Track at the RSA Conference*. Springer, 90–107.
[37] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. 2005. Concurrency Control and Recovery for Balanced B-link Trees. *The VLDB journal* 14, 2 (2005), 257–277.
[38] David Kaplan. 2016. AMD x86 Memory Encryption Technologies. In *Proceedings of the 25th USENIX Security Symposium (SEC'16)*.
[39] David Kaplan. 2017. Protecting VM Register State with SEV-ES. *White paper, Feb* (2017).
[40] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the European Conference on Computer Systems (EuroSys'19)*. 1–15.
[41] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic Authenticated Index Structures for Outsourced Databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 121–132.
[42] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.
[43] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 317–328.
[44] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. *A Survey of Published Attacks on Intel SGX*. Technical Report. Tech. rep.
[45] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. 238–253.
[46] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 359–374.
[47] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptology ePrint Archive* 2016 (2016), 591.
[48] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database using Semantically Secure Encryption. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1664–1678.
[49] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 85–100.
[50] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 264–278.
[51] Thomas Ristenpart and Scott Yilek. 2010. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *NDSS*.

[52] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–217.

[53] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[54] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 44–55.

[55] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 299–310.

[56] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annula International Conference on Supercomputing (ICS'03)*. 160–171.

[57] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX'18)*. 22–27.

[58] Stephen Lyle Tu, M Frans Kaashoek, Samuel R Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proceedings of the VLDB Endowment* 6, 5 (2013), 289–300.

[59] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. *http://cacheoutattack. com/* (2020).

[60] Verizon. 2018. *2016 Data Breach Investigations Report*. https://regmedia.co.uk/2016/05/12/dbir_2016.pdf

[61] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. Stealthdb: a scalable encrypted database with full SQL query support. *Proceedings on Privacy Enhancing Technologies* 2019, 3, 370–388.

[62] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 81–93.

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

[64] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 283–298.