Leafblower: a Leakage Attack Against TEE-Based Encrypted Databases

Zachary Espiritu[†], Seny Kamara[*†], Tarik Moataz[†], Valentin Ogier[‡§]
[*]*Brown University* [†]*MongoDB Research* [‡]*NetApp*
{*zachary.espiritu, seny.kamara, tarik.moataz*}*@mongodb.com, contact@vogier.fr*

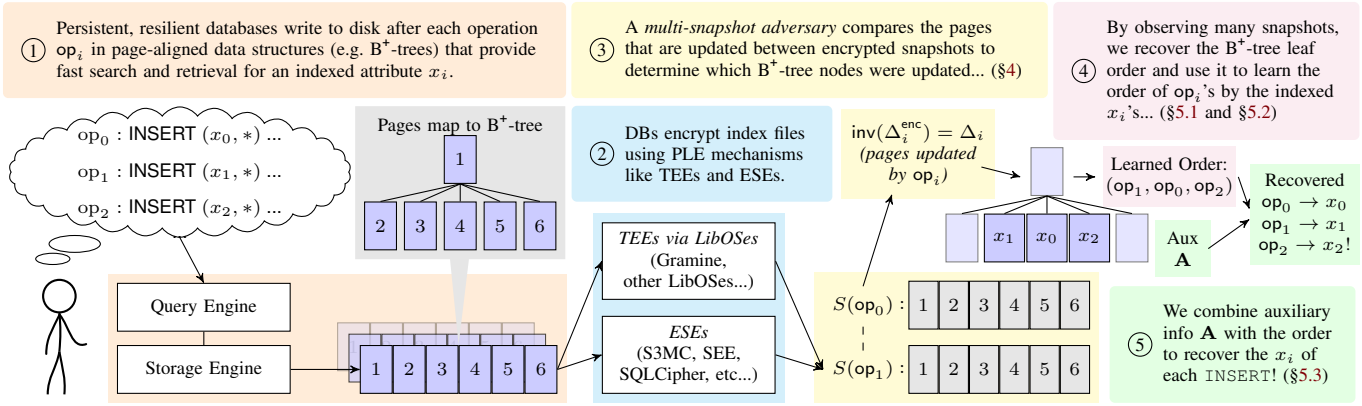


Figure 1: Overview of this paper and the LEAFBLOWER attack.

***Abstract*—Trusted execution environments (TEEs) have emerged as a common solution for database systems to provide encryption in use. Several encrypted databases (EDBs) have been deployed within TEEs using library operating system toolchains that transparently allow existing applications to run within TEEs without modification. This "lift-and-shift" paradigm greatly simplifies the design of EDBs but leaves open questions about the security of the resulting system.**

In this work, we propose a new leakage attack against TEE-based EDBs which use B[+]-trees in the *multi-snapshot external memory model*, a weaker adversary which only observes snapshots of the encrypted database index files after each operation. We show how to approximately order insertions by their inserted value by exploiting the "structural leakage" of the on-disk index format. Then, we leverage auxiliary information to recover the approximate plaintext values of insert operations with significant advantage over a naive adversary that makes guesses based on equivalent auxiliary information. Under optimal conditions—when the auxiliary is accurate and the domain is small—we achieve up to 96% exact recovery in experiments on real-world datasets which increases to 100% when scoped to later operations in the transcript. Our attack requires no injections and no information about read operations.

While our work is primarily motivated by TEE-based encrypted databases, we demonstrate that our attack generalizes to other kinds of *page-level encryption* systems including encrypted storage engines and disaggregated database systems.

§ *Work done while at MongoDB Research.*

***Index Terms*—multi-snapshot, leakage attacks, TEEs**

1. Introduction

Database encryption technologies have significantly evolved due to increasing demands for data security and privacy. Broadly, these technologies can be categorized as:

- *encrypted storage engines* (ESE), which are server-side database components designed to encrypt database files before they are stored on persistent storage media. These engines protect data at rest but typically do not protect data during processing of queries.
- *encrypted databases* (EDB), which encrypt data client-side before it is transmitted to the database server. These systems aim to protect data both at rest and in use and provide broader confidentiality guarantees.

Encrypted databases have garnered significant attention from both the research community and industry. Various methods underpin these technologies such as property-preserving encryption (PPE) [5], [18], [22], structured encryption (STE) [33], [38], and trusted execution environments (TEE) like Intel SGX [37], Intel TDX [35], AMD SEV [82], AMD TrustZone [117], and RISC-V Keystone [95]. PPE enables specific queries on encrypted data by preserving certain plaintext properties, whereas STE provides leakage-minimizing data structures for encrypted data. TEEs offer a hardware-secured environment that encrypts code and memory while still preserving the ability to perform operations on data so that plaintext memory is not directly exposed to other untrusted system components.

TEE-based encrypted databases, in particular, have seen extensive practical deployment. Typically, these systems encrypt data at the client and leverage TEEs on the server to securely execute queries within an isolated execution context. While TEEs can significantly simplify EDB design compared to purely cryptographic solutions, as we will soon explain, their use introduces several complexities when trying to reason about the security of the overall system.

Security analysis of EDBs. Analyzing the security of EDBs is nuanced and complex. Over time, however, a standardized framework for security analysis has emerged, which consists of the following complementary methodologies:

- *leakage specification [33], [38]:* precisely characterizing the leakage of an encrypted database and proving that the scheme reveals no information beyond its stated leakage profile. However, specification alone only assures that the construction leaks nothing beyond the specified profile; it does not address whether the defined leakage is exploitable.
- *leakage attacks [20], [77], [80]:* developing cryptanalytic attacks to extract information from the leakage profiles of the scheme. These attacks concretely identify specific scenarios under which adversaries can realistically compromise confidentiality.
- *leakage analysis [50], [79], [87]:* using formal mathematical frameworks to bound the probability that an adversary can extract sensitive information from the leakage profile.

While each methodology serves a distinct purpose, they are all carried out across the same set of different adversarial models. Two common archetypes include the *persistent* model, where the adversary has access to the encrypted data and a full transcript of operations; and the *snapshot* model, where the adversary can only observe periodic snapshots of encrypted data (in the case of the *multi-snapshot* model, a snapshot after every operation).

Architectures of TEE-based EDBs. The architecture of TEE-based encrypted databases can be broadly categorized into the following groups:

- *hybrid systems:* systems designed to integrate TEEs directly with plaintext database engines. For example, HeDB [100], StealthDB [149] and Microsoft SQL Server's Always Encrypted [9] delegate specific query operations or stages to the TEE, while keeping most database operations external to the secure environment.
- *TEE-native systems:* systems such as CryptSQLite [152], EnclaveDB [123], SecuDB [156], and TrustedDB [14] extensively integrate their query processing and storage engines within the TEE. This architecture maximizes the portion of database functionality protected by the secure environment.
- *lift-and-shift systems:* to minimize the overhead associated with adapting existing databases to TEEs, this category employs *library operating system* (LibOS) frameworks. LibOSes enable existing, unmodified (or minimally modified) applications to operate within TEEs [8], [11], [12], [15], [42], [55], [90], [127], [131].

The conceptual appeal of these designs (especially the lift-and-shift approach) lies in treating the TEE as a secure and isolated environment, intuitively envisioned as a protective "enclave". However, this analogy oversimplifies the actual operational reality in practice.

A new leakage vector. The most obvious reason: as demonstrated by a rich line of work, adversaries that are able to *persistently* compromise the host system (potentially physically) can extract information from TEE-protected memory through various side-channels [112], [147] by abusing microarchitectural details such as memory caches [26], [39], [68], [76], [102], [103], [109], [146], [160], speculative execution [29], [125], [145], memory page metadata [28], [30], [99], [130], [151], [154], and vulnerabilities in specific block cipher constructions for encrypting memory [97], [98], [124], [155], [157], [158]. The adversaries in these works often require kernel-level access to manipulate and/or observe caches, registers, and other microarchitectural components to infer information about the victim program's control-flow or memory access patterns. When kernel access is not required, such works still require the adversary to have the ability to execute code on the same host as the targeted process. Ultimately, the adversaries in prior work must have the ability to persistently compromise various *compute*-related components of the victim machine and/or program.

But such adversaries are not the only risk to EDBs. A more subtle issue arises from a fundamental reality of persistent databases—they eventually must write data to disk. Thus, database indexes created by TEE-based EDBs must eventually reside on untrusted storage (though in an encrypted form). The handling and on-disk format of encrypted data across insecure storage—even assuming the data is processed securely within the TEE—can inadvertently leak sensitive information, potentially weakening the overall security guarantees provided by TEE-based EDBs.

A new leakage attack. In this work, we demonstrate that the use of untrusted storage by TEE-based EDBs can be exploited by a *significantly weaker* adversarial model that has not been used in prior work. Specifically, we formalize a subset of the leakage of a wide class of TEE-based EDBs and exploit it via a new leakage attack under the *multi-snapshot external memory (disk) model*, where the adversary has access to a snapshot of the disk after each operation. This model is motivated by recent industry trends towards designing EDBs that primarily defend against "insider threats", or adversaries that can compromise an account and gain read access to a limited set of database components (here, the storage layer).[1] At the heart of our attack are two key observations: (1) the set of disk pages modified by database insert operations are data-dependent; and (2)

1. There are real-world case studies of adversaries who can repeatedly read snapshots of disk but cannot compromise the compute layer or observe individual disk reads and writes (e.g. 2019 Capital One data breach [85]).

the TEE's resulting modifications to encrypted pages on disk are correlated with these data-dependent page modifications. By leveraging these correlations, our attack ultimately reconstructs the order of insertions. Then, using auxiliary information, we can approximate the inserted values. We describe the attack in more detail in Section 1.1.

Application to other systems. While our attack is primarily motivated by TEE-based EDBs, the use of a weaker adversary model means it also can be used to attack other page-level encryption systems, including:

- *encrypted storage engines*, which implement page-level encryption to protect against server administrators that have access to the mounted filesystem. ESEs are included in Azure SQL [108], InnoDB [113], and WiredTiger [111], or as add-on plugins like SQLCipher [159], SQLite Encryption Extension [137], and SQLite3 Multiple Ciphers [141] for SQLite.
- *disaggregated database systems*, which separate compute, memory, and storage to enable independent scaling in each dimension. The economic and performance benefits of disaggregation have spurred commercial disaggregated databases such as Amazon's Aurora [148], Alibaba's PolarDB [96], Azure SQL Hyperscale [10], Databricks's Neon / disaggregated PostgreSQL [115], Huawei's GaussDB [43], [101], and Turso's disaggregated SQLite [143]. Such systems use cloud storage abstractions like Amazon S3 [25] as their storage layer.

Since ESEs, disaggregated database systems and TEE-based EDBs all either implement or rely on some form of page-level encryption, we refer to them collectively as *page-level encryption (PLE) systems* and to the modules that implement PLE (e.g., an ESE or a LibOS) as PLE modules.

1.1. Contributions

Exploiting page-level leakage in B$^+$-tree-based TEE-backed EDBs. Most analysis of TEEs focuses on recovering secrets from the TEEs via side-channels under adversaries with persistent compromise of the compute-related components of the system. Here, we instead study page-level leakage in the multi-snapshot *disk* model where the adversary has access to a snapshot of the disk after every operation. As described above, this model is *significantly* weaker than the traditional models often used to study cryptographic encrypted databases or typical TEE-based applications. Furthermore, our model is *passive* and does not require the adversary to inject or tamper with any data.

A framework for attacking page-level leakage. We propose a novel framework that allows a multi-snapshot disk adversary to recover the approximate order of inserted operations from page-level leakage. For readability, we focus on one specific database, SQLite [59]. We chose SQLite because: (1) it is the most widely deployed database in existence [69]; (2) it is an example enclave application for many LibOSes; (3) it can be used with a large number of ESEs; and (4) though it may seem simpler than other major databases, its (plaintext) index design introduces unique challenges for our attack that are not present in other databases (discussed in Appendix A). At a high-level, our attack consists of four phases:

1) *inversion:* this step consists of mapping encrypted filesystem blocks to their corresponding plaintext database pages. The goal is to reduce the attack to a setting where the adversary is able to observe the page modifications as they would occur over plaintext.
2) *disambiguation:* given a pattern of plaintext page updates, the attack needs to partition the pages into two sets: the pages corresponding to the records and the pages corresponding to the index. The former are the pages storing the SQLite table, whereas the latter are the pages storing the indexed column. Isolating the index is crucial for our attack, as it is from this structure that we extract information about the values. The techniques and observations employed here primarily relate to algorithmic details of how SQLite's B$^+$-tree index functions, including node splits.
3) *restructuring:* once the pages of the index are identified, the next step is to determine the structure of the tree, which (ideally) allows us to track and map the insertions to their exact pages throughout the construction of the index tree. When two operations belong to different leaves, their order is known; but when they belong to the same leaf, their order remains uncertain. Here, we leverage the behavior of B$^+$-tree *rebalances* to reconstruct the order of leaves and map each inserted value to its corresponding leaf in the tree.
4) *inference:* given the (approximate) order from the previous phase and assuming the existence of auxiliary information, we can then execute an inference attack similar to prior attacks on property-preserving encryption such as the SORTING-ATK by Naveed, Kamara, and Wright [114]. This results in an approximate recovery of the value inserted by each operation.

We remark that under the *persistent* adversarial models typically used to analyze the security of cryptographic EDBs or TEE-based applications, many phases of our attack can be simplified. For instance, folklore B$^+$-tree inference attacks have been mentioned in the TEE literature which leverage the ability of an adversary who can persistently observe memory page tables and can learn the order in which pages are accessed (e.g. [6], [30]). This assumes such an adversary can determine which memory pages correspond to the B$^+$-tree, but, if they can, the order of page accesses immediately reveals the structure of the tree and trivializes the *disambiguation* and *restructuring* steps. Furthermore, these adversaries can recover B$^+$-tree structure from not just *insert* operations, but also *reads*. In comparison, our attack in the multi-snapshot disk model requires a more complex approach as we only learn the set of disk pages updated by each *insert* (without any information on the order the pages were accessed) and learn no information about reads.

Attacking SQLite with various PLE modules. We instantiate and implement our attack against SQLite protected with

various PLE modules, including two SQLite ESEs, SQLCipher Community Edition (SQLC) and SQLite3 Multiple Ciphers (S3MC), and the LibOS Gramine [61] in Intel SGX. We chose Gramine since it is the only actively maintained, open-source LibOS recommended by Intel and is the most widely used [90].

In Section 6, we evaluate our attack against both synthetic and real datasets. In the synthetic case, we sample one million insert operations uniformly at random, where the target attribute belongs to domains of different sizes. We achieve high exact recovery when the domain size is small—for a one-byte integer, the recovery rate reaches up to 88% for operations that occur in the last 25% of the transcript. For larger domain sizes, exact recovery becomes significantly less effective. However, approximate recovery remains consistent regardless of domain size. In particular, we achieve an average absolute error under 0.3% for the last 25% of transcript operations. Depending on the targeted attribute, this level of recovery can still be highly relevant—for instance, in the case of salary information, where knowing the value with precision down to cents may not be necessary. For real datasets, we use star ratings from Amazon products [71], [72] and data from Texas hospital discharges [142]. Similarly, when domain sizes are small and we choose accurate auxiliary information, we achieve up to 96% exact recovery over the whole transcript, increasing to 100% when scoped to the last 25% of the transcript.

Assumptions and limitations. We make two major assumptions in our attack. First, our snapshot adversary requires that each snapshot corresponds to the insertion of exactly 1 record (that is, we do not target workloads that use transactions to insert multiple records at once). Similar assumptions have been used in several recent related works that *additionally* require injections [24], [58], [70]. Second, we assume that no delete or `VACCUM` [140] operations occur. (This assumption is not strictly required, but certain sequences of deletions can add noise to our recovery algorithm so we simplify by not considering any deletions.) We discuss ways of potentially relaxing these restrictions in Section 8.

Responsible disclosure. While our findings are primarily theoretical in nature, we disclosed our work with the developers of SQLC, S3MC, and Gramine (as well as InnoDB, MongoDB, PostgreSQL, and Turso due to Appendix A) on April 14, 2025.

2. Related Work

We have already discussed works that attack TEEs via various microarchitectural side-channels. Here, we focus on a broader class of techniques used to analyze EDBs.

Structural leakage attacks on range ESAs. Our attacks on B^+-trees leverage structural co-occurrences that appear in the multi-snapshot model due to *updates*. The idea of exploiting "structural leakage" is similar to the methodology used by Markatou, Falzon, Espiritu, and Tamassia [104] where a *persistent* adversary exploits structural properties of *read* queries on specific STE-based range indexes [54] to learn how encrypted sub-queries map to the index shape (and from there, the plaintext value of the queries). There are more persistent model attacks that exploit other kinds of range ESA leakage which do not show up in the multi-snapshot model (e.g. [52], [53], [63], [64], [67], [84], [88], [89], [91], [105], [106], [107]).

Leakage attacks on PPE. Naveed, Kamara, and Wright [114] proposed the first attacks on deterministic (DTE) and order-revealing encryption (ORE) [22] which combine the leakage of DTE and ORE with *auxiliary information* to reconstruct the plaintext values. ORE is not used in the systems we consider in this work, but we use similar reconstruction techniques to turn an approximate insert *order* reconstruction into an approximate insert *value* reconstruction. However, since ORE leaks both order *and* frequency (whereas our attack only recovers order information), most prior PPE attack techniques (e.g. [19], [47], [66], [92]) are not directly applicable here. We expand on our work's connections to PPE attacks in Section 5.3.

Other side-channel attacks on databases. Various side-channel attacks specifically target databases (e.g. [40], [57], [65], [78], [83], [126], [129], [150]). The closest works use the *size* of encrypted database files to learn information. Bourassa, Michalevsky, and Eskandarian [24], Hogan, Michalevsky, and Eskandarian [70] and Fábrega, Pérez, Namavari, Nassi, Agarwal, and Ristenpart [58] leverage *compression side-channels* in cloud storage by *injecting* records and measuring changes in storage size to infer information about previous content. Pei and Shmatikov [119] present a case study where the adversary distinguishes between varying lengths of inserted records by measuring the length of write-ahead logs in WiredTiger.

Leakage attacks on TEE-backed databases. We are aware of one prior attack targeted against a specific TEE-backed database: SQL Server's *Always Encrypted* (AE) [9], which includes a indexing scheme for performing sort-based queries on randomized ciphertexts by storing ciphertexts by their sorted plaintext order inside of a B^+-tree. Seah, Khu, Hoover, and Ng [128] describe an attack in which an adversary who has query access to internal tables that reveal the B^+-tree contents can use ORE reconstruction techniques to learn the values of the ciphertexts. (This leakage was previously acknowledged in the original AE paper [9].)

3. Preliminaries

Adversary model. We consider a *n-snapshot external memory (disk) adversary* which sees the encrypted index files written by a database after every insert operation. To make this concrete—persistent databases use explicit `fsync` [74] syscalls to flush potentially buffered writes to disk at the end of every transaction (to ensure that committed data is resilent to unexpected outages), so we say the adversary

learns the contents of the filesystem after every `fsync`. Since an `fsync` happens at the end of each operation, we will simply say that a *snapshot* $S(\mathsf{op}_i)$ is the set of files representing the encrypted database after op_i was performed.

We stress that the adversary does not see internal memory, does not have the ability to perform their own operations on the database, and does not have access to the encryption keys. Additionally, since reads do not affect the indexing structures on disk, we will simplify discussion by assuming that each op_i is an insertion operation.

SQLite. SQLite [59] is an embedded database that uses a B$^+$-tree-based storage engine. It supports page sizes $P = \{4096, 8192, 16384, 32768, 65536\}$, and, in conjunction with the B$^+$-tree index file, uses one of six different journaling schemes (`DELETE`, `TRUNCATE`, `PERSIST`, `WAL`, `MEMORY`, and `OFF`) to allow the database to recover gracefully from unexpected crashes. We focus on the defaults of $P = 4096$ and `DELETE` mode as they are likely the most widely-used, though our attacks work for every mode except for `WAL` mode (as discussed in Appendix A).

B$^+$-trees. SQLite uses a variant of B$^+$-trees [16] credited to a description by Knuth [86]. A B$^+$-tree is a m-ary search tree which maps keys to values. Every interior node stores pointers to at most $w \leq m$ children nodes $(p_1, \ldots, p_w)$ along with $m-2$ keys $(k_1, \ldots, k_{w-2})$. The subtree pointed to by p_1 contains search keys that are less than k_1. For each $i \in [2, w-1]$, the subtree pointed at by p_i contains *divider keys* that are greater than or equal to k_{i-1} and are less than k_i. The subtree pointed to by p_w contains search keys that are greater than or equal to k_{w-2}. The leaf nodes contain no pointers and contain search keys and their associated values. Querying for a key is done by traversing the tree and performing a search at each level until a leaf is reached and the key's associated value is found.

Inserting a new key-value pair starts in the same way as a query—first, we find the leaf to insert the key into—then, we insert the key-value pair in sorted position in the node.[2] If inserting the key would cause the node to reach capacity, we *split* the at-capacity node's keys across two new nodes, then update the parent's pointers and divider keys accordingly. Splits may recursively propagate up the tree if inserting the new child into the parent causes the parent to go over capacity. B$^+$-trees commonly use a *rebalance* optimization that attempts to reduce the number of splits in expectation. Instead of splitting nodes to a new level when capacity is reached, a local rotation occurs where keys from the at-capacity node are flowed into its neighbors. This limits the tree's height and more effectively utilizes the space in each node. We discuss rebalances in detail in Section 5.2.

While the theoretical description of an B$^+$-tree with a fixed number of keys and children is convenient to illustrate examples, practical B$^+$-trees are usually defined in terms of a *node capacity* in bytes. In SQLite, the node capacity is $P - c$ where c is a constant amount of space reserved for node metadata. Then, the pointers and key-value pairs are stored using variable-length encodings. This packs as much data as possible into each node.

2. Modern B$^+$-tree databases treat every key as unique by concatenating a salt or a record identifier to the end of the key, so the insertion algorithm does not behave differently on previously inserted values.

SQLite file layout. A SQLite database's on-disk representation consists of two files: the *main file* and the *rollback journal* [133]. (The journal is not used in our attack so we do not discuss it further.) The main file is logically organized in *logical database pages* of size P and can be viewed as a collection of multiple, interleaved B$^+$-trees. (This is to be distinguished from *filesystem blocks*, which correspond to the storage system's fundamental unit of data storage; here, we assume a block is always 4 KiB.) Each SQL table is associated with one *record tree*, which maps 64-bit record identifiers to row data. Each SQL index is associated with one *index tree* which maps arbitrary keys to record identifiers. In this work, we consider SQL tables with one index. When an insertion occurs, SQLite first performs the corresponding insertions on the B$^+$-trees in memory. Then, SQLite updates the main index file and ensures that the file has been written to persistent storage using `fsync`.

We now describe the representation of a SQLite main file that contains just one table and one index. The file contains 3 pages. Page 0 stores metadata like the schema of the database and the page identifiers of the roots of any B$^+$-trees. Then, the root pages of the table and index are in the order they were defined in the schema. Since the table must be defined before the index, page 1 corresponds to the root of the record tree and page 2 corresponds to the root of the index tree. When new nodes are allocated for either tree, new pages are appended to the end of the file in the order they were requested. This means that the record and the index trees are interleaved with each other as the database grows. This introduces another challenge for our adversary—in addition to learning each tree's structure, the adversary must determine which tree each page belongs to.

4. Warmup: Learning from Pairs of Snapshots

We start with an example (Figure 2) where the adversary already knows the structure of the plaintext B$^+$-tree after some initial operation op_i. Then, consider the following two operations op_{i+1} and op_{i+2} and the encrypted snapshots taken after each operation $S(\mathsf{op}_i)$, $S(\mathsf{op}_{i+1})$, and $S(\mathsf{op}_{i+2})$. What can we learn about op_{i+1} and op_{i+2} from the snapshots? To answer this, we introduce two concepts: *encrypted and plaintext deltas* and *inversion functions*.

Inverting encrypted deltas. The *encrypted delta of* op_{i+1}, denoted $\Delta^{\mathsf{enc}}_{i+1}$, is the set of the encrypted *filesystem block* indices that have changed as a result of executing op_{i+1}. As illustrated in Figure 2, given the pair of snapshots $S(\mathsf{op}_i)$ and $S(\mathsf{op}_{i+1})$, we can easily compute $\Delta^{\mathsf{enc}}_{i+1}$ by pairwise comparing each 4 KiB block in $S(\mathsf{op}_i)$ and $S(\mathsf{op}_{i+1})$. However, what we really want to know is the *plaintext delta of* op_{i+1}, denoted Δ_{i+1}, or the set of indices corresponding to the logical *database pages* that changed as a result of

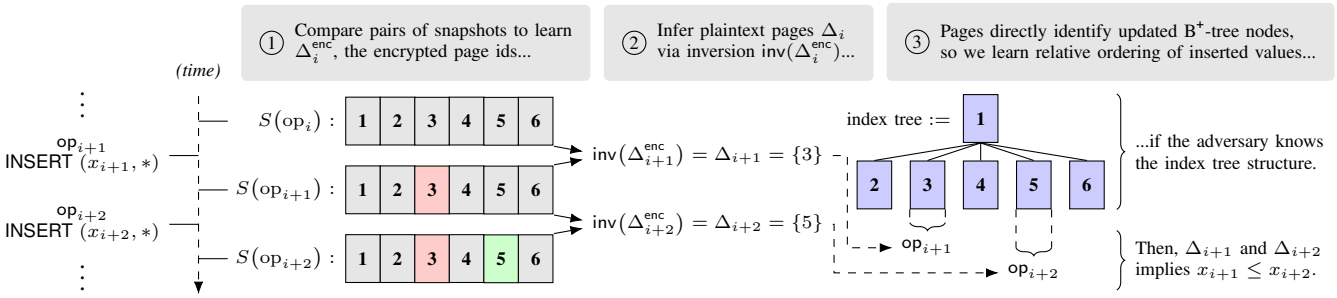


Figure 2: Example of what we can infer using the snapshots surrounding two operations, op_{i+1} and op_{i+2} (assuming the database page size $P = 4096$).

executing op_{i+1}. Since each B+-tree node corresponds to a page, the indices in Δ_{i+1} can be used to identify the nodes that were updated by op_{i+1}.

We recover this information through a mechanism we call *inversion* where we create an *inversion function* inv for each scheme such that $\text{inv}(\Delta_{i+1}^{\text{enc}}) = \Delta_{i+1}$. This comes from the observation that PLE modules structure files so they can access random pages in sublinear time and thus access and update pages deterministically. In most cases, determining inv is easy. SQLC and S3MC modify SQLite's B+-tree nodes so all cryptographic metadata is stored within existing database pages. Thus, every P-size page fits within $P/4096$ filesystem blocks. (This can be verified by direct inspection of the respective paging routines in [141], [159].) Since all valid database page sizes P are divisible by 4 KiB, SQLC and S3MC's inv function just needs to convert the 4 KiB block indices to P-size page indices).[3]

Proposition 4.1. *The* inv *function of SQLC and S3MC is*

$$\text{inv}_{\text{ESE}}(\Delta_*^{\text{enc}}) = \left\{ e \;\middle|\; b \in \Delta_*^{\text{enc}}; e = \left\lfloor \frac{b}{P/4096} \right\rfloor \right\}.$$

Inverting Gramine's Intel Protected File System (PFS) format is more complex [75]. At a high-level, the first 3 KiB of a PFS-encrypted file is stored in filesystem block 0; then, block 1 is a Merkle tree node, followed by 96 data blocks (each of which store 4 KiB of the original file). This pattern starts again with another Merkle tree node every 97 filesystem blocks. Despite this additional complexity, when the logical database page size P is larger than 4 KiB, we can still directly invert the encrypted deltas into plaintext deltas since the Merkle tree structure is deterministic.

Proposition 4.2. *The* inv *function of Gramine is*

$$\text{inv}_{\text{Gram}}(\Delta_*^{\text{enc}}) = \left\{ e \;\middle|\; b \in \Delta_*^{\text{enc}}; b > 0; q = \left\lfloor \frac{b-1}{M+1} \right\rfloor; \right.$$
$$r = b - q(M+1); r > 0;$$
$$\left. e = \left\lfloor \frac{(Mq + r - 1)}{P/4096} \right\rfloor \right\}$$

where $M = 96$ is the number of data blocks per Merkle tree node in Gramine and $P \in \{8192, 16384, 32768, 65536\}$ is the logical database page size of SQLite.

3. SQLite always flushes database pages in full P-sized increments (regardless of how much of the page was modified) so all 4 KiB blocks that span a given page are modified when the database page is modified.

Proposition 4.2 can be verified by direct inspection of [62]. There is one last edge case, however—when Gramine is used with a SQLite page size P of 4 KiB. Due to Gramine's 1 KiB offset at the start of the file, some ambiguities arise where there are two or more possible plaintext deltas that could have generated an encrypted delta. Fortunately, we can overcome this noise with a simple trick—we simply attempt to run the attack, and if certain structural properties are violated along the way, we can *backtrack* and try a different plaintext delta. We discuss backtracking as a broader attack technique in Section 5.2 and defer more detailed discussion of this specific case to Appendix B.

With the inv framework, all we need to consider is the behavior of page writes in the plaintext world. This makes things substantially easier to analyze and allows us to generalize our attack to multiple PLE modules. As such, in the rest of the attack, we only need to consider the behavior of page writes in the plaintext database file.

Exploiting plaintext deltas. The inversion function allows us to recover Δ_{i+1} and Δ_{i+2}, or the sets of indices of the plaintext logical database pages that were updated by op_{i+1} and op_{i+2}. By design, these pages map directly to nodes in the underlying B+-tree. We can use this information to infer something about the relationship between op_{i+1} and op_{i+2}.

For the purposes of this warmup, let us assume that the adversary knows how the plaintext pages map to nodes in the underlying index tree after operation op_i. (We eliminate this assumption in the description of the attack in Section 5.) As illustrated in Figure 2, the most significant interactions between the plaintext deltas and the known tree structure are in leaves of the tree. If the sets of leaves updated by op_{i+1} and op_{i+2} are disjoint, we learn the relative ordering of the values inserted by op_{i+1} and op_{i+2}. On the other hand, if the leaves updated by each operation overlap, we cannot definitively conclude the ordering of the values of op_{i+1} and op_{i+2}. Nevertheless, we still learn that the values of op_{i+1} and op_{i+2} are "close" to each other because they landed in the same leaf. Our attack performs approximate order reconstruction by repeatedly leveraging this behavior.

This exercise also reveals an observation that simplifies our attack—the inference does not require knowledge of the entire tree layout but rather just the layout of the *leaves* of the index tree. In the next section, we will develop techniques to recover the structure of these leaves.

(P1)	Nodes at depth d are updated before nodes at depth d' where $d > d'$.
(P2)	New nodes are appended to the file in the order they were created.
(P3)	The index tree is updated before the record tree is updated.
(P4)	In the no-deletion setting, nodes never switch between the index tree or the record tree after they are created.

Figure 3: Properties of SQLite B$^+$-trees, determined by direct inspection of the B$^+$-tree insertion algorithms.

5. The Attack: Learning the Leaves

Learning the structure of the index tree requires us to understand how page updates (as captured by plaintext deltas) can tell us about the position of nodes in the tree. In this section, we will show how we can reason about various properties of B$^+$-tree insertions to learn the ordering of the leaves of the SQLite index tree. Along the way, we will solve for one additional challenge introduced by SQLite through a process called *disambiguation*.

Notation. We use $\max_k(S)$ to denote the largest k elements of S. We denote the index tree after operation op_i as T_i^{idx} and the record tree after operation op_i as T_i^{rec}. We denote their roots respectively as $\mathsf{root}^{\mathsf{idx}}$ and $\mathsf{root}^{\mathsf{rec}}$. Then, given $T_i^* \in \{T_i^{\mathsf{idx}}, T_i^{\mathsf{rec}}\}$, we denote the set of nodes in T_i^* as $N(T_i^*)$, the set of leaf nodes in T_i^* as $\textsc{leaves}(T_i^*) = \{p \in N(T_i^*) \mid p \text{ has degree } 1\}$ and the set of non-leaf nodes in T_i^* as $\textsc{nonleaf}(T_i^*) = N(T_i^*) \setminus \textsc{leaves}(T_i^*)$. Finally, given a node $p \in N(T_i^*)$, we denote the parent of p as $\mathsf{parent}(p)$. We previously established the one-to-one mapping between B$^+$-tree nodes and the plaintext page numbers that identify them, so we abuse notation and do not distinguish between node identifiers (page numbers) as in $p \in \Delta_i$ and the nodes themselves as in $p \in N(T_i^*)$.

5.1. Disambiguation

Each SQLite insertion modifies both the index tree and the record tree. We only care about the index tree, but recall that the pages representing these two trees are interleaved in the same file. Aside from the initial state of the file, the logical order of the pages in the main file is not the same across different insertion transcripts. This requires us to *disambiguate* which tree each page belongs to. To do this, we leverage properties of SQLite's implementation of B$^+$-trees (Figure 3) and various algorithmic and graph-theoretic observations about them.[4] First, we capture the notion of a "new node" being added by an operation.

Definition 5.1. *The* new nodes introduced by op_i *are*

$$\mathsf{new}_i = \{p \in \Delta_i \mid p \notin \Delta_j \ \forall\ 0 \le j < i\}.$$

Then, p is a new node introduced by op_i *if $p \in \mathsf{new}_i$. (When op_i is clear from context, we simply say p is a* new node*.)*

4. Due to space restrictions, the proofs of most lemmas and theorems in this section are deferred to the full version of the paper.

Our idea is to iterate over the transcript of deltas in order and find operations that introduce a new node. Then, we identify which tree the new node belongs to. Given (P4), it suffices to identify the correct tree for each node as it is introduced to fully disambiguate the nodes across all deltas. There is a core property of the B$^+$-tree that allows us to learn each node's tree—whenever a new node is introduced by some operation, its parent node must be updated in the same operation since parents store pointers to their children.

Lemma 5.2 (Parental presence at inception)**.** *If p is a new node introduced by op_i, then $\mathsf{parent}(p) \in \Delta_i$.*

Lemma 5.2 is obvious in an algorithmic sense, but for an adversary who does not know the edges between the nodes ahead of time, the contrapositive of Lemma 5.2 leads to the following crucial observation. Let $T_i^* \in \{T_i^{\mathsf{idx}}, T_i^{\mathsf{rec}}\}$. Since $\textsc{nonleaf}(T_{i-1}^*)$ are the only possible parents of a new node, if there is *exactly one* T_{i-1}^* where $\Delta_i \cap \textsc{nonleaf}(T_{i-1}^*) \neq \varnothing$, then we can conclude that the new node belongs to T_i^*. However, there are cases where non-leaf nodes from *both* the index tree and the record tree appear in Δ_i. As such, we need additional tricks to determine which tree the new node(s) belong to. To start, we combine (P2) and (P3) to observe that, when a list of new nodes (sorted by page number) is added in a given operation, there is a way to partition the sorted list such that all the nodes on one side were added to the index tree and all of the nodes on the other side were added to the record tree.

Lemma 5.3 (Split point)**.** *Let $\mathsf{new}_i = \{p_1, \ldots, p_k\}$ be the set of new nodes introduced by op_i and let $p_1 < \cdots < p_k$. Let $\mathsf{new}_i^{\mathsf{rec}} = \mathsf{new}_i \cap N(T_i^{\mathsf{rec}})$ and suppose $\mathsf{new}_i^{\mathsf{rec}} \neq \varnothing$. Let $p_j = \min(\mathsf{new}_i^{\mathsf{rec}})$. Then, $p_1, \ldots, p_{j-1} \in N(T_i^{\mathsf{idx}})$ and $p_j, \ldots, p_k \in N(T_i^{\mathsf{rec}})$.*

Proof. It follows directly from (P2) and (P3) that there exists $1 \le j \le k+1$ such that $p_1, \ldots, p_{j-1} \in N(T_i^{\mathsf{idx}})$ and $p_j, \ldots, p_k \in N(T_i^{\mathsf{rec}})$. Then, suppose $\mathsf{new}_i^{\mathsf{rec}} = \mathsf{new}_i \cap N(T_i^{\mathsf{rec}})$ is not empty. This implies $j < k+1$, so $\min(\mathsf{new}_i^{\mathsf{rec}})$ exists. By definition, p_j is the smallest node in $\{p_j, \ldots, p_k\}$ so $p_j = \min(\mathsf{new}_i^{\mathsf{rec}})$. □

By identifying the split point for each set of new nodes, we can use Lemma 5.3 to correctly categorize the rest of the nodes. But Lemma 5.3 may seem circular at first—how can we know $\mathsf{new}_i^{\mathsf{rec}}$ if we do not know the split point p_j? To identify the *split point* for each new_i, we start with a simplifying assumption that the page size of the B$^+$-tree is set "reasonably" with respect to the size of the inserted key-value pairs—specifically, each node fits at least two key-value pairs. (This is a conservative assumption since two key-value pairs per node is very low.)

Assumption 5.4. *Let $T_i^* \in \{T_i^{\mathsf{idx}}, T_i^{\mathsf{rec}}\}$ and let c be the node capacity of T_i^*. Then, every key-value pair (k_i, x_i) stored in T_i^* has positive size $s_i < c/2$.*

Using Assumption 5.4, we can prove a technical lemma which states that when a set of new nodes are added to a

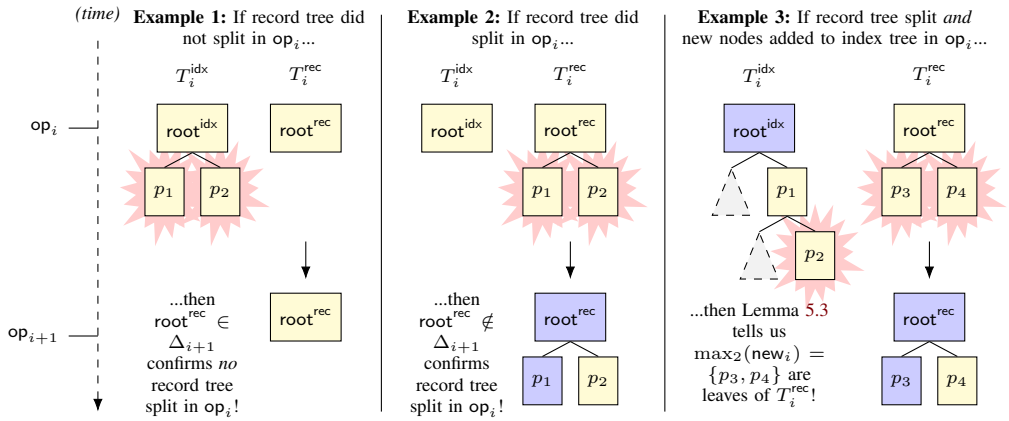


Figure 4: Examples of how to identify when initial record tree split occurs and the new leaves that were added to the record tree in that initial split. Nodes highlighted in yellow were updated by op_i and thus are in Δ_i. Red outlines denote that the node is new.

single tree in some op_i, each node is placed at distinct depth (except for operations which increase the tree's height).

Lemma 5.5 (Unique new node depths)**.** *Let* $T_i^* \in \{T_i^{\mathsf{idx}}, T_i^{\mathsf{rec}}\}$ *and let* c *be the node capacity of* T_i^**. Suppose a pair* (k', x') *with size* $s' < c/2$ *is inserted resulting in the updated tree* T_{i+1}^**. Then,*

1) *If the height of* T_{i+1}^* *is equal to the height of* T_i^**, at most one new node per depth was added to the tree.*
2) *If the height of* T_{i+1}^* *was increased by 1, at most one new node per depth was added to the tree except two nodes are added below the root.*

An important corollary emerges from a direct combination of Lemma 5.5 and the properties (P1) and (P2)—given a set of new nodes introduced by some op_i that all belong to the same tree, the node with the lowest page number is the only possible leaf if the tree's height is greater than 1.

Lemma 5.6 (Lowest new page is only candidate leaf)**.** *Let* new_i *be the set of new nodes introduced by* op_i*, let* $T_i^* \in \{T_i^{\mathsf{idx}}, T_i^{\mathsf{rec}}\}$*, and let* $\mathsf{new}_i^* = \mathsf{new}_i \cap N(T_i^*)$*. If the height of* T_{i-1}^* *is greater than 1, then at most one* $p_i^* \in \mathsf{new}_i^* \cap \textsc{leaves}(T_i^*)$*. If such a* p_i^* *exists, then* $p_i^* = \min(\mathsf{new}_i^*)$*.*

Applying Lemma 5.6 to the record tree's construction tells us that if we can identify the leaves of the record tree as they are introduced, we can then use Lemma 5.3 to disambiguate all of the other new nodes that were added in the same operation. But Lemma 5.6 has two caveats:

1) It requires that the tree's height is greater than 1 prior to the new node's addition to the tree. If the height of the tree is 1 (meaning there is only the root node), then more than one new leaf might have been added to the tree. This requires some special handling which we discuss in Section 5.1.1.
2) Lemma 5.6 leaves open the possibility that no new leaf was introduced. This would introduce a number of edge cases. Luckily, we show that a new record leaf is *always* introduced whenever the record tree gets new nodes (and how to identify such leaves) in Section 5.1.2.

5.1.1. Record tree's initial split. Figure 4 is a companion illustration for this section. To explain why Lemma 5.6 does not generalize to trees of height 1, consider a B$^+$-tree with only 1 node root^*. Inserted key-value pairs are placed into root^* as long as it has remaining capacity. When an insertion occurs that cannot be placed into root^* due to lack of space, a split occurs. As part of the split, root^* remains the tree's root and two new nodes are created as children of root^*. Then, the contents of root^* are distributed between its new children through a *rebalance* process. Exactly how this rebalance works is not important here—the important point is that the addition of these nodes at the initial split violates the "at most one" property of Lemma 5.6.

Because of this, we identify the record tree's initial split in a slightly different way. Suppose the record tree splits in op_j. (We stress that the adversary does not know j upfront.) Then, for all $i < j$, each op_i will insert its associated record into the root of the record tree $\mathsf{root}^{\mathsf{rec}}$ and so $\Delta_i \cap N(T_i^{\mathsf{rec}}) = \{\mathsf{root}^{\mathsf{rec}}\}$. When op_j is reached, two new nodes, p_1 and p_2 are added as children of $\mathsf{root}^{\mathsf{rec}}$ and so $\Delta_j \cap N(T_j^{\mathsf{rec}}) = \{\mathsf{root}^{\mathsf{rec}}, p_1, p_2\}$. However, just seeing two new nodes is not enough to conclude that the record tree had its initial split since it could have been new nodes added to the index tree. To do this, we can verify that the record tree actually did split in op_j by checking that $\mathsf{root}^{\mathsf{rec}}$ was not modified in the following operation op_{j+1}. Finally, even if new nodes were *also* added to the index tree in op_j, Lemma 5.3 tells us that the new nodes with the two greatest page numbers are the two nodes that were added to the record tree.

5.1.2. After the initial split. We now present our technique for disambiguating all new nodes that appear after the record tree's initial split. First, we explain why a new record leaf is *always* introduced whenever the record tree gets new nodes. Then, we show how to identify those leaves using a surprisingly simple trick in our core disambiguation result (Theorem 5.11). To start, we make a series of observations about the default behavior of insertions with respect to the record tree in SQLite. All rows within SQLite tables have

a 64-bit signed integer key, the *record ID*, that uniquely identifies the row within its table. (This is even the case when a custom primary key is assigned to the table.) The default behavior in SQLite is that record IDs are assigned automatically as an incrementing unique integer.

Assumption 5.7. *Record IDs are monotonically increasing over the transcript of insertions.*

Given the ubiquity of Assumption 5.7, SQLite's B$^+$-tree implementation has a special optimization. Since each new record ID is always greater than all previously existing IDs, new entries on the record tree are always at the extreme right end of the tree. Thus, instead of performing a normal rebalance on the record tree, SQLite appends a new node to the right-hand side of the page and adds the new entry to that page without performing a full rebalance. Subsequent insertions add records to that new node until it reaches capacity—at which point the same so-called "quick rebalance" routine [138] occurs again. To capture this special insertion behavior, we start with the following definition.

Definition 5.8. *The* active record node *for* op_i *is the* $p \in \mathsf{leaves}(T_i^{\mathsf{rec}})$ *in which* $r_i = (x_i, *)$ *is inserted.*

Such a node exists for all op_i since all data in a B$^+$-tree is stored in the leaves and, since data is never split up between two nodes, only one such node exists for a given record r_i. We observe an interesting behavior about the active record node: it is always at the right side of the tree, and it remains the active record node for a range of operations until the active record node fills up and requires a new node to be added to the tree. We formalize this with the following.

Lemma 5.9 (Active record node chaining)**.** *Let* p *be the active record node for* op_i*. Given* op_{i+1}*, the subsequent insertion operation, and* new_{i+1}*, the new nodes introduced by* op_{i+1}*, exactly one of the following is true:*

1) $N(T_{i+1}^{\mathsf{rec}}) \cap \mathsf{new}_{i+1} = \varnothing$ *and* p *is the active record node for* op_{i+1}*.*
2) *There exists* $q \in N(T_{i+1}^{\mathsf{rec}}) \cap \mathsf{new}_{i+1}$ *where* q *is the active record node for* op_{i+1}*.*

A key corollary emerges from the contrapositive of Lemma 5.2 and Lemma 5.9—a non-leaf node of the record tree is only updated exactly when the active record node changes from the previous operation.

Lemma 5.10 (Record non-leaves only updated on new leaf)**.** *Let* new_i *be the new nodes introduced by* op_i*.* $\Delta_i \cap \textsc{nonleaf}(T_i^{\mathsf{rec}}) \neq \varnothing$ *if and only if there exists* $q \in \mathsf{new}_i \cap \textsc{leaves}(T_i^{\mathsf{rec}})$*.*

Lemma 5.10 tells us that operations in which a non-leaf node of the record tree appears are exactly the operations that add a new record tree leaf. If we can identify which of the new nodes is the new leaf, then Lemma 5.3 and Lemma 5.6 tell us that the new leaf "splits" the ordered set of new nodes such that we can identify which nodes belong to the index tree and which nodes belong to the record tree. We identify new record tree leaves with the following result.

```
1: leaves := ∅                                          ▷ Leaf nodes of T^rec
2: nonLeaf := {root^rec}                                ▷ Non-leaf nodes of T^rec
3: maxNode := max({root^idx, root^rec})                 ▷ Max node seen so far
4: for i ∈ [0, n) do
5:     new_i := {p ∈ Δ_i | p > maxNode}
6:     maxNode = max(new_i ∪ {maxNode})
7:     if leaves = ∅ then                               ▷ Special-case: Section 5.1.1
8:         if root^rec ∉ Δ_{i+1} then
9:             leaves = max_2(new_i)                    ▷ By Lemma 5.3
10:    else if Δ_i ∩ nonLeaf ≠ ∅ then                   ▷ After initial split: Section 5.1.2
11:        leaf := max (new_i ∩ Δ_{i+1})                ▷ By Theorem 5.11
12:        leaves = leaves ∪ {leaf}
13:        nonLeaf = nonLeaf ∪ [leaf + 1, maxNode]
14: return recordNodes := leaves ∪ nonLeaf
```

Algorithm 1: LEAFBLOWER disambiguation.

Theorem 5.11 (Leaf disambiguation)**.** *Let* new_i *be the new nodes introduced by* op_i *denoted such that* $p_1 < \cdots < p_k$ *and* $p^* = \max(\mathsf{new}_i \cap \Delta_{i+1})$*. If the height of* T_{i-1}^{rec} *is greater than 1 and* $\Delta_i \cap \mathsf{nonleaf}(T_{i-1}^{\mathsf{rec}}) \neq \varnothing$*, then*

(a) $p_L \in N(T_i^{\mathsf{idx}})$ *for all* $p_L \in \mathsf{new}_i$ *where* $p_L < p^*$*,*
(b) $p^* \in \mathsf{leaves}(T_i^{\mathsf{rec}})$*, and*
(c) $p_R \in \mathsf{nonleaf}(T_i^{\mathsf{rec}})$ *for all* $p_R \in \mathsf{new}_i$ *where* $p_R > p^*$*.*

Using Theorem 5.11, we derive Algorithm 1, the first step in the LEAFBLOWER attack. Algorithm 1 first handles the initial split edge case by Lemma 5.3. After that, it identifies the nodes that belong to the record tree by repeated application of Theorem 5.11. This set of nodes will be used in the second part of the attack to isolate the nodes that belong to the index tree.

Theorem 5.12. *Algorithm 1 returns* $N(T_n^{\mathsf{rec}})$*.*

5.2. Tracking the Leaves via Rebalances

At this point, we have identified the set of pages that correspond to the index tree. We now need to do a few things: (1) identify the leaves of the index tree, (2) determine the operations that are contained in each leaf, and (3) track the ordering of the leaves throughout the construction of the tree. By doing this, we will end up with an ordered list of the leaves and the operations that are contained within them.

Leaf identification works using the same techniques from Section 5.1. Starting from the knowledge that Page 1 is the $\mathsf{root}^{\mathsf{idx}}$, we first use the observation from Section 5.1.1 to identify the initial split of the index tree—namely, if the initial split happens in op_j, then $\mathsf{root}^{\mathsf{idx}} \in \Delta_i$ for all $i \leq j$ and $\mathsf{root}^{\mathsf{idx}} \notin \Delta_{j+1}$. This also means that two new leaves are introduced by op_j as children of $\mathsf{root}^{\mathsf{idx}}$. After handling the initial split, we can identify the leaves by directly applying Lemma 5.6—when new nodes new_i are introduced by an operation op_i, the smallest new node $p = \min(\mathsf{new}_i)$ is the only node that could be a leaf. By default, we simply assume that p is a leaf and classify it as such. (We discuss later how to handle rare cases where it turns out p is not a leaf.)

To track the ordering of the leaves throughout the transcript, we leverage the behavior of SQLite's *rebalance* subroutine [135]. When SQLite rebalances a set of nodes p_a, p_b, p_c, it reassigns their page numbers so that the logical

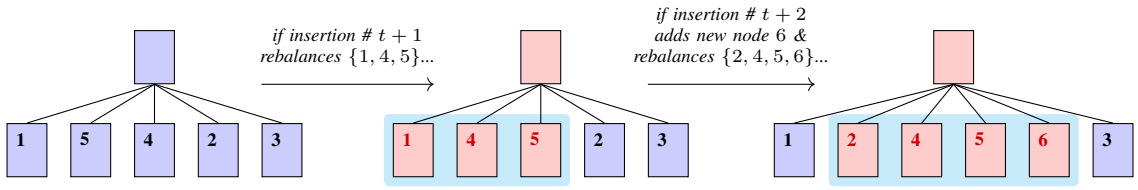


Figure 5: Rebalances renumber the filesystem pages of nodes so that the filesystem page numbers correspond to the logical order of the rebalanced pages.

```
1: exclusions := ∅                                  ▷ Used for backtracking "imposter" leaves
2: leaves := ∅                                      ▷ Leaf nodes of T^idx
3: maxPage := max({root^idx, root^rec})             ▷ Max node seen so far
4: Initialize LO as LeafOrganizer                   ▷ See Section C
5: for i ∈ [0, n) do
6:     Δ_i^idx := Δ_i \ recordNodes                 ▷ recordNodes from Algorithm 1
7:     new_i^idx := {p ∈ Δ_i^idx | p > maxPage}
8:     if leaves = ∅ then
9:         if |new_i^idx| = 2 then                  ▷ Special case: initial split
10:            Add operation ids [0, i] to page min(new_i^idx) in LO
11:            Reorder and redistribute sort(new_i^idx) in LO
12:    else
13:        balanced = sort(Δ_i^idx ∩ leaves)
14:        if not LO.IsConsistent(balanced) then
15:            Add inconsistent node to exclusions and backtrack
16:        if new_i^idx ≠ ∅ then
17:            probableNewLeaf := min(new_i^idx)    ▷ By Lemma 5.6
18:            if probableNewLeaf ∉ backtrackExclusions then
19:                leaves = leaves ∪ {probableNewLeaf}
20:                balanced = balanced || ⟨probableNewLeaf⟩
21:        p = balanced[⌈|balanced|/2⌉]             ▷ Pick the middle leaf
22:        Add operation identifier i to page p in LO
23:        Reorder and redistribute balanced in LO
24: Iterate over pages in LO in order and concatenate their operation
    identifiers into a single ordered sequence orderedOps
25: return orderedOps
```

Algorithm 2: LEAFBLOWER index leaf and insertion ordering.

ordering of p_a, p_b, p_c is the same as the ordering of their page numbers on disk. Then, it redistributes the keys stored in p_a, p_b, and p_c so that each node contains roughly the same amount of keys. Figure 5 illustrates this behavior. Because new nodes in the index tree are only created as a result of rebalances that overflowed, we can always learn the position of new nodes as they appear using this technique.

Identifying the leaves that each operation landed in is simple—we just look at the leaves that each operation modified. If only one leaf is modified, then we know exactly which leaf the operation belongs to. We do not know its ordering relative to other operations in the same leaf—but we find that placing the operation in the middle of the leaf is enough to achieve good reconstruction results in our evaluation. If multiple leaves are modified (because a rebalance occurred), we do not know definitively in which leaf it landed. However, similar to the single leaf case, we find that placing it in the middle of all operations that were involved in the rebalance achieves good reconstruction.

This observation leads us to our reconstruction attack: we maintain an ordered list of leaves, each of which contains its own ordered list of operations (representing the operations whose values are contained inside the leaf). Then, we iterate through the plaintext deltas again, and only consider those edits that correspond to leaves of the index tree. If a single leaf was edited, then we add that operation to that leaf in the structure in the middle of all previous operations that were in the leaf. If multiple leaves were edited, we perform a "rebalance" over the operations across all of the edited leaves, reorder the leaves themselves so that the rebalanced leaves' page numbers are in ascending order. Then, we update the list of operations stored in each leaf according to the result of the rebalance. To handle reordering efficiently, we instantiate the ordered list of leaves as an addressable doubly-linked list with an additional rebalance operation which approximates SQLite's rebalance procedure. We refer to this structure as the LeafOrganizer in Algorithm 2. More detail on this data structure is in Appendix C.

Backtracking due to "imposter" leaves. While in most cases, the candidate leaf p^* given by Lemma 5.6 is indeed a leaf, Lemma 5.6 unfortunately leaves open the possibility that p^* is actually an interior node of the tree. Because the B$^+$-tree supports variable-length keys, it is possible that a rebalance at the lowest level did not require a new leaf node but pushed new divider keys into the rebalanced nodes' parent that caused a rebalance at a higher level. This, in turn, could have created a new node at a level higher than the leaf level. Treating the interior node as a leaf can mess up the actual ordering of leaves in our bookkeeping structure and result in potentially incorrect results.

To handle this, we observe that rebalances always occur on sets of leaves whose logical ordering is contiguous. In the case where the p^* is not actually a leaf, a leaf rebalance operation later in the transcript that initially seems to include p^* may reveal that including p^* in the rebalance would result in a violation of the contiguity property. If so, we then know that p^* could not have been a leaf. With additional bookkeeping, we can *backtrack* to the place where we inserted p^* as a leaf and instead reclassify it as an interior node. It is possible that we never see such a contiguity violation, but we find that this does not affect the performance of our attack. (In practice, backtracking is rare—our experiment showed an average of two backtracks across workloads.)

The attack. Given the plaintext deltas $(\Delta_i)_{i\in[n]}$ and recordNodes from Algorithm 1, Algorithm 2 reconstructs the ordering of the leaves of the index tree after each operation op_i and traces a possible ordering of the inserted indexed values x_i throughout the transcript. At the end, we iterate over the sequence of leaves in-order and concatenate the sequence of record identifiers held by each leaf. This gives us the final approximate ordering of the insertions.

5.3. Mapping Insertions to Values

As the last step, we show how to map the approximate insertion order that was recovered via Algorithm 2 to plaintext inserted values if the adversary has auxiliary information about the insertions—specifically, information about the distribution of inserted values.

Notation. Given a distribution $\mathbf{A}$, we denote the cumulative distribution function of $\mathbf{A}$ as $\mathrm{cdf}_{\mathbf{A}}$ and its inverse as $Q(x) := \mathrm{cdf}_{\mathbf{A}}^{-1}(x)$ (also known as $\mathbf{A}$'s *quantile function*).

The attack. Given the approximate insertion order reconstruction from Algorithm 2 and an auxiliary distribution $\mathbf{A}$, the adversary computes the quantile function of $\mathbf{A}$ to retrieve the estimated value $\widetilde{x}_i$ for each $i \in [n]$ and $n \in \mathbb{N}$:[5]

$$\widetilde{x}_i = \mathrm{cdf}_{\mathbf{A}}^{-1}\left(\frac{i}{n}\right).$$

The intuition behind our value reconstruction is straightforward: we observe that ordered elements can be *placed* to conform within the surface defined by a discrete probability mass function characterized by $\mathbf{A}$. This placement not only preserves the order of elements but also aligns proportionally with each probability in the support of $\mathbf{A}$.

Remark. As mentioned in Section 2, our approach shares similarities with inference attacks targeting order-preserving (OPE) and order-revealing encrypted (ORE) columns; in particular, it is similar to the dense case of the SORTING-ATK by Naveed, Kamara, and Wright [114]. However, the key distinction lies in the type of leakage: while OPE- and ORE-encrypted columns reveal both order *and* frequency when two values are identical, our attack only recovers the order. This difference prevents us from applying techniques from improvements to the SORTING-ATK (e.g. [19], [66], [92]) which require either frequency leakage (which is not recovered by our attack[6]) or leakage from multiple columns.

5. For simplicity, we assume that the cumulative distribution function is continuous and strictly increasing, as was the case for all our auxiliaries in the evaluation section. As a technical remark, if this condition is not met, one could instead utilize the generalized inverse distribution function [49].

6. There are several barriers to recovering exact or even approximate frequency information. As described in Section 3, B$^+$-tree insertions treat every inserted key as unique, so the insertion algorithm's behavior is identical whether or not the key has been inserted before. Even if the same leaf node is updated by multiple operations, it only tells us that the values on that page are close together in *rank*. As such, under the current adversarial model, it is difficult to conclude anything about the frequency of inserted values. Nevertheless, we conjecture that slightly strengthening the adversary model may allow frequency reconstruction. To consider a strong case, the persistent adversaries used in prior TEE attacks (discussed in Section 1) can see the number of times B$^+$-tree pages are updated or read in a given operation (whereas our model cannot see how many times each page was accessed during the course of a single insertion). Frequency information may be derivable in this scenario. Even "weaker" models like injection-based adversaries (e.g., [24], [58], [70]) may also be able to use injections to learn more fine-grained information about the ordering and frequency of values within each page. In all these scenarios, improvements to the SORTING-ATK (e.g. [19], [66], [92]) may be applicable.

6. Evaluation

We implemented and evaluated LEAFBLOWER against SQLite 3.49.1 [136], SQLCipher 4.7.0 [159], SQLite3 Multiple Ciphers 2.1.0 [141], and Gramine 1.8 [61]. The performance of LEAFBLOWER is nearly equivalent regardless of the underlying PLE module, so we present a single set of results that show the worst-case performance across all PLE modules. Our artifact containing our implementation and reproducibility instructions is available at [51].

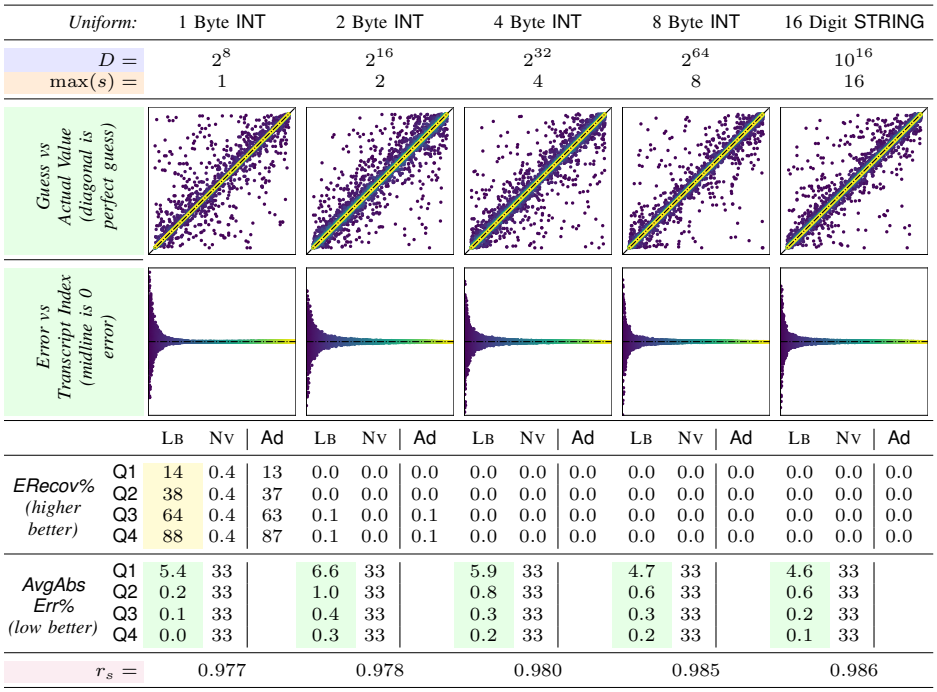
Methodology. Given a dataset of n records, we create a SQLite database with page size $P = 4096$[7] under each PLE module with one table and one index. Then, we insert records from the dataset into the table. After each insertion op_i, we take a snapshot $S(\mathsf{op}_i)$ of the index file and record the encrypted delta Δ_i^{enc} between it and the previous snapshot. We then pass the list of encrypted deltas $\left(\Delta_i^{\mathsf{enc}}\right)_{i\in[n]}$ to the LEAFBLOWER algorithm. LEAFBLOWER computes the inversion function for the given PLE module to recover the plaintext deltas $\left(\Delta_i\right)_{i\in[n]}$, performs approximate insert order reconstruction, and performs approximate value reconstruction using an auxiliary dataset. We then compare the reconstructed values to the actual value of each insertion.

Synthetic data. In Figure 6, we evaluate the accuracy of the approximate insert order reconstruction on synthetic data. Each dataset contains $n = 1000000$ insertions sampled uniformly at random from a given domain using an equivalent uniform distribution as the auxiliary information. Each experiment uses numerical values from different domain sizes D and with different *maximum encoding size* $\max(s)$ to investigate how the accuracy of the attack is affected by the size of the domain as well as the number of values that could fit into each B$^+$-tree page.

We report *exact recovery %* (ERecov%, the percent of insertions with correctly guessed values) and *average absolute error as a % of the domain size* (AvgAbsErr%, the absolute difference between the guessed and actual value, divided by the domain size). For ERecov%, we additionally include an *advantage* (Ad) comparison recently introduced by Espiritu, Kamara, and Moataz [50] as a method for evaluating attacks that use auxiliary information.[8] The Ad for ERecov% is the absolute difference between LEAFBLOWER's ERecov% to the ERecov% of a "naive" adversary who guesses based on the same auxiliary information $\mathbf{A}$. This verifies that the attack learns more information than what $\mathbf{A}$ tells us. We define the naive adversary as the adversary which guesses the value of each operation by sampling from $\mathbf{A}$ and report the average ERecov% from running the naive adversary 10,000 times. Additionally, we report Spearman's rank correlation coefficient ($-1 \leq r_s \leq 1$) [132] between the guessed and actual values for each operation. r_s

7. In Figure 9 in the Appendix, we discuss further results on synthetic data where we increase the page size P as a potential attack mitigation.

8. [50] does not define an equivalent notion of Ad for AvgAbsErr%, so the Ad column is blank in AvgAbsErr% rows. Future work may come up with other advantage-like notions that can capture these types of metrics.



① As the domain size D increases, ERecov% decreases, but AvgAbsError% remains small as long as our auxiliary information is accurate.

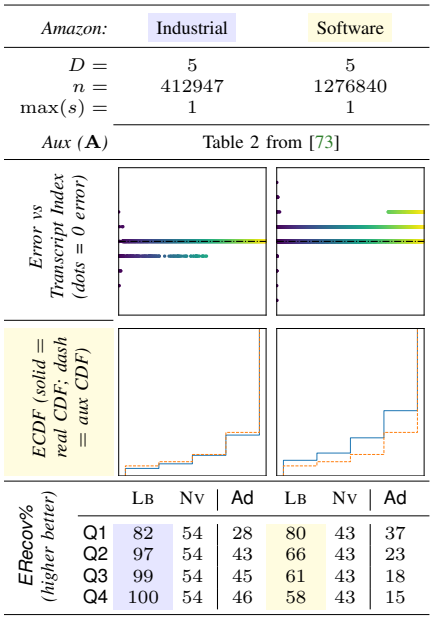
② AvgAbsError% decreases as the byte size s increases, as larger s means less keys are stored in each node (which reduces noise in order recov.).

③ Our attack has best ERecov% on small D—we achieve positive Ad over the NV baseline (which achieves 0% ERecov% in most cases).

④ Even when D is large, AvgAbsErr% is lower on later inserts (e.g. Q4) compared to earlier (e.g. Q1). LB*'s reconstruction gets better later in the transcript* since LB recovers better ordering information when the # of leaves in the B$^+$-tree is larger (compared to early insertions which get inserted into the same few nodes). The graphs also indicate later inserts (yellow points) are more accurately ordered than early inserts (purple). In comparison, NV's AvgAbsErr% remains constant at 33%.

⑤ The Spearman correlation r_s (see Section 6) is always close to 1. This indicates accurate order reconstruction (independent of the value recovery).

Figure 6: Evaluation and commentary on synthetic datasets (where $n = 1000000$). LB is LEAFBLOWER; NV is the naive baseline. (For visual clarity, the scatter plots are downsampled to every 100th point.)



① When $\text{cdf}_{\mathbf{A}}$ is close to the CDF of the real data, ERecov% Adv *increases* from Q1 to Q4. This follows our synthetic results from Figure 6, where we had accurate $\mathbf{A}$ by design...

② ...but when $\mathbf{A}$ is off (shown in the ECDF graph), LB has *decreasing* ERecov% from Q1 to Q4 since as the order recovery gets better, the value recovery shifts more away from the real CDF. This stresses that LB can decline if "inaccurate" $\mathbf{A}$ is chosen. Even so, LB's ERecov% Adv is good as long as the shape of $\mathbf{A}$ is close.

Figure 7: Evaluation on 2023 Amazon review data [71], [72].

measures the monotonicity between pairs of two variables. $r_s = 0$ indicates there is no correlation while $r_s = 1$ or $r_s = -1$ means the variables are perfectly monotone. We use this to independently evaluate the correctness of the *order reconstruction* portion of the attack. Finally, we report the above metrics independently on each *quarter* of the transcript index (e.g. Q1 corresponds to the metric calculated on the first 25% of the insertions; Q2 corresponds to the second 25% of the insertions, etc.). We use this to highlight an interesting behavior of our attack—the order reconstruction (and thus, the value reconstruction when $\mathbf{A}$ is accurate) improves for later operations in the transcript.

Interpretation of the synthetic results is in Figure 6. *Our accuracy improves for later operations in the transcript compared to earlier operations*. Also, when the auxiliary $\mathbf{A}$ is very close to the real CDF and when the domain size D is small, we can achieve a good ERecov% rate. Even when D is large, our AvgAbsErr% is consistent.

Real datasets. We now evaluate LEAFBLOWER against indexed attributes from two real-world datasets.[9] Based on our findings from Figure 6, we primarily focus on attributes with small domain size to highlight cases where the exact recovery of the attack performs well. First, in Figure 7, we use star ratings from two different categories of Amazon products from a 2023 dataset [71], [72]. As auxiliary information, we directly use a histogram from a 2009 article by Hu, Zhang, and Pavlou on the distributions of online product reviews (Table 2 in [73]). While unconventional, this experiment is useful for two reasons. First, it highlights that the LEAFBLOWER attack only requires a CDF and not concrete records. Second, while the accuracy improves in the *Industrial* dataset for later insertions (as in the

9. Our experiments do not attempt to reconstruct information that was not already in publicly available datasets. We intentionally chose datasets that did not require any payment, required at most an online request form to access, and whose download and parsing could be mostly automated.

Texas PUDF:		Ethnicity			Sex			Race			Length of Stay		
$D =$		3			3			5			365		
Aux ($\mathbf{A}$)		Texas Census QuickFacts [144]									NY SPARCS [116]		
ERecov% (higher better)		LB	NV	AD	LB	NV	AD	LB	NV	AD	LB	NV	AD
	Q1	84	53	31	99	50	49	45	54	−9	70	13	57
	Q2	100	53	47	100	50	50	73	54	19	83	13	70
	Q3	100	53	47	100	50	50	85	54	31	74	13	61
	Q4	100	53	47	66	50	16	93	54	39	51	13	38

① As in Figure 7, when the auxiliary $\mathbf{A}$ is close to the real CDF, we achieve increasingly higher ERecov% later in the transcript.

② As in Figure 7, when $\mathbf{A}$ is off, our guesses are shifted away from the real CDF causing ERecov% to decrease. This emphasizes the importance of choosing "good" $\mathbf{A}$—otherwise, the performance of the attack degrades.

Figure 8: Evaluation on attributes from the 2018 Texas PUDF [142].

experiments in Figure 6), the *Software* dataset's accuracy actually decreases for later insertions. This is because the choice of auxiliary is not quite correct—this can be seen by comparing the ECDF of the real dataset to that of the auxiliary. This serves to emphasize the importance of *accurate* auxiliary information for the value reconstruction portion of the attack (as commonly stressed in prior work [19], [50], [52], [114]). Nevertheless, even when the auxiliary is off, LEAFBLOWER's advantage is better than that of the naive baseline for the reported metrics.

To demonstrate that our attack works on sensitive and categorical data, in Figure 8, we use records from the 2018 Texas Hospital Inpatient Discharges dataset [142]. This dataset contains approximately 700k records about hospital discharges from 2018. As auxiliary for some columns, we use histogram information from the United States Census Bureau's Texas Census QuickFacts publication from April 2025 [144]. In one case, where information is not available in QuickFacts, we use records from the New York SPARCS 2022 dataset [116] of hospital discharges in New York. We omit error and ECDF graphs for space, but the shape of the graphs follows patterns similar to those in Figure 7.

7. Possible Mitigations

Recommendations for ESE. For ESEs like SQLC and S3MC, our recommendation is to clarify in the adversarial threat model that such mechanisms are meant for adversaries that can only observe some fixed number of snapshots (smaller than the number of insertions) and do not provide strong confidentiality guarantees against multi-snapshot adversaries. A more long-term mitigation would be to explore *write-only oblivious RAM* techniques (e.g. [13], [21], [31], [34]) which are more efficient than general *oblivious RAM* (ORAM) [60] as they only hide writes. Another approach would be to use *breach-resistant STE* schemes (e.g. [7], [81], [120]) that are resilient to multi-snapshot adversaries. One potential roadblock with using state-of-the-art breach-resistant STE schemes (and most STE schemes in general) is that they do not directly handle concurrent write operations which is an explicit requirement for many practical database storage engines—combining breach-resistance with techniques from an emerging line of work on concurrency in STE might close this practical gap [2], [3], [4], [27], [81].

Recommendations for TEEs. Mitigations for TEEs are tricker since the usual adversarial models are even more powerful than the multi-snapshot setting. ORAM [60] has already been used in TEEs to hide access patterns to memory pages [1], [36], [45] and within filesystem syscalls [6], [41]. Our work provides more evidence supporting the use of such techniques. However, the $\Omega(\log n)$ bandwidth lower-bound of ORAM [94] introduces additional overhead which may not be appropriate for the setting of online transactional databases. Snapshot-secure ORAMs [46] are also appropriate for this setting, but recent lower-bounds by Persiano and Yeo suggest this may be nearly as hard as in the persistent setting for adversaries that receive multiple snapshots [120].

Remark. Our mitigation recommendations introduce additional cryptographic primitives, but one could attempt to mitigate the attack by breaking the some of the structural properties we exploit. For example, Lemma 5.3 and Lemma 5.6 could be broken by randomizing the order of nodes when multiple nodes are added in a single operation. It is not clear if such an approach is robust enough to prevent leaf order recovery—early in the development of this work, we found that we could also use PQ-trees [23] (which were previously used to attack encrypted range search ESAs in the persistent model in Grubbs, Lacharité, Minaud, and Paterson [64] and Markatou and Tamassia [106]) to recover the order of the B$^+$-tree leaf nodes (even though this ended up being more complicated than needed in our final attack). We stress that all the mitigations mentioned above introduce additional overhead and assessing their impact on existing workloads would be a valuable area of further research.

8. Conclusions and Future Work

In this paper, we show a concrete attack against TEE-based EDBs (and more broadly, PLE modules) that leverages a multi-snapshot external memory adversary model that has not been explored in depth in prior work. Our work makes the following points: (1) since persistent databases must eventually write data to disk, the filesystem is a vector for side-channel attacks against TEE-based EDBs; and (2) while it may be unsurprising that filesystem writes leak *something*, it is surprising (and novel) how *much* you can recover from just them. This work is intended to be a first step to motivate further study of multi-snapshot adversaries (and other weaker adversaries) when designing TEE-based EDBs and ESEs, and, more broadly, spur investigation of PLE modules with stronger multi-snapshot guarantees. We discuss additional directions for future work below.

Less snapshots and deletions. The main limitation of our attack is that it currently only works in the n-snapshot setting where the adversary receives a snapshot after each of the n insertions. Another open question is what knowledge can be extracted in the s-snapshot setting, where $s < n$, as an adversary obtains s snapshots throughout the n insertions. Another limitation of our work is the assumption that the workload consists solely of insertions, without accounting for deletions. We leave this for future research.

Multi-index SQLite tables. We focused on the single-index scenario to provide a baseline for what is possible for to reconstruct under "simple" conditions (and, as discussed in Appendix A, can already be used to attack non-SQLite databases with multiple indexes). A natural follow-up is to consider SQLite tables with multiple indexes. This substantially increases the complexity of the disambiguation problem from Section 5.1. Even so, we conjecture one can use rebalances to constrain which nodes can appear in the same tree, albeit with less guarantees about correctness.

Other index structures. SQLite provides interfaces for more expressive persistent index structures such as *R^*-trees* [17] for multi-dimensional search [139]. Such indexes are also supported by third-party database plugins like SpatiaLite [56] as well as in other databases like PostGIS [121]. Similar attacks in the multi-snapshot model may be possible not just on these indexes but also other key-value storage engine structures such as *log-structured merge* (LSM) trees [118] (as used in, e.g. BigTable [32], Cassandra [93], DynamoDB [48], and RocksDB [44]).

Acknowledgments

We thank Shweta Shinde for helpful conversations about TEEs and Marilyn George and Casey Nelson for feedback on early drafts. We also thank the developers of all software involved in the disclosure process for their feedback. In particular, the developers of SQLCipher and SQLite3 Multiple Ciphers gave very helpful and detailed suggestions that helped us improve the presentation of this work. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] S. Aga and S. Narayanasamy, "InvisiPage: oblivious demand paging for secure enclaves," in *Proc. 46th ISCA*, 2019.

[2] A. Agarwal and Z. Espiritu, "Sequentially Consistent Concurrent Encrypted Multimaps," in *Proc. 10th IEEE Euro. S&P*, 2025.

[3] A. Agarwal and S. Kamara, "Encrypted Key-Value Stores," in *INDOCRYPT 2020*, 2020.

[4] A. Agarwal, S. Kamara, and T. Moataz, "Concurrent Encrypted Multimaps," in *ASIACRYPT 2024*, 2024.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proc. 2004 ACM SIGMOD*, 2004.

[6] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX," in *Proc. 2018 NDSS*, 2018.

[7] G. Amjad, S. Kamara, and T. Moataz, "Breach-Resistant Structured Encryption," *Proc. PETS*, vol. 2019, no. 1, 2019.

[8] Anjuna Security. (2025). [Online]. Available: https://www.anjuna.io/

[9] P. Antonopoulos, A. Arasu, K. D. Singh, K. Eguro, N. Gupta, R. Jain, R. Kaushik, H. Kodavalla, D. Kossmann, N. Ogg, R. Ramamurthy, J. Szymaszek, J. Trimmer, K. Vaswani, R. Venkatesan, and M. Zwilling, "Azure SQL Database Always Encrypted," in *Proc. 2020 ACM SIGMOD*, 2020.

[10] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade, "Socrates: The New SQL Server in the Cloud," in *Proc. 2019 ACM SIGMOD*, 2019.

[11] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. 12th USENIX OSDI*, 2016.

[12] Asylo. (2018). [Online]. Available: https://asylo.dev/

[13] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "ObliviSync: Practical Oblivious File Backup and Synchronization," in *Proc. 2017 NDSS*, 2017.

[14] S. Bajaj and R. Sion, "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality," *IEEE Trans. Knowl. and Data Eng.*, vol. 26, no. 3, 2014.

[15] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. Comp. Syst.*, vol. 33, no. 3, 2015.

[16] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indices," in *Proc. 1970 ACM SIGFIDET Workshop Data Desc. Acc. and Cont.*, 1970.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *ACM SIGMOD Record*, vol. 19, no. 2, 1990.

[18] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and Efficiently Searchable Encryption," in *CRYPTO 2007*, 2007.

[19] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, "The tao of inference in privacy-protected databases," *Proc. VLDB Endow.*, vol. 11, no. 11, 2018.

[20] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting Leakage Abuse Attacks," in *Proc. 2020 NDSS*, 2020.

[21] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward Robust Hidden Volumes Using Write-Only Oblivious RAM," in *Proc. 2014 ACM CCS*, 2014.

[22] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *CRYPTO 2011*, 2011.

[23] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comp. and Sys. Sci.*, vol. 13, no. 3, 1976.

[24] B. Bourassa, Y. Michalevsky, and S. Eskandarian, "G-DBREACH Attacks: Algorithmic Techniques for Faster and Stronger Compression Side Channels," in *Proc. ACNS*, 2025.

[25] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on S3," in *Proc. 2008 ACM SIGMOD*, 2008.

[26] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proc. 11th USENIX WOOT*, 2017.

[27] T. Brézot and C. Hébant, "Findex: A Concurrent and Database-Independent Searchable Encryption Scheme," Cryptology ePrint Archive, Paper 2024/1541, 2024.

[28] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," in *Proc. 2021 ACM CCS*, 2021.

[29] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proc. 27th USENIX Sec.*, 2018.

[30] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution," in *Proc. 26th USENIX Sec.*, 2017.

[31] A. Chakraborti, C. Chen, and R. Sion, "DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries," *Proc. PETS*, vol. 2017, no. 3, 2017.

[32] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comp. Syst.*, vol. 26, no. 2, 2008.

[33] M. Chase and S. Kamara, "Structured Encryption and Controlled Disclosure," in *ASIACRYPT 2010*, 2010.

[34] C. Chen, A. Chakraborti, and R. Sion, "PD-DM: An efficient locality-preserving block device mapper with plausible deniability," *Proc. PETS*, vol. 2019, no. 1, 2019.

[35] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX Demystified: A Top-Down Approach," *ACM Comp. Surv.*, vol. 56, no. 9, 2024.

[36] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *Proc. 25th USENIX Sec.*, 2016.

[37] ——, "Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture," *Found. Trends Elec. Des. Autom.*, vol. 11, no. 3, 2017.

[38] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," in *Proc. 13th ACM CCS*, 2006.

[39] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks," *IACR Trans. CHES*, vol. 2018, 2018.

[40] C. Dar, M. Hershcovitch, and A. Morrison, "RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time," *Proc. 2023 ACM SIGMOD*, 2023.

[41] E. Dauterman, V. Fang, I. Demertzis, N. Crooks, and R. A. Popa, "Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage," in *Proc. ACM SIGOPS 28th SOSP*, 2021.

[42] Decentriq. (2025). [Online]. Available: https://www.decentriq.com/

[43] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, and Y. He, "Taurus Database: How to be Fast, Available, and Frugal in the Cloud," in *Proc. 2020 ACM SIGMOD*, 2020.

[44] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications," *ACM Trans. Storage*, vol. 17, no. 4, 2021.

[45] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged Side-Channel attacks," in *Proc. 27th USENIX Sec.*, 2018.

[46] Y. Du, D. Genkin, and P. Grubbs, "Snapshot-Oblivious RAMs: Sublogarithmic Efficiency for Short Transcripts," in *CRYPTO 2022*, 2022.

[47] F. B. Durak, T. M. DuBuisson, and D. Cash, "What else is revealed by order-revealing encryption?" in *Proc. 2016 ACM CCS*, 2016.

[48] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, S. Sivasubramanian, J. C. Sorenson III, S. Sosothikul, D. Terry, and A. Vig, "Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service," in *Proc. 2022 USENIX ATC*, 2022.

[49] P. Embrechts and M. Hofert, "A note on generalized inverses," *Math. Meth. Ops. Research*, vol. 77, no. 3, 2013.

[50] Z. Espiritu, S. Kamara, and T. Moataz, "Bayesian Leakage Analysis," *IACR Comm. Crypto.*, vol. 2, no. 1, 2025.

[51] Z. Espiritu, S. Kamara, T. Moataz, and V. Ogier, "Artifact for "Leafblower: a Leakage Attack Against TEE-Based Encrypted Databases"," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17114340

[52] Z. Espiritu, S. Kamara, T. Moataz, and A. Park, "PolySys: an Algebraic Leakage Attack Engine," in *Proc. 34th USENIX Sec.*, 2025.

[53] F. Falzon, E. A. Markatou, Akshima, D. Cash, A. Rivkin, J. Stern, and R. Tamassia, "Full Database Reconstruction in Two Dimensions," in *Proc. 2020 ACM CCS*, 2020.

[54] F. Falzon, E. A. Markatou, Z. Espiritu, and R. Tamassia, "Range search over encrypted multi-attribute data," *Proc. VLDB Endow.*, vol. 16, no. 4, 2022.

[55] Fortanix. (2025). [Online]. Available: https://www.fortanix.com/

[56] A. Furieri. (2025) SpatiaLite. [Online]. Available: https://www.gaia-gis.it/fossil/libspatialite/index

[57] A. Futoransky, D. Saura, and A. Waissbein, "The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms," in *Proc. 1st USENIX WOOT*, 2007.

[58] A. Fábrega, C. O. Pérez, A. Namavari, B. Nassi, R. Agarwal, and T. Ristenpart, "Injection Attacks Against End-to-End Encrypted Applications," in *Proc. 2024 IEEE S&P*, 2024.

[59] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, "SQLite: past, present, and future," *Proc. VLDB Endow.*, vol. 15, no. 12, 2022.

[60] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, 1996.

[61] Gramine Authors, "Gramine," 2024. [Online]. Available: https://gramine.readthedocs.io

[62] ——, "`get_node_numbers` routine: Gramine v1.8," 2024, SWHID: swh:1:cnt:2fbac44c1f6146b9285278023bd1f78550cb4d25;origin=https://github.com/gramineproject/gramine;visit=swh:1:snp:3d9d3753a5c0231ee393f9de6192bb3053f7f513;anchor=swh:1:rev:4629f068f68d2821607c6dc8d5d7502e0b2ab79a;path=/common/src/protected_files/protected_files.c;lines=335-377.

[63] P. Grubbs, M.-S. Lacharite, B. Minaud, and K. G. Paterson, "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries," in *Proc. 2018 ACM CCS*, 2018.

[64] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks," in *Proc. 2019 IEEE S&P*, 2019.

[65] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why Your Encrypted Database Is Not Secure," in *Proc. 16th Workshop HotOS*, 2017.

[66] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-Abuse Attacks against Order-Revealing Encryption," in *Proc. 2017 IEEE S&P*, 2017.

[67] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted Databases: New Volume Attacks against Range Queries," in *Proc. 2019 ACM CCS*, 2019.

[68] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *Proc. 10th Euro. Workshop Sys. Sec.*, 2017.

[69] D. R. Hipp. (2025) Most Widely Deployed SQL Database Engine. [Online]. Available: https://www.sqlite.org/mostdeployed.html

[70] M. Hogan, Y. Michalevsky, and S. Eskandarian, "DBREACH: Stealing from Databases Using Compression Side Channels," in *Proc. 2023 IEEE S&P*, 2023.

[71] Y. Hou, J. Li, Z. He, A. Yan, X. Chen, and J. McAuley, "Amazon reviews 2023: Video games 5-core," 2024. [Online]. Available: https://amazon-reviews-2023.github.io/data_processing/5core.html

[72] ——, "Bridging language and items for retrieval and recommendation," 2024, arXiv:2403.03952 [cs.IR].

[73] N. Hu, J. Zhang, and P. A. Pavlou, "Overcoming the J-shaped distribution of product reviews," *Comm. ACM*, vol. 52, no. 10, 2009.

[74] T. IEEE and T. O. Group. (2024) The Open Group Base Specifications Issue 8: fsync. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/fsync.html

[75] Intel, "Intel Software Guard Extensions (Intel® SGX) SDK for Linux OS 2.23," Intel, Tech. Rep., 2024. [Online]. Available: https://download.01.org/intel-sgx/sgx-linux/2.23/docs/Intel_SGX_Developer_Reference_Linux_2.23_Open_Source.pdf

[76] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES," in *Proc. 2015 IEEE S&P*, 2015.

[77] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *Proc. 2012 NDSS*, 2012.

[78] Q. Jiang and C. Wang, "Sync+Sync: A Covert Channel Built on fsync with Storage," in *Proc. 33rd USENIX Sec.*, 2024.

[79] M. Jurado and G. Smith, "Quantifying information leakage of deterministic encryption," in *Proc. 2019 ACM CCSW*, 2019.

[80] S. Kamara, A. Kati, T. Moataz, T. Schneider, A. Treiber, and M. Yonli, "SoK: Cryptanalysis of Encrypted Search with LEAKER – A framework for LEakage AttacK Evaluation on Real-world data," in *Proc. 2022 IEEE 7th Euro. S&P*, 2022.

[81] S. Kamara and T. Moataz, "Design and Analysis of a Stateless Document Database Encryption Scheme," MongoDB, Tech. Rep., 2023. [Online]. Available: https://www.mongodb.com/resources/products/capabilities/stateless-document-database-encryption-scheme

[82] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," AMD, Tech. Rep., 2021. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf

[83] A. Kaufman, M. Hershcovitch, and A. Morrison, "Prefix siphoning: Exploiting LSM-Tree range filters for information disclosure," in *Proc. 2023 USENIX ATC*, 2023.

[84] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic Attacks on Secure Outsourced Databases," in *Proc. 2016 ACM CCS*, 2016.

[85] S. Khan, I. Kabanov, Y. Hua, and S. Madnick, "A systematic analysis of the capital one data breach: Critical lessons learned," *ACM TOPS*, vol. 26, no. 1, Nov. 2022.

[86] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison-Wesley Longman, 1998.

[87] E. M. Kornaropoulos, N. Moyer, C. Papamanthou, and A. Psomas, "Leakage inversion," in *Proc. 2022 ACM CCS*, 2022.

[88] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution," in *Proc. 2020 IEEE S&P*, 2020.

[89] ——, "Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks," in *Proc. 2021 IEEE S&P*, 2021.

[90] M. Kowalczyk, D. Kuvaiskii, P. Marczewski, B. Popławski, W. Porczyk, D. E. Porter, K. Qin, C.-C. Tsai, M. Vij, and I. Yamahata, "Rapid Deployment of Confidential Cloud Applications with Gramine," in *Proc. 40th ACSAC*, 2024.

[91] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage," in *2018 IEEE S&P*, 2018.

[92] M.-S. Lacharité and K. G. Paterson, "A note on the optimality of frequency analysis vs. ℓ_p-optimization," Cryptology ePrint Archive, Paper 2015/1158, 2015.

[93] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS OS Rev.*, vol. 44, no. 2, 2010.

[94] K. G. Larsen and J. B. Nielsen, "Yes, there is an oblivious ram lower bound!" in *CRYPTO 2018*, 2018.

[95] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in *Proc. 15th Euro. Conf. Comp. Sys.*, 2020.

[96] F. Li, "Cloud-native database systems at Alibaba: opportunities and challenges," *Proc. VLDB Endow.*, vol. 12, no. 12, 2019.

[97] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP," in *Proc. 2022 IEEE S&P*, May 2022.

[98] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel," in *Proc. 30th USENIX Sec.*, 2021.

[99] ——, "TLB Poisoning Attacks on AMD Secure Encrypted Virtualization," in *Proc. 37th ACSAC*, 2021.

[100] M. Li, X. Zhao, L. Chen, C. Tan, H. Li, S. Wang, Z. Mi, Y. Xia, F. Li, and H. Chen, "Encrypted Databases Made Secure Yet Maintainable," in *Proc. 17th USENIX OSDI*, 2023.

[101] S. Lin, A. P. Marathe, P. Larson, C. Chen, C. Sun, P. Lee, W. Yu, J. Li, J. Meng, R. Lin, X. Chenxi, and Q. Zhuxii, "Near Data Processing in Taurus Database," in *Proc. 2022 IEEE ICDE*, 2022.

[102] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *Proc. 15th ACM Asia CCS*, 2020.

[103] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *Proc. 2015 IEEE S&P*, 2015.

[104] E. A. Markatou, F. Falzon, Z. Espiritu, and R. Tamassia, "Attacks on encrypted response-hiding range search schemes in multiple dimensions," *Proc. PETS*, vol. 2023, no. 1, 2023.

[105] E. A. Markatou, F. Falzon, R. Tamassia, and W. Schor, "Reconstructing with less: Leakage abuse attacks in two dimensions," in *Proc. 2021 ACM CCS*, 2021.

[106] E. A. Markatou and R. Tamassia, "Full Database Reconstruction with Access and Search Pattern Leakage," in *Info. Sec.*, 2019.

[107] ——, "Reconstructing with Even Less: Amplifying Leakage and Drawing Graphs," in *Proc. 2024 ACM CCS*, 2024.

[108] Microsoft. (2024) Transparent data encryption (TDE). [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption

[109] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies the Power of Cache Attacks," in *CHES 2017*, 2017.

[110] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, 1992.

[111] MongoDB. (2025) Encryption at Rest. [Online]. Available: https://www.mongodb.com/docs/manual/core/security-encryption-at-rest/

[112] A. Muñoz, R. Ríos, R. Román, and J. López, "A survey on the (in)security of trusted execution environments," *Comp. & Sec.*, vol. 129, 2023.

[113] MySQL. (2025) The InnoDB Storage Engine. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/innodb-storage-engine.html

[114] M. Naveed, S. Kamara, and C. V. Wright, "Inference Attacks on Property-Preserving Encrypted Databases," in *Proc. 22nd ACM CCS*, 2015.

[115] Neon. (2025). [Online]. Available: https://neon.tech/

[116] New York Office of Health Services Quality and Analytics, "Hospital Inpatient Discharges (SPARCS De-Identified): 2022," 2024. [Online]. Available: https://health.data.ny.gov/d/5dtw-tffi

[117] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," in *2016 IEEE Intl. Conf. CIC*, 2016.

[118] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996.

[119] J. Pei and V. Shmatikov, "Bigfoot: Exploiting and mitigating leakage in encrypted write-ahead logs," 2021, arXiv:2111.09374 [cs.CR].

[120] G. Persiano and K. Yeo, "Limits of Breach-Resistant and Snapshot-Oblivious RAMs," in *CRYPTO 2023*, 2023.

[121] PostGIS. (2023). [Online]. Available: https://postgis.net/

[122] PostgreSQL. (2025). [Online]. Available: https://postgresql.org

[123] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database Using SGX," in *Proc. 2018 IEEE S&P*, 2018.

[124] B. Schlüter, C. Wech, and S. Shinde, "Heracles: Chosen Plaintext Attack on AMD SEV-SNP," in *Proc. 2025 ACM CCS*, 2025.

[125] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *Proc. 2019 ACM CCS*, 2019.

[126] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss, "Practical Timing Side-Channel Attacks on Memory Compression," in *Proc. 2023 IEEE S&P*, 2023.

[127] SCONE. (2025). [Online]. Available: https://scontain.com/

[128] R. Seah, D. Khu, A. Hoover, and R. Ng, "LAMA: Leakage-Abuse Attacks Against Microsoft Always Encrypted," in *Proc. 21st SECRYPT*, 2024.

[129] A. Shahverdi, M. Shirinov, and D. Dachman-Soled, "Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite," in *Proc. 30th USENIX Sec.*, 2021.

[130] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Page Faults from Telling Your Secrets," in *Proc. 11th ACM Asia CCS*, 2016.

[131] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications with SGX Enclaves," in *Proc. 2017 NDSS*, 2017.

[132] C. Spearman, "The Proof and Measurement of Association between Two Things," *American J. Psych.*, vol. 15, no. 1, 1904.

[133] SQLite Authors. (2025) Database file format. [Online]. Available: https://www.sqlite.org/fileformat.html

[134] ——, "Key distribution in `balance_nonroot`: SQLite v3.49.1," 2025, SWHID: swh:1:cnt:1bd59a1b1fbc173298836772ac49e90409ec7531;origin=https://github.com/sqlite/sqlite;visit=swh:1:snp:2c586edae6bafc0069751f68571e03c88d46dfee;anchor=swh:1:rev:24fe85b99a975094d835ea3f895b16e572512ad4;path=/src/btree.c;lines=8450-8572.

[135] ——, "Page reassignment in `balance_nonroot`: SQLite v3.49.1," 2025, SWHID: swh:1:cnt:1bd59a1b1fbc173298836772ac49e90409ec7531;origin=https://github.com/sqlite/sqlite;visit=swh:1:snp:2c586edae6bafc0069751f68571e03c88d46dfee;anchor=swh:1:rev:24fe85b99a975094d835ea3f895b16e572512ad4;path=/src/btree.c;lines=8624-8664.

[136] ——, "SQLite," 2025, SWHID: swh:1:rev:3cd92ce875fd4e5601e535c35fef33494a6684e3;origin=https://github.com/sqlite/sqlite;visit=swh:1:snp:2c586edae6bafc0069751f68571e03c88d46dfee. [Online]. Available: https://sqlite.org

[137] ——. (2025) SQLite Encryption Extension: Documentation. [Online]. Available: https://www.sqlite.org/see/doc/release/www/readme.wiki

[138] ——, "`balance_quick` routine: SQLite v3.49.1," 2025, SWHID: swh:1:cnt:1bd59a1b1fbc173298836772ac49e90409ec7531;origin=https://github.com/sqlite/sqlite;visit=swh:1:snp:2c586edae6bafc0069751f68571e03c88d46dfee;anchor=swh:1:rev:24fe85b99a975094d835ea3f895b16e572512ad4;path=/src/btree.c;lines=7910-8027.

[139] ——. (2025) The SQLite R*Tree Module. [Online]. Available: https://www.sqlite.org/rtree.html

[140] ——. (2025) VACCUM. [Online]. Available: https://sqlite.org/lang_vacuum.html

[141] U. Telle, "SQLite3 Multiple Ciphers v2.1.0," 2025, SWHID: swh:1:rel:3a7b6ce42760ee0f4707497f5456273b8efeb532;origin=https://github.com/utelle/SQLite3MultipleCiphers;visit=swh:1:snp:50ababf6a7cb041fb0785c504ffc834204cdab7e.

[142] Texas Department of State Health Services, "Hospital Inpatient Discharge Public Use Data File: 2018, First quarter," 2018. [Online]. Available: https://www.dshs.texas.gov/center-health-statistics/texas-health-care-information-collection/

[143] Turso, "libSQL," 2025. [Online]. Available: https://turso.tech/libsql

[144] United States Census Bureau. (2025) QuickFacts: Texas (vintage year 2024). Archived Apr. 12, 2025. [Online]. Available: https://web.archive.org/web/20250412161446/https://www.census.gov/quickfacts/fact/table/TX/RHI125223#RHI125223

[145] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," in *Proc. 2019 IEEE S&P*, 2019.

[146] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," in *Proc. 2021 IEEE S&P*, 2021.

[147] S. Van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, D. Genkin, A. Miller, E. Ronen, Y. Yarom, and C. Garman, "SoK: SGX.Fail: How Stuff Gets eXposed," in *Proc. 2024 IEEE S&P*, 2024.

[148] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," in *Proc. 2017 ACM SIGMOD*, 2017.

[149] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "StealthDB: a Scalable Encrypted Database with Full SQL Query Support," *Proc. PETS*, vol. 2019, no. 3, 2019.

[150] L. Wang, P. Grubbs, J. Lu, V. Bindschaedler, D. Cash, and T. Ristenpart, "Side-Channel Attacks on Shared Search Indexes," in *Proc. 2017 IEEE S&P*, 2017.

[151] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proc. 2017 ACM CCS*, 2017.

[152] Y. Wang, Y. Shen, C. Su, J. Ma, L. Liu, and X. Dong, "CryptSQLite: SQLite With High Data Security," *IEEE Trans. Comp.*, vol. 69, no. 5, 2020.

[153] WiredTiger. (2025) WiredTiger. [Online]. Available: https://source.wiredtiger.com/

[154] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proc. 2015 IEEE S&P*, 2015.

[155] Y. Yan, W. Huang, I. Grishchenko, G. Saileshwar, A. Mehta, and D. Lie, "Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels," in *Proc. 34th USENIX Sec.*, 2025.

[156] X. Yang, C. Yue, W. Zhang, Y. Liu, B. C. Ooi, and J. Chen, "SecuDB: An In-Enclave Privacy-Preserving and Tamper-Resistant Relational Database," *Proc. VLDB Endow.*, vol. 17, no. 12, 2024.

[157] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels," in *Proc. 2024 ACM CCS*, 2024.

[158] ——, "CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels," in *Proc. 2025 IEEE S&P*, 2025.

[159] Zetetic, "SQLCipher," 2025, SWHID: swh:1:rel:f7632dc0d8f858d6c1a17f5837bb4e01a5dd1abc;origin=https://github.com/sqlcipher/sqlcipher;visit=swh:1:snp:15cdef72c67f91077c44bc287f311fd7d9047d97.

[160] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based Fault Injection using Selective State Reset," in *Proc. 33rd USENIX Sec.*, 2024.

Appendix A. Other Databases

As a proof-of-concept for the generalizability of our techniques, we briefly consider how our attack may work on other B$^+$-tree databases. Our attack can also be generalized to other B$^+$-tree databases under the n-snapshot model by adapting the rebalancing and node insertion steps to the specific behaviors of the targeted B$^+$-tree implementation.

SQLite Encryption Extension (SEE). SEE [137] is the official transparent encryption mechanism for SQLite. We did not evaluate our attacks on SEE because it costs $2000 to access the source code. However, the documentation states that SEE encrypts files in a way that preserves the original page-alignment, just as in SQLC and S3MC. Consequently, we hypothesize our methodology immediately generalizes with an identity function as the inversion function.

libSQL. libSQL by Turso [143] is a open-source fork of SQLite with additional features such as replication and support for disaggregated storage. libSQL is based off of SQLite and thus shares nearly identical B$^+$-tree behavior. It includes its own encrypted storage engine which is built on SQLite3 Multiple Ciphers [141]. Because of this, we believe LEAFBLOWER (using the S3MC inv function) generalizes immediately against libSQL without any modifications.

PostgreSQL and others. Due to space restrictions, we briefly describe some differences as to how our attack generalizes to PostgreSQL [122] and leave a complete description to the full version. Surprisingly, PostgreSQL is *easier* to attack than SQLite since it stores each B$^+$-tree index in separate files. Thus, no disambiguation is needed and the leaf ordering algorithm from Section 5.2 works with a simple adaptation to the rebalance procedure. This also means that, unlike in SQLite, the attack immediately generalizes to PostgreSQL tables with multiple indexed columns.

There are some limits on the adversarial model in this setting due to PostgreSQL's use of a *write-ahead log* (WAL) [110]. Unlike the default behavior of SQLite (which updates the index after every operation), a PostgreSQL index is not updated immediately—updates are instead appended to a separate *WAL file*. Every 5 minutes or when the WAL reaches a size limit, a *checkpoint* occurs where the contents of the WAL are copied back into the existing index. The WAL has a very minimal structure and (at least in the multi-snapshot model we consider) the checkpointing process reveals minimal information about the index structure.

To be clear, the n-snapshot model still works if each checkpoint contains only one operation to the index of interest—we just believe that this is a less likely model than in SQLite given the constraints above, so we do not make these databases the main focus of this paper. Nevertheless, our results show that our attack may generalize to other B$^+$-tree-based storage engines since they exhibit similar structural properties. Such databases (e.g. MongoDB [153], MySQL [113]) have their own ESEs and may also be run under enclaves using a LibOS.

Appendix B. Inverting Gramine with 4 KiB SQLite Pages

In this section, we elaborate on the concerns described in Section 4 about how Gramine's use of a 1 KiB offset at the start of encrypted files can introduce additional ambiguities when SQLite's logical database page size P is set to 4 KiB. Consider such a database deployment and any contiguous sequence of SQLite pages of size 4 KiB p_1, p_2, p_3 (that is, $p_1 + 2 = p_2 + 1 = p_3$). This sequence of three pages spans a set of four 4 KiB blocks $\{j_1, j_2, j_3, j_4\}$ in the Gramine encrypted file format. Then, consider the inversion process for the plaintext delta Δ_i for an encrypted delta Δ_i^{enc} that includes all four of these Gramine blocks (that is, $\{j_1, j_2, j_3, j_4\} \subseteq \Delta_i^{\mathsf{enc}}$). We can easily see that $p_1 \in \Delta_i$ and $p_3 \in \Delta_i$, but it is not possible to definitively determine whether $p_2 \in \Delta_i$. At first glance, the additional noise introduced by these cases could theoretically lead to a loss of accuracy in the attack.

Fortunately, our attack methodology already provides us with several mechanisms that we can use to disambiguate nearly all of these cases. Our algorithm already makes use of several structural properties of B$^+$-tree insertions that, if violated, alert us that our initial decision for an ambiguity was incorrect—in particular, recall the *backtracking* mechanism that we use to handle the ambiguous "imposter" leaf cases discussed in Section 5.2. As such, we can use similar backtracking techniques to handle this edge case. Specifically, for each ambiguous Δ_i^{enc}, our attack makes an initial guess as to whether $p_2 \in \Delta_i^{\mathsf{enc}}$. Then, we attempt to run the disambiguation and restructuring portions of the attack as normal. If processing an ambiguous Δ_i^{enc} results in a structural violation, we backtrack and toggle our original guess for $p_2 \in \Delta_i^{\mathsf{enc}}$. We repeat this process until we find a "satisfiable" assignment of encrypted deltas Δ_i^{enc}.

Theoretically, we may need to try every possible assignment of ambiguous Δ_i^{enc}'s in the worst-case. In practice, this is not a concern since (1) we experimentally found that using a conservative approach (where we guess $p_2 \notin \Delta_i^{\mathsf{enc}}$ by default) results in a relatively small number of deltas that actually need backtracking in many workloads and

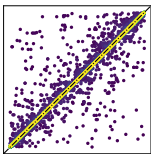
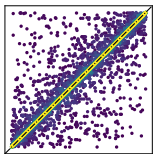
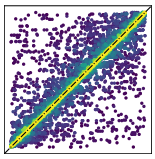
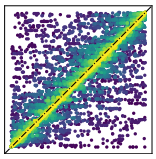
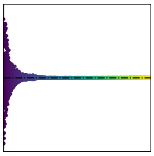
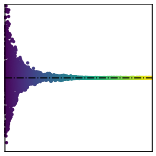
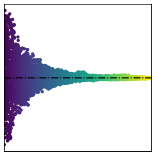
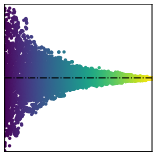
Page Size $P =$		4096			8192			16384			32768		
Guess vs Actual Value (diagonal is perfect guess)													
Error vs Transcript Index (midline is 0 error)													
		LB	NV	Ad	LB	NV	Ad	LB	NV	Ad	LB	NV	Ad
ERecov% (higher better)	Q1	14	0.4	13	3	0.4	2	1	0.4	0	1	0.4	0
	Q2	38	0.4	37	19	0.4	18	4	0.4	3	1	0.4	0
	Q3	64	0.4	63	21	0.4	20	15	0.4	14	3	0.4	2
	Q4	88	0.4	87	29	0.4	28	18	0.4	17	7	0.4	6
AvgAbs Err% (low better)	Q1	5.4	33		11	33		18	33		26	33	
	Q2	0.2	33		0.8	33		3	33		10	33	
	Q3	0.1	33		0.3	33		0.9	33		4	33	
	Q4	0.0	33		0.3	33		0.5	33		1	33	
	$r_s =$	0.977			0.950			0.903			0.815		

Figure 9: Evaluation on synthetic, random 1 byte `INT` datasets (where $D = 2^8$, $n = 1000000$, and $\max(s) = 1$), but with different page sizes $P \in \{4096, 8192, 16384, 32768\}$. As the page size increases, the exact recovery rate decreases and the average error increases. This shows that a potential mitigation for the attack is to increase the page size from the default $P = 4096$ to a larger page size. However, this is not necessarily a robust mitigation in the long-term, as the error graphs show that the error minimizes as the transcript length increases.

(2) such ambiguities primarily appear very early in the operation transcript (when the size of the database index is small). Furthermore, manual inspection of the contiguity violations can usually avoid significant exploration of the search space. Since this is an edge case, we opt for a simpler implementation of this technique in our attack—when violations occur, the attacker manually inspects the contiguity violations and input a new guesses that can reduce the backtracking considerably. Once all violations are resolved, we find that the accuracy of our attacks is not affected by these ambiguities. This shows that the "structural leakage" methodology (leveraging expected properties of the underlying data structure to identify possible errors) can actually *correct* "noisy" leakage. This gives further credence to this attack methodology for future work.

Appendix C. Addressable Linked List with Rebalance

The LeafOrganizer data structure (denoted LO) used in Algorithm 2 is instantiated as an *addressable doubly-linked list* and captures the current order of the index tree's leaves (and the values in each leaf). Specifically, each LO node n_p corresponds to a node p in the index tree. Each n_p maintains a sequence of operation identifiers corresponding to the insertions whose values are (currently) contained within p. A LeafOrganizer supports the following operations:

- $\mathsf{Add}(p_i, j)$, which adds an operation identifier j to the node n_i (corresponding to the B$^+$-tree leaf p).
- $\mathsf{IsConsistent}(p_1, \ldots, p_k)$, which returns true if the nodes $n_p, \ldots, n_k$ are currently contiguous in LO and false otherwise.
- $\mathsf{Rebalance}(p_1, \ldots, p_k)$, which:
  1) Verifies that $\mathsf{IsContiguous}(p_1, \ldots, p_k) = \text{true}$.
  2) Reorders the nodes $\{n_1, \ldots, n_k\}$ so that they are in the order $n_1, \ldots, n_k$.
  3) Redistributes the operation identifiers contained in the nodes in $\{n_1, \ldots, n_k\}$ in a way that approximates[10] the redistribution algorithm used in the B$^+$-tree rebalance operation.
- $\mathsf{Enumerate}()$, which outputs the operation identifiers stored in the list in their current order.

The complete pseudocode of the LeafOrganizer is contained in the full version and the artifact [51].

10. As shown in [134], the actual redistribution algorithm implementation is more complex than simply evenly dividing the keys between the nodes because it needs to handle the redistribution of variable-length keys. Since the encoded length of a key is not learned as part of our attack, we simplify and implement the redistribution algorithm as if it only handled fixed-length keys. We find that, in practice, this is enough to maintain a good approximate order reconstruction for the purposes of Algorithm 2.

Appendix D.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper explores a new leakage abuse attack against encrypted SQLite databases that use B-tree data structures. The attack assumes it sees a snapshot of the encrypted pages as seen from the file system after each update operation. The attack recovers the structure of the underyling trees and, from that, the approximate order of plaintext values stored. From this they can use auxiliary data to mount an inference attack to guess the values. They evaluate their attack on synthetic and real data, showing it works well in many cases.

D.2. Scientific Contributions

The paper provides new insights on leakage abuse in a setting that has not previously been explored, for practically important targets (encrypted SQLite databases).

D.3. Reasons for Acceptance

1) No prior work has shown attacks that exploit this kind of leakage.
2) This paper adds to the evidence that the "lift-and-shift" model of TEE deployments has pitfalls in terms of side channel attacks.
3) The attack techniques themselves are notable and of potential broader value, showing how in the weaker-than-usually-assumed leakage model an adversary can recover approximate ordering information from write accesses to B-tree data structures.