Snapshots of TEE-Based Encrypted Databases Leak Plaintext Histograms via Dendrochronology

(Full Version)

Zachary Espiritu
MongoDB Research
New York, New York, USA
New York University
New York, New York, USA
zachary.espiritu@mongodb.com

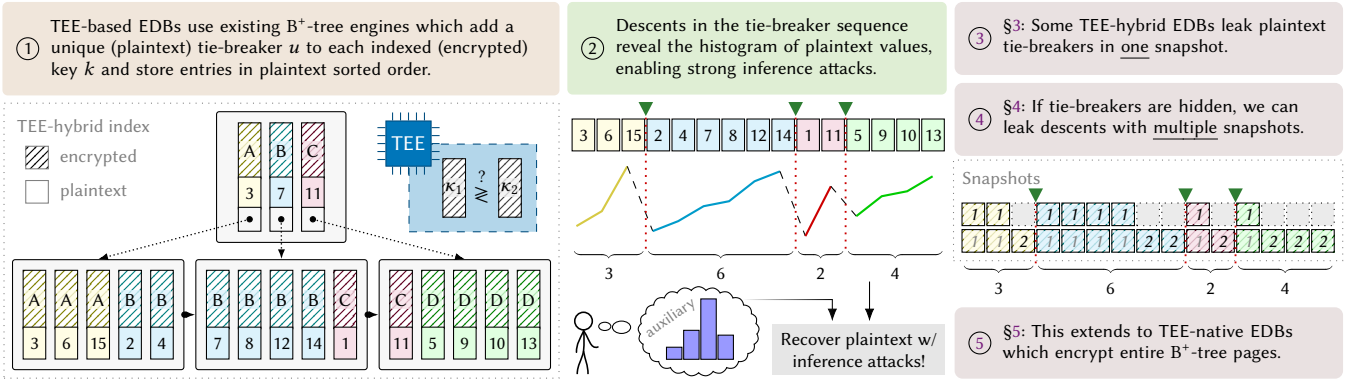

Figure 1: Overview of this paper and the *dendrochronology* vulnerability.

Abstract

Arborists use *dendrochronology*, or the study of tree "growth rings" (as in, the plant), to infer past environmental events. We identify an analogous, exploitable phenomenon in the indexes used by several *encrypted databases* (EDBs) based on *trusted execution environments* (TEEs). Such EDBs have been shown to leak the *order* of indexed values, but not the histogram (or *frequency* of each value), with respect to a weaker class of *snapshot adversaries*. This *order-only* leakage substantially reduces the impact of possible inference attacks.

We show, in fact, *frequency is recoverable* in the snapshot model from many TEE-based EDBs, even if the TEE itself is "secure". Our insight is simple: practical storage engines impose a strict total order on indexed keys by augmenting each key with a unique tie-breaker. If the adversary learns information about the tie-breakers, they can recover the frequencies of plaintext keys. We show this enables stronger inference attacks on several TEE-based EDBs (including Microsoft SQL Server's Always Encrypted, StealthDB, and HeDB) to demonstrate this vulnerability's ubiquity and point to broader lessons for TEE applications designed around legacy code.

CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands*, https://doi.org/10.1145/3830454.3846542.

This full version of [42] contains expanded Ethical Considerations in Appendix B, the proof of Theorem 3.2 in Appendix C, and discussion of other TEE-hybrid databases in Appendix D.

1 Introduction

Demand for stronger protections on outsourced data motivates the field of *encrypted databases* (EDBs), where data is end-to-end encrypted between a trusted client and an untrusted database cluster, and (ideally) no information about plaintext data or queries is exposed to the cluster. Cryptographers have used various primitives to design EDBs such as property-preserving encryption [2, 15], structured encryption [26, 31], oblivious data structures [60], and trusted execution environments (TEEs) [27, 29, 82, 95].

Constructions based on the first three primitives involve well-known tradeoffs and have been extensively cryptanalyzed in prior work [80]. In contrast, there is significantly less targeted analysis of EDBs based on TEEs, which provide a hardware-backed abstraction for encrypted code and memory. Nevertheless, TEE-based EDBs have seen large interest from both academic and industrial parties—likely because they significantly simplify EDB design compared to purely cryptographic solutions—and are used in practice by various financial [133], healthcare [111], and compliance [62] organizations.

While TEE-based EDBs themselves have largely eluded targeted cryptanalysis, there is a rich line of attacks that focus on the general TEE primitive itself. These show that powerful, *persistent* adversaries who can compromise various kernel components or execute

arbitrary code on the victim can break the confidentiality guarantees of TEEs via microarchitectural side channels [20, 32, 70, 77, 103, 104, 112, 121, 140, 153], transient execution [21, 22, 100, 129, 144, 148], ciphertext-level vulnerabilities [99, 123, 149, 152], syscall/signal attacks [122, 131], interconnect corruption attacks [59, 124], and physical memory bus interposition [28, 35, 94, 127].

But these are not the only adversaries relevant for practical databases. Recent commercial systems emphasize weaker threat models in which an attacker exfiltrates point-in-time *snapshots* of database state without fully controlling the live compute stack [81]. Unlike persistent models which capture sustained server compromise, snapshot models capture incidents arising from database user or server administrator compromise, such as the 2019 Capital One [87] and 2022 LastPass breaches [56]. In response, emerging work [45, 126] studies whether the security claims of TEE-based databases still hold under weaker *snapshot adversaries* that obtain one or more point-in-time views of the EDB state but do not observe fine-grained execution during operations.

Two snapshot axes matter in this literature. First, the snapshot may expose either a view of the encrypted index as represented in memory (*internal-memory snapshots*) or in persistent disk storage (*external-memory snapshots*). Second, the adversary may receive either one (*single-snapshot*) or multiple snapshots (*q-snapshot*). The models used to analyze a TEE-based EDB depend on its architecture, which we classify across two broad categories:

- *TEE-hybrid systems* [6, 7, 12, 101, 132, 141, 143, 156], which keep most existing DBMS logic outside the enclave and only invoke the TEE for comparisons over encrypted data. For example, Microsoft Always Encrypted [6] and Alibaba's AlwaysConfidential [143] are TEE-hybrid EDBs deployed in practice.
- *TEE-native systems* (e.g., [120, 146, 151]), where the entire DBMS executes inside of the TEE. This also includes "lift-and-shift" systems that run an existing, unmodified DBMS in a TEE using virtualization [27, 82] or a *libOS* [9, 13, 130, 135]. Some "lift-and-shift" databases have been commercialized [5, 51].

TEE-hybrid systems which use B$^+$-trees have been shown to leak order—but not frequency—when the adversary gets a *single snapshot* of the internal-memory index (e.g., Seah, Khu, Hoover, and Ng's Lama attack [126] against Always Encrypted). Outside of the snapshot model, many TEE-hybrid EDBs (e.g., [6, 7, 101, 141, 156]) state that a *persistent* adversary that can observe query execution is required to learn frequency information, and only learns frequency information about data that is actively queried. Other EDBs (e.g., [143]) only explicitly analyze frequency leakage in the persistent model and do not make any claims about snapshot adversaries.

In TEE-native systems, Espiritu, Kamara, Moataz, and Ogier's Leafblower [45] attack shows that a *multi-snapshot* adversary can exploit B$^+$-tree invariants to learn approximate *order-only* leakage from snapshots of encrypted index files. Leafblower explicitly conjectures that frequency information is impossible to recover without a stronger adversary.

Why does frequency matter? To understand why frequency leakage is dangerous, it is useful to compare it to *order-only* leakage. Suppose an encrypted index contains a multiset of encrypted keys whose plaintexts are from a *key space* $\mathcal{K} = \{\kappa_1 < \cdots < \kappa_m\}$. (We assume m is the number of present, distinct keys.) For simplicity, suppose the adversary sees four ciphertexts in sorted order

$$\langle \mathrm{ct}_A, \mathrm{ct}_B, \mathrm{ct}_C, \mathrm{ct}_D \rangle.$$

Under order-only leakage, the adversary learns the relative rank of each plaintext key, but not which keys are equal to each other. This small example of an order-only view is consistent with many different plaintext multisets:

$$\{\kappa_1, \kappa_1, \kappa_1, \kappa_1\}, \quad \{\kappa_1, \kappa_2, \kappa_2, \kappa_3\}, \quad \{\kappa_1, \kappa_2, \kappa_3, \kappa_4\}, \quad \text{etc.}$$

All of these produce the same order leakage, but they have very different equality structure and frequencies. From an information-theoretic perspective, an adversary cannot tell from order alone the number of distinct keys in the dataset.

Frequency leakage significantly changes this picture. We will capture frequency leakage as a *vector* of each unique value's cardinality. Specifically, let $\vec{\mathbf{f}} = (f_1, \ldots, f_m)$ denote the frequency vector, where f_i is the number of occurrences of κ_i. Unlike in the order-only case, the adversary can "align" their auxiliary information about the plaintext distribution to the histogram defined by $\vec{\mathbf{f}}$. Furthermore, since $|\vec{\mathbf{f}}| = m$, the vector also reveals the number of unique keys. This allows for significantly more accurate reconstruction.

This observation enables a line of inference attacks against *order-revealing* and *order-preserving encryption* (OPE) [2, 18, 97]. Naveed, Kamara, and Wright [114] presented the first attacks which combined the frequency and order leakage of OPE with *auxiliary information* to reconstruct plaintexts. Later works improved these attacks [16, 40, 67, 145]. However, these inference techniques are not applicable without frequency information—with order-only leakage, reconstruction significantly degrades unless the adversary's auxiliary information is very accurate [45, 126]. This observation also has motivated the development of *frequency-hiding OPE* (FH-OPE) schemes [85, 98], but such schemes are largely theoretical and impractical for deployment. As such, claims of order-only leakage for TEE-based EDBs are significant: *they exclude the frequency information which gives classical inference attacks most of their power.*

1.1 Contributions

We correct the record on order-only leakage in TEE-based EDBs. We identify several TEE-based EDBs that have been either independently cryptanalyzed as having order-only leakage under a snapshot model or require a much stronger, persistent model to recover frequency leakage. For these EDBs, we identify a surprisingly simple idea that allows for frequency recovery in the snapshot model—practical DBMS storage engines must enforce a *strict total order* among index entries. To do this, they break ties by augmenting each key with a *tie-breaker* such as a record ID. Although the tie-breaker in itself is not necessarily sensitive, we show that the stored order of tie-breakers reveals boundaries between distinct plaintext values and allows frequency recovery.

We call this phenomenon *dendrochronology*. Just as the "growth rings" of a tree's trunk reveal the history of external environmental events [52], descents in the sequence of tie-breakers reveal historical information about how the encrypted index was populated. In particular, if the tie-breaker sequence decreases between two adjacent entries in final sorted order, then those entries cannot have the same plaintext key. Such descents therefore reveal plaintext key boundaries and allow an adversary to recover the ordered frequency

vector, or an approximation of it. This is enough to turn many purported *order-only* leakage profiles into *order-plus-frequency* leakage profiles, enabling substantially stronger inference attacks.

We will spend the bulk of the paper discussing TEE-hybrid EDBs. In Section 2, we will discuss necessary DBMS indexing background and a unifying architecture for the TEE-hybrid databases we examine in this work. Then, in Section 3, we show that several TEE-hybrid systems directly leak the associated record ID tie-breakers to a *single-snapshot adversary* who takes a point-in-time snapshot of the indexing data structure in the untrusted *internal-memory* of the DBMS. We directly validate our attacks on Microsoft SQL Server's Always Encrypted [6], StealthDB [141], and HeDB [101].

We then consider a natural mitigation for our first attack in Section 4, where we remove the plaintext tie-breakers from the index structure. While this prevents our single-snapshot attack, we show that a slight strengthening of our adversary still permits frequency recovery. Specifically, we show that a *multi-snapshot adversary* who receives multiple snapshots of the index over time can synthesize a "proxy" for the record ID tie-breaker by exploiting the fact that ciphertexts are not rewritten between snapshots.

As a final demonstration, in Section 5 we show that our attack generalizes to TEE-native EDBs, where the entire DBMS is executed inside of the TEE. Here, the adversary no longer sees internal-memory and thus cannot see individual encrypted records. However, we show that a *multi-snapshot external-memory adversary* can extend the attack of Leafblower [45] (which recovers approximate *order-only* leakage in the same model) using the same dendrochronology technique. This allows us to recover approximate frequency information which increases reconstruction accuracy.

We discuss related work in Section 6 and point towards broader lessons in Section 7. Overall, our work demonstrates how reusing existing plaintext database machinery can inherit subtle algorithmic invariants that become unintended vectors for leakage.

Limitations. Our attacks—which exploit properties of B$^+$-tree index implementations—necessarily only work on *indexed* attributes. In some TEE-hybrid systems, TEE-based comparison and indexing are an explicit opt-in per attribute, and such systems support *non-indexed* encrypted columns—our attack does not apply to such non-indexed attributes. Finally, our multi-snapshot attacks largely assume that index tie-breakers are *monotonically increasing*. By default, this is true for all EDBs discussed in this work.

Disclosure. We disclosed our findings to relevant industry vendors (Microsoft and Alibaba Cloud). We discuss disclosure in Appendix B.

2 Background

As the first part of this paper focuses on TEE-hybrid EDBs, we discuss only preliminaries necessary for TEE-hybrid constructions and defer discussion of TEE-native preliminaries to Section 5.

2.1 Plaintext B$^+$-Tree Storage Engines

B$^+$-trees. Databases use different external memory data structures for persistently storing and searching over key-value pairs—all of the databases discussed in this paper use *B$^+$-trees* [14]. A *B$^+$-tree of order $z \geq 2$* is a tree which maps *search keys* to *values*. Each *leaf node* contains an ordered sequence of $w \leq z$ totally ordered search keys $k_1 < k_2 < \cdots < k_w$, their associated values, and a *sibling pointer* to the next leaf node in key order. The sibling pointers logically form a linked list over the leaf nodes. Scanning over this list from left-to-right provides the *sorted key sequence* of the tree.

Usage of B$^+$-trees. Each database table is associated with one *record tree*, which maps record IDs to record data. Each indexed attribute on the table is associated with one *index tree* which maps arbitrary search keys to record IDs. When a query performs a search over an indexed attribute, the query engine first accesses the index tree to determine the IDs of records whose queried attribute matches the queried value, then uses those record IDs to query the record tree to retrieve the corresponding record data.

Tie-breakers. To the view of database users, a B$^+$-tree index can either be *unique* (all keys are distinct) or *non-unique* (duplicate keys are allowed). However, regardless of the uniqueness of user-facing keys, essentially all practical B$^+$-tree storage engines enforce uniqueness of physically stored keys. This invariant's ubiquity comes from its advantages—it heavily simplifies the implementation of fine-grained locking needed to support concurrency [61, 96, 147] (and is even assumed by some algorithms [96]) and meaningfully reduces write amplification [57].

Thus, to satisfy the uniqueness invariant in non-unique indexes, databases extend the user-visible key k with a unique *tie-breaker* component u. This yields an *augmented key* (k, u) that is unique and totally ordered. Consequently, the B$^+$-tree entries are lexicographically sorted by (k, u):

$$(k, u) < (k', u') \iff (k < k') \lor (k = k' \land u < u').$$

Databases can instantiate this tie-breaker in different ways. As an example, the tie-breaker in Microsoft SQL Server's index trees is the record ID associated with each key [36]. Critically, in all of the databases we discuss, this tie-breaker is *strictly increasing* with each insertion by default. (This monotonicity property improves write locality in the index.) Thus, given two keys k_i and k_j inserted at timesteps i and j respectively, if $i < j$, then their associated tie-breakers u_i and u_j must follow $u_i < u_j$.

> *Observation 1.* By default, index B$^+$-tree tie-breakers are strictly increasing with each insertion.

2.2 TEE-Hybrid Indexes

TEE-hybrid EDBs [6, 7, 12, 101, 132, 141, 143, 156] preserve existing DBMS logic over encrypted data by executing comparison operations inside of a TEE. While there are slight variations in how each database is implemented, all of the databases we discuss in this work utilize the same architecture which we describe below.

Each database augments the client with an encryption-aware driver which knows the protected table's schema and the *encryption key* used to encrypt each attribute. When a client issues a query containing a protected attribute, the driver symmetrically encrypts the plaintext value with the attribute's encryption key. The query containing encrypted attributes is then sent to the server. In order for the TEE to decrypt the ciphertexts for comparisons, the driver attests and establishes a secure connection with the TEE—it uses

this connection to securely transmit the encryption key to the TEE. (The encryption key is cached for subsequent operations.)

Aside from the TEE-based comparisons and client-side encryption logic, the existing DBMS works essentially identically to the plaintext setting. This means that, when a protected attribute is indexed, its index B$^+$-tree maps *encrypted* search keys to plaintext record IDs. Due to the comparison operator, search key ciphertexts are stored in sorted *plaintext* order.

Importantly, just as with normal B$^+$-trees, the plaintext record IDs are used as a tie-breaker when two search keys are equal to each other (as determined by the TEE comparator). This means that when two encrypted search keys have the same plaintext k, their corresponding tie-breakers u and u' are used to uniquely position the search key in the tree.

Observation 2. The order of entries in the TEE-hybrid encrypted index B$^+$-tree is defined by lexicographically sorting by (k, u) over plaintext search keys k and their associated record ID u.

A final, critical observation comes from the storage of the tie-breakers—it turns out in the TEE-hybrid databases we examine in this work, the B$^+$-tree tie-breakers are stored in plaintext, even when the search keys are stored in encrypted form.

Observation 3. Tie-breakers are stored in plaintext in the TEE-hybrid encrypted index B$^+$-tree.

3 Dendrochronology with One Snapshot

In this section, we introduce our *dendrochronology attack* against TEE-hybrid databases. We demonstrate that the tie-breaker behavior inherited from the original DBMS leads to a simple attack in the single-snapshot model which allows us to recover some, if not all, of the key frequency information in the database.

Notation and threat model. We start with our weakest adversary, the *single-snapshot internal-memory adversary* [126] which obtains a single point-in-time snapshot of the DBMS state—specifically, a snapshot of the in-memory encrypted index B$^+$-tree. Formally, we say that after the database has processed a workload of n insertions $\langle \mathsf{op}_1, \ldots, \mathsf{op}_n \rangle$, the adversary obtains T_n, the in-memory B$^+$-tree after the nth operation:

$$\mathsf{View}_{\Pi}^{1,\mathsf{int}}\big(\langle \mathsf{op}_1, \ldots, \mathsf{op}_n \rangle\big) := T_n.$$

The adversary can parse T_n to obtain its sorted encrypted key sequence and corresponding plaintext tie-breakers, which we denote

$$\mathsf{Parse}(T_n) \to \Big\langle \big(\mathsf{Enc}(k_1), u_1\big), \ldots, \big(\mathsf{Enc}(k_n), u_n\big) \Big\rangle.$$

This adversary can arise naturally in practice: many real-world database deployments expose low-level index state to service accounts for administration or backup purposes [81], and, later in this section, we will show for different databases how such accounts can extract this state through the normal DBMS query interface. (We stress the adversary does not issue its own database operations, does not observe code execution during operations, and does not have access to the encryption keys.)

The single-snapshot attack. We now show a simple frequency recovery attack. Consider the sorted encrypted key sequence

$$\Big\langle \big(\mathsf{Enc}(k_1), u_1\big), \ldots, \big(\mathsf{Enc}(k_n), u_n\big) \Big\rangle.$$

Then, given any fixed plaintext search key k, all encrypted instances of k form a contiguous *block* in the sequence, and the B$^+$-tree's lexicographic ordering means that the tie-breakers must monotonically increase across the block. This means that, given two adjacent entries $(\mathsf{Enc}(k_j), u_j)$ and $(\mathsf{Enc}(k_{j+1}), u_{j+1})$,

$$u_j > u_{j+1} \implies k_j \neq k_{j+1}.$$

We call such an occurrence a *descent* at index j.[1]

Observation 4. Descents in the tie-breaker sequence between two adjacent entries $(\mathsf{Enc}(k_j), u_j)$ and $(\mathsf{Enc}(k_{j+1}), u_{j+1})$ in the TEE-hybrid encrypted index B$^+$-tree imply $k_j \neq k_{j+1}$.

Observation 4 leads to the following attack, illustrated in Figure 1. First, we extract the set of indices where descents occur:

$$\text{descents} := \big\{\, j \in \{1, \ldots, n-1\} : u_j > u_{j+1} \,\big\}.$$

Writing the descent indices in order as $d_1 < \cdots < d_t$ and setting $d_0 = 0$ and $d_{t+1} = n$, we calculate the number of entries between each descent as

$$\vec{\mathbf{f}}' = \Big(d_i - d_{i-1}\Big)_{i \in \{1, \ldots, t+1\}}.$$

We will show that $\vec{\mathbf{f}}'$ is an approximation of (and possibly exactly recovers) the true frequency vector $\vec{\mathbf{f}}$.

Formalizing the approximation. It will be convenient for us to formalize this frequency vector approximation in the language of a combinatorial object known as a *composition* [71], which is a vector of integers which sum to a given integer. We can view a frequency vector as a composition of n, the total number of records.

Definition 1 (Composition). *A* composition *of a positive integer n is an ordered tuple of positive integers $\vec{\mathbf{c}} = (c_1, \ldots, c_t)$ such that*

$$\sum_{i=1}^{t} c_i = n.$$

Given two compositions $\vec{\mathbf{a}}$ and $\vec{\mathbf{b}}$, we say that $\vec{\mathbf{b}}$ is a *coarsening* of $\vec{\mathbf{a}}$ if $\vec{\mathbf{b}}$ can be obtained by summing consecutive elements of $\vec{\mathbf{a}}$ (i.e., merging together adjacent frequencies).

Definition 2 (Coarsening). *Let $\vec{\mathbf{a}} = (a_1, \ldots, a_m)$ and $\vec{\mathbf{b}} = (b_1, \ldots, b_t)$ be compositions of n. We say $\vec{\mathbf{b}}$ is a* coarsening *of $\vec{\mathbf{a}}$ if there exist indices $0 = i_0 < i_1 < \cdots < i_t = m$ such that*

$$b_\ell = \sum_{j=i_{\ell-1}+1}^{i_\ell} a_j \qquad \text{for each } \ell \in \{1, \ldots, t\}.$$

We can then use this to formalize the approximation behavior: our attack recovers a *coarsening* of the true frequency vector $\vec{\mathbf{f}}$.

Theorem 3.1. *$\vec{\mathbf{f}}'$ is a* coarsening *of the true frequency vector $\vec{\mathbf{f}}$.*

[1] Graphing the tie-breakers and observing the descents visually parallels how the patterns of tree rings reveal information about events that have occurred in a tree's external environment, hence the name *dendrochronology* [52].

Proof. Let $B_1, \ldots, B_m$ be the blocks of equal keys in sorted order, so $\vec{\mathbf{f}} = (|B_1|, \ldots, |B_m|)$. Within any block, lexicographic ordering by (k, u) forces the tie-breakers to strictly increase (Observation 2), so no descent occurs inside a block and every index in descents lies on a block boundary. As each part of $\vec{\mathbf{f}}'$ counts the entries between consecutive descents, and descents fall only on block boundaries, each part of $\vec{\mathbf{f}}'$ must be the sum of a consecutive run of block lengths. This means $\vec{\mathbf{f}}'$ is a coarsening of $\vec{\mathbf{f}}$. □

We will empirically evaluate the accuracy of this coarsening on various workloads in our evaluation later in the paper.

As a reminder, once we have recovered the frequency vector, we can combine this with order leakage to instantiate inference attacks on the ciphertexts using standard OPE inference attacks [16]. We defer detailed discussion of this to our evaluation.

Remark: when are exact frequencies recoverable? Theorem 3.1 provides a lower bound on what the adversary can learn about frequencies based on its knowledge of the tie-breakers, but it does not guarantee that the adversary learns the exact frequency vector since key boundaries may not necessarily result in a descent.

For example, suppose we insert keys in sorted order. In this case, the tie-breakers u_i and u_{i+1} for any pair of adjacent keys in the sorted key sequence k_i and k_{i+1} will always satisfy $u_i < u_{i+1}$ since the tie-breakers are monotonically increasing by definition. Then, there are no descents (descents $= \varnothing$), which results in the degenerate approximate frequency vector $\vec{\mathbf{f}}' = (n)$.

We can capture the conditions where exact recovery holds under Observation 1—the tie-breakers themselves are monotonically increasing over insertion time. Specifically, given two distinct keys κ_r and κ_{r+1} where $\kappa_r < \kappa_{r+1}$, if any instance of κ_{r+1} is inserted before the last instance of κ_r, then the largest (right-most) tie-breaker of the κ_r block must be greater than the lowest (left-most) tie-breaker of the κ_{r+1} block. This guarantees a descent at the exact boundary between the κ_r block and the κ_{r+1} block.

Theorem 3.2 (Exact recovery). *Let $\kappa_1 < \cdots < \kappa_m$ be the plaintext keys and let $t_{\min}(\kappa_r)$ and $t_{\max}(\kappa_r)$ denote the first and last insertion times of κ_r. If tie-breakers are monotonically increasing, then $\vec{\mathbf{f}} = \vec{\mathbf{f}}'$ if and only if*

$$t_{\min}(\kappa_{r+1}) < t_{\max}(\kappa_r) \qquad \text{for all } r \in \{1, \ldots, m-1\}.$$

The proof of Theorem 3.2 is in Appendix C, where we also discuss how to *optionally* use injections to guarantee this condition.

3.1 Case Studies

We now concretely instantiate the aforementioned attack in three TEE-hybrid systems: Microsoft SQL Server's Always Encrypted [6], StealthDB [141], and HeDB [101], each of which leaks plaintext tie-breakers in index B$^+$-tree snapshots.

Microsoft SQL Server's Always Encrypted (AE). Our first case study is Always Encrypted (AE)'s TEE-hybrid *randomized indexing* scheme (rnd). AE inherits the storage engine from Microsoft SQL Server, which appends each record's unique ID to each (encrypted) search key in the index B$^+$-tree as a tie-breaker. The record ID is unique and monotonically increasing (Observation 1). As illustrated in Figure 2, this gives us the necessary ingredients for our attack. Concretely,

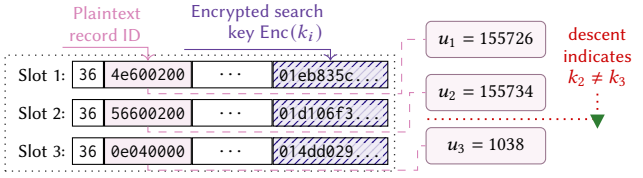

Figure 2: Parsing an Always Encrypted [6] B$^+$-tree snapshot. Record ID tie-breakers u_i are in plaintext (Observation 3) and are sorted lexicographically by (k_i, u_i) (Observation 2).

an attacker who compromises an untrusted database administrator account can use SQL Server's Database Console Command API [36] to memory dump the encrypted index B$^+$-tree. It then parses the structure to retrieve the ordered ciphertexts and their corresponding tie-breakers.

StealthDB. StealthDB [141] is built on top of PostgreSQL. Like SQL Server, PostgreSQL stores unique and monotonically increasing record IDs next to (encrypted) index B$^+$-tree search keys to break ties. As before, the use of plaintext tie-breakers is sufficient for our attack. We use an untrusted database administrator account to capture a memory dump of the B$^+$-tree using PostgreSQL's `pageinspect` API [119] and parse the structure similarly.

As a remark, even if record IDs were not stored in plaintext in the index B$^+$-tree, the attacker can still recover them indirectly as StealthDB stores identical search key ciphertexts in both the index B$^+$-tree and the record B$^+$-tree. The attacker can dump both trees and link the ordered ciphertext in the index tree to its record ID by matching it to corresponding ciphertexts in the record tree. Thus, StealthDB still remains vulnerable even if the index B$^+$-tree did not co-locate plaintext record IDs with encrypted search keys.

HeDB. Like StealthDB, HeDB [101] is based on PostgreSQL, and so it inherits the plaintext record ID tie-breaker behavior for (encrypted) search trees. We operationalize the attacker identically as in StealthDB using the `pageinspect` API [119].

Interestingly, even if record IDs were not revealed directly, HeDB leaks an indirect "proxy" for the tie-breaker via its selection of IVs for search key AES-GCM ciphertexts. As illustrated in Figure 3, HeDB maintains a counter for the number of previously encrypted AES blocks. When a new ciphertext is generated, it sets the ciphertext's IV to the total number of previously encrypted AES blocks. That is, if $B(k_i)$ denotes the number of blocks used to encrypt search key k_i, then the IV iv_i for the ith inserted search key k_i is

$$\mathsf{iv}_i = \sum_{t<i} B(k_t).$$

As each ciphertext has at least one block, the iv_i's are strictly increasing in insertion order. The record IDs used as the actual tie-breaker are also strictly increasing (Observation 1). Thus, lexicographically ordering by the IVs produces the same sorted key sequence as lexicographically ordering by the record IDs, and the sequence of IVs has descents at exactly the positions where the hidden record ID sequence has descents. Applying our attack directly to the ciphertext IVs thus produces the same frequency approximation.

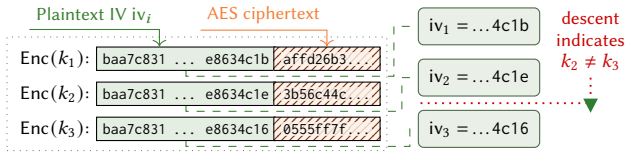

Figure 3: Parsing a HeDB [101] B⁺-tree snapshot. IVs are strictly increasing in insertion time, allowing them to be used as a proxy for the tie-breaker.

Other systems. In Appendix D, we discuss Alibaba's ApsaraDB Always-Confidential [143], GaussDB's Full Encryption [156], Enclage [132], Cipherbase [7], and TrustedDB [12]—all of whose published architectures suggest they are similarly vulnerable when explicitly configured with attributes with TEE-based comparisons enabled. For these systems, we were unable to access public artifacts on which we could verify our attacks[2]. Nevertheless, each of their public descriptions satisfies the structural observations above, so we believe these schemes are also vulnerable.

3.2 Brief Evaluation and Validation

We implemented our frequency reconstruction attack against Microsoft SQL Server 2025 [6], StealthDB [141], and HeDB [101]. We will show more detailed end-to-end experiments in Section 4, so we just provide a brief validation of the frequency reconstruction portion of our attack here. Since we are able to instantiate the tie-breaker attack identically across all EDBs, the attack performance is equivalent regardless of the underlying EDB. Thus, we present a single set of results that show the attack across all EDBs.

Workload. We generate synthetic Zipf distributions with skew s with a given domain size $|D|$, and n records. We then insert the records into each database according to one of the following three schedules: Sorted inserts records in plaintext order. Sorted-Jitter keeps the workload roughly sorted by plaintext value, but assigns each record a local jitter proportional to its rank $r + U[0, 2)$, so adjacent values overlap while distant values remain mostly separated. Random uniformly permutes the data before insertion.

Metrics. We run 1,000 trials and report **% Exact Freq**, the percentage of trials where the adversary recovers the *exact frequency vector* in Table 1 and Figure 4. (Trials where the partial vector is recovered are treated as failures.) In Table 1 we also report the average *boundary recall* (**Avg. BR**), the percent of true adjacent key boundaries that appear as descents in the recovered tie-breaker sequence.

Interpretation. In Table 1, we see that when inserts are sorted by plaintext value, the final tie-breaker vector is monotone and so the attacker sees no descents. Thus, exact recovery is always 0%, exactly as Theorem 3.2 predicts. With schedules where records are even just slightly interleaved with each other (Sorted-Jitter), the exact recovery condition formalized in Theorem 3.2 is more likely to hold and so the adversary can recover frequencies provided that the number of insertions is sufficiently large.

[2]Alibaba independently verified that our attack works against Always-Confidential indexes that have TEE-based comparison operations enabled on them. Alibaba states that they consider "the resulting leakage an inherent consequence of the authorized configuration and its relaxed security guarantees".

Table 1: Accuracy of the single-snapshot attack on Zipf data with $|D| = 64$, n records, and skew s. Takeaway: Random insert order exposes enough boundaries for near-perfect recovery once values repeat, but sorted inserts reveal no boundaries.

Workload		Avg. BR (%) ↑		Exact Freq. (%) ↑	
Insert Schedule	n	$s = 0$	$s = 1.25$	$s = 0$	$s = 1.25$
Sorted	1024	0.0	0.0	0.0	0.0
	4096	0.0	0.0	0.0	0.0
Sorted-Jitter	1024	99.7	66.4	82.2	0.0
	4096	100.0	97.6	100.0	19.5
Random	1024	100.0	92.6	100.0	0.3
	4096	100.0	99.9	100.0	96.5

Figure 4 shows that when the domain is large but the number of inserted records is small, the adversary is sometimes not able to recover complete frequencies. This is because the support size of the rarest elements becomes too small for the exact recovery condition in Theorem 3.2 to hold. In other words, the fewer instances of a given key, the less likely it is that a descent will appear in the tie-breaker sequence at that key's boundary.

3.3 Possible Mitigations

We first note two system-specific fixes for the more subtle leakage channels above. StealthDB's ciphertext linkability issue can be mitigated by not reusing the same ciphertexts in both the index tree and the record tree. HeDB's counter leakage can be mitigated by randomizing the IVs with each ciphertext. Then, across all schemes, the primary vulnerability is that each scheme's encrypted index B⁺-tree stores the associated record ID tie-breakers in plaintext.

A simple mitigation for the single-snapshot attack is to additionally encrypt the tie-breaker with the attribute key and perform comparisons inside the enclave on the entire search key and tie-breaker tuple.[3] Then, a single snapshot of the index B⁺-tree no longer exposes the tie-breaker required by our attack.[4]

It is tempting to view this as a complete fix. While this mitigates the *single-snapshot* attack, in the next section, we will show how a *multi-snapshot* adversary can still perform the same attack even when tie-breakers are not directly exposed.

4 Dendrochronology with Multiple Snapshots

We now show that the dendrochronology attack generalizes to a hypothetical TEE-hybrid indexing scheme that does not directly expose plaintext tie-breakers (e.g., using the straw man mitigation of Section 3.3). We will show that a *multi-snapshot* adversary can use multiple snapshots of the index B⁺-tree to synthesize a "proxy" tie-breaker whose descents reveal key block boundaries. By linking the same encrypted search key across snapshots, we can label each record by its *birth time*, or the timestamp of the first snapshot in

[3]A variation of this is done in Enclage [132], which we discuss in Appendix D.

[4]Under this setup, the record ID still must be exposed to the untrusted DBMS engine during query execution since the DBMS needs the plaintext ID to retrieve full record data from the record B⁺-tree. However, transient exposure during queries is outside the snapshot adversary's view.

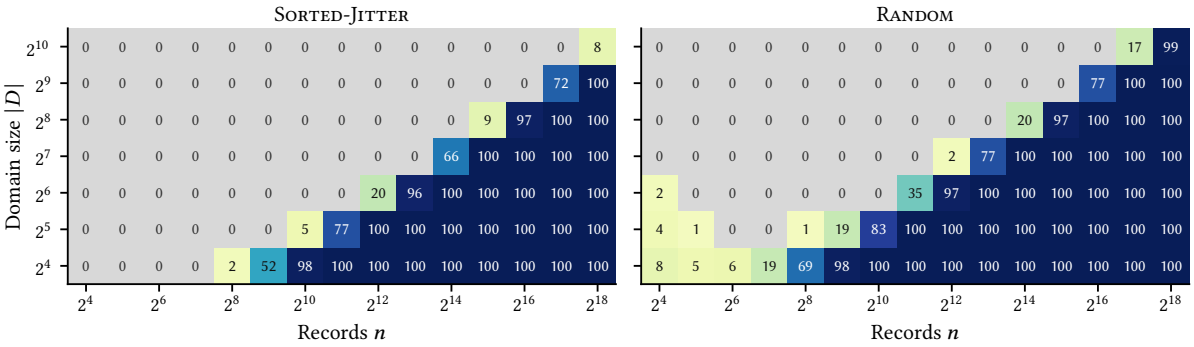

Figure 4: % Exact Freq. based on workload size n and domain size $|D|$ for Zipf skew $s = 1.25$. Takeaway: Frequency recovery is best when $n > |D|$, as increasing n increases the count of each unique value, which increases the probability that Theorem 3.2's condition holds. When $n \ll |D|$, the lower values of n make fewer tail values in the Zipf distribution appear. This creates fewer distinct plaintext values and, therefore, fewer values that need to be recovered, which occasionally increases recovery.

which it appears. As we will show, this birth time has the same descent structure as the record IDs.

Notation and threat model. We now consider a stronger *q-snapshot internal-memory adversary* that obtains a sequence of point-in-time snapshots of portions of the DBMS state—specifically, the in-memory encrypted index B$^+$-tree. Formally, we say that after the database has processed a workload of n insertions $\langle \mathrm{op}_1, \ldots, \mathrm{op}_n \rangle$, the adversary obtains $(T_{t_1}, T_{t_2}, \ldots, T_{t_q})$, the sequence of index B$^+$-tree snapshots at timestamps $t_1 < \cdots < t_q$:

$$\mathsf{View}_{\Pi}^{q,\mathsf{int}}(\langle \mathrm{op}_1, \ldots, \mathrm{op}_n \rangle) := (T_{t_1}, T_{t_2}, \ldots, T_{t_q}).$$

The adversary can parse each snapshot T_{t_i} to obtain its sorted encrypted key sequence, but cannot see plaintext tie-breakers:

$$\begin{aligned}\mathsf{Parse}(T_{t_q}) &\to \langle \mathsf{Enc}(k_1), \ldots, \mathsf{Enc}(k_n) \rangle \\ &= \langle \mathrm{ct}_1, \ldots, \mathrm{ct}_n \rangle.\end{aligned}$$

Just as in Section 3, the attacker does not issue database operations, does not observe code execution during operations, and does not have access to the attribute keys. For simplicity, we focus on extracting frequency information about the final snapshot T_{t_q}. Thus, we will assume $t_q = n$ (so the final snapshot is T_n).

Our insight. Our attack will leverage the monotonicity of tie-breakers (Observation 1) and a new observation: in AE, StealthDB, and HeDB, search key ciphertexts in the encrypted index B$^+$-tree are not rewritten between snapshots unless an operation explicitly updates the associated record. This allows for search key *linkability* between snapshots—the adversary can tell when the same (encrypted) search key instance appears in two different snapshots.

> *Observation 5.* Search key ciphertexts in the TEE-hybrid encrypted index B$^+$-tree are not rewritten between insertions unless an operation explicitly updates the associated record.

We will now describe how the adversary can exploit linkability to reconstruct a "proxy" for the tie-breakers. To provide intuition, we will start with the simplest case where the adversary receives a snapshot after all insertions ($q = n$). Then, we will generalize this to the case where the adversary receives fewer snapshots ($q < n$).

Warmup: $q = n$. Consider the n-snapshot adversary who obtains the sequence of index B$^+$-tree snapshots after every operation,

$$(T_1, T_2, \ldots, T_n).$$

Given the sorted encrypted key sequence in the final snapshot T_n,

$$\mathsf{Parse}(T_n) \to \langle \mathrm{ct}_1, \ldots, \mathrm{ct}_n \rangle,$$

for each encrypted search key ct_i, we compute each record's *birth time* b_i, the timestamp at which ct_i first appeared in the snapshots:

$$b_i := \min \{ t_j : \mathrm{ct}_i \in \mathsf{Parse}(T_{t_j}) \}$$

This produces a *birth time sequence*:

$$\langle b_1, \ldots, b_n \rangle.$$

This birth time sequence serves as a *proxy* for the hidden record ID tie-breaker. The reasoning is similar to why HeDB's IV counters serve as a proxy—by definition, the birth time b_i of each ct_i is strictly increasing in insertion order, and the record IDs used as the actual tie-breaker are strictly increasing in insertion order (Observation 1). Thus, the birth time sequence must have the same descents at exactly the positions where the hidden record ID sequence has descents. Applying our attack directly to the birth time sequence thus produces the same frequency approximation.

Generalizing to fewer snapshots: $q < n$. We now consider the weaker q-snapshot adversary that receives only a subset of index B$^+$-tree snapshots at timestamps $t_1 < \cdots < t_q$. In this setting, multiple encrypted search key insertions could have happened between snapshots, which means that multiple encrypted search keys have the same birth time. Nevertheless, we can still extract (some, if not all) frequency information with essentially the same approach.

As before, we start by computing the birth time sequence of all encrypted search keys ct_i that appear in the final snapshot T_n:

$$\langle b_1, \ldots, b_n \rangle.$$

Then, consider a pair of encrypted search keys ct_i and ct_{i+1} and their corresponding birth times b_i and b_{i+1}. While the birth times are no longer unique, if

$$b_i > b_{i+1},$$

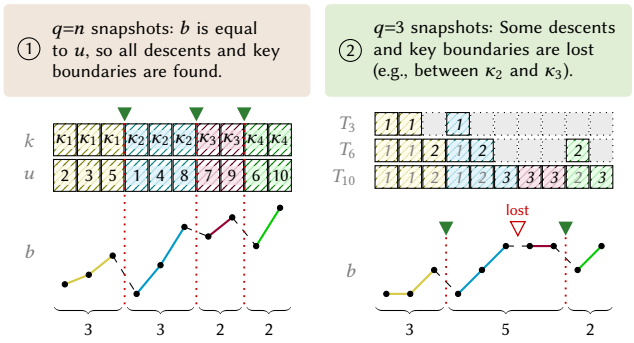

Figure 5: Example where fewer snapshots lose descents. This decreases the accuracy of frequency recovery, though the remaining descents may still reveal meaningful boundaries.

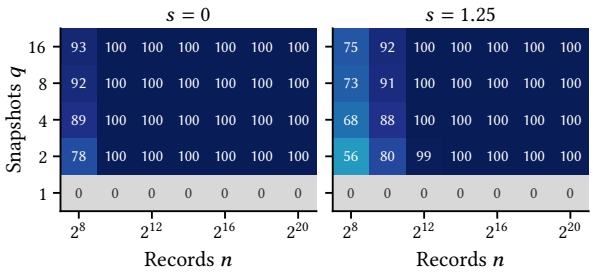

Figure 6: Average BR@q over 1,000 synthetic Zipf workloads with skew s, domain size $|D| = 64$, with varying numbers of q uniformly spaced snapshots and n insertions. Takeaway: With only $q = 2$, boundary recovery jumps sharply. With only one snapshot, the adversary cannot recover frequencies.

then ct_i must have been inserted later than ct_{i+1}, and by Observation 1, the true hidden record IDs u_i and u_{i+1} must follow $u_i > u_{i+1}$. This shows us that birth time descents still directly indicate hidden tie-breaker descents, just as required by our attack.

Intuitively, the reduction in snapshots occasionally reduces the accuracy of frequency recovery. Generalizing Theorem 3.2 to this setting, a descent in the hidden tie-breaker sequence is unrecoverable if both records at the descent boundary have the same birth time (e.g., because they first appear in the same snapshot). Figure 5 shows an example of this. Nevertheless, we will show in our experiments on real-world datasets that this is often not a problem—in fact, we will show we can often recover meaningful frequency information even at just $q = 2$ snapshots.

4.1 Evaluation

We continue the evaluation from Section 3.2 but now in the multi-snapshot setting. We aim to answer two questions: First, how many snapshots are needed before birth times recover the hidden equality-block boundaries? Second, once those boundaries are recovered, how much do they improve inference?

To do this, we implemented the attacks across AE, StealthDB, and HeDB. We loaded each workload into the target EDB and captured the stated number of snapshots using the attacker instantiations for each EDB from Section 3.1. Then, we parse the extracted index B$^+$-tree state, link encrypted search keys across snapshots, and compute the birth time of each search key. Since the attack works identically in all EDBs, we present one set of unified results.

Snapshot sensitivity. Figure 6 measures the fraction of true key boundaries recovered from birth time descents. We generate 1,000 Zipf workloads (sampled i.i.d.) with domain size $|D| = 64$, vary the number of inserted records n, and give the adversary q uniformly spaced snapshots. We report **BR@q**, the percent of true plaintext key boundaries recovered at q snapshots (Section 3.2).

The results show that the attack is effective with surprisingly few snapshots. With only $q = 2$ snapshots, boundary recovery jumps sharply. For uniform workloads it reaches at least 78% once $n \geq 2^8$, and reaches 100% for $n \geq 2^{10}$. For skewed Zipf workloads with $s = 1.25$, the same trend holds, although recovery is slightly harder for small n because rare values have small equality blocks and are more likely to only appear within the same snapshot.

End-to-end inference. To demonstrate the impact of frequency recovery, we show end-to-end inference experiments which recover the ordered frequency vector from the TEE-hybrid snapshot using our dendrochronology attack, and then attempt to map the resulting ciphertext blocks to plaintext values using the state-of-the-art OPE *multinomial* inference attack of Bindschaedler, Grubbs, Cash, Ristenpart, and Shmatikov [16].

Table 2 reports our results. **BR@2** reports boundary recall using only $q = 2$ snapshots, where the first snapshot is captured halfway through the transcript. **% ERecov** is the percentage of inserted records whose plaintext value is correctly guessed. The baseline Ord uses only the final order and the auxiliary distribution. Our attack, Den, leverages multinomial inference with the recovered ordered composition in addition to the final order. We stress both Ord and Den use the same auxiliary source in each row *a priori*.

We evaluate our attack on real-world datasets including names from the US Social Security [139] and US Census [136, 137], US ZIP codes [76], Amazon product reviews [74], and birthdays [50]. Each real dataset defines a multiset of values to insert. We pick a uniformly random permutation of that multiset as the insertion order. Then, we take $q = 2$ snapshots and proceed with our attack.

Across these datasets, two snapshots are already enough to recover almost all frequency boundaries: BR@2 is 98.7% for SSA names and 100% for every other real dataset. This translates into large inference gains. For example, on U.S. Census surnames [136], order-only inference achieves only 17.5% accuracy, whereas Den's recovered frequency raises accuracy to 99.5%; on ZIP code data [76], the Ord baseline is almost uninformative at 0.1%, while Den reaches 97.6%. These results validate broader findings from prior inference attacks [16, 44, 46, 114, 126] showing that order-only leakage requires the auxiliary information to be very accurate to achieve good value reconstruction.

To illustrate the effects of sparse domains (where $|D|$ is similar to or exceeds n), we demonstrate the attack on synthetic uniform and Zipf workloads (sampled i.i.d.) and use the same distribution from which the data was generated as auxiliary information. This intuitively lowers BR@2 (because the conditions of Theorem 3.2

Table 2: End-to-end inference from $q = 2$ snapshots. Ord and Den use the same auxiliary. Takeaway: Our dendrochronology attack (Den) substantially improves end-to-end inference in most cases over the order-only baseline. On the real datasets, where $n \gg |D|$, BR@2 is near 100% and reconstruction is accurate. On the synthetic datasets, where $|D|$ is comparable to or exceeds n, BR@2 is lower but still improves reconstruction when the distribution is skewed.

Dataset				Descents	ERecov (%) ↑		
Workload	Auxiliary	$\|D\|$	n ($\times 10^6$)	BR@2 (%) ↑	Ord → Den		Gain (pp)
Real datasets							
SSA first names (2020) [139]	SSA first names (2019) [139]	34,489	3.3	98.7	31.4 →	**98.7**	+67.3
Census surnames (2010) [136]	Census surnames (2000) [137]	5,206	178.3	100.0	17.5 →	**99.5**	+82.0
IRS ZIP codes (2022) [76]	Census ZIP codes (2020) [23]	33,778	156.2	100.0	0.1 →	**97.6**	+97.5
Amazon Industrial [74]	Hu et al., Table 2 [75]	5	0.4	100.0	94.4 →	**100.0**	+5.6
Amazon Software [74]	Hu et al., Table 2 [75]	5	1.3	100.0	66.2 →	**100.0**	+33.8
SSA birthdays (2014) [50]	Murphy histogram [113]	365	4.0	100.0	39.1 →	**100.0**	+60.9
SSA birthdays (2013) [50]	Murphy histogram [113]	365	4.0	100.0	43.8 →	**100.0**	+56.2
SSA birthdays (2012) [50]	Murphy histogram [113]	365	4.0	100.0	36.7 →	**100.0**	+63.3
Synthetic datasets							
Zipf ($s = 1$, $\|D\| = 2^{16}$)	Zipf ($s = 1$, $\|D\| = 2^{16}$)	2^{16}	1.0	69.6	44.7 →	**79.0**	+34.3
Zipf ($s = 1$, $\|D\| = 2^{18}$)	Zipf ($s = 1$, $\|D\| = 2^{18}$)	2^{18}	1.0	47.5	41.7 →	**70.1**	+28.4
Zipf ($s = 1$, $\|D\| = 2^{20}$)	Zipf ($s = 1$, $\|D\| = 2^{20}$)	2^{20}	1.0	38.6	35.8 →	**62.9**	+27.1
Zipf ($s = 1$, $\|D\| = 2^{22}$)	Zipf ($s = 1$, $\|D\| = 2^{22}$)	2^{22}	1.0	34.4	31.5 →	**56.9**	+25.4
Uniform ($\|D\| = 2^{16}$)	Uniform over $[0, 65535]$	2^{16}	1.0	99.9	2.1 →	**7.0**	+4.9
Uniform ($\|D\| = 2^{18}$)	Uniform over $[0, 262143]$	2^{18}	1.0	75.8	**0.4** →	**0.4**	+0.0
Uniform ($\|D\| = 2^{20}$)	Uniform over $[0, 1048575]$	2^{20}	1.0	38.1	**0.1** →	**0.1**	+0.0
Uniform ($\|D\| = 2^{22}$)	Uniform over $[0, 4194303]$	2^{22}	1.0	28.1	**0.0** →	**0.0**	+0.0

are less likely to occur), but when the distribution is skewed we still achieve improved reconstruction.

4.2 Possible Mitigations

Our multi-snapshot attack exploits both (1) the fact that tie-breakers are correlated with insertion order and (2) ciphertext linkability. Removing either property prevents our attack, though, as we will discuss, there are some limitations to each approach.

Randomized tie-breakers. As discussed in Section 2.1, there are many practical benefits to strictly increasing tie-breakers in real-world storage engines. Nevertheless, in theory, there is no strict limitation that prevents us from changing the tie-breaker behavior. Thus, a natural idea is to completely decouple tie-breakers from insertion order by randomizing tie-breakers. As it turns out, we can show that, if the insertion workload is sampled i.i.d., then randomized tie-breakers reduce multi-snapshot leakage to order-only leakage.

We now formalize this. For all plaintext insertions k_i, we assume each k_i is drawn i.i.d. from a fixed distribution $\mathcal{D}$ over the key space. For each insertion, we independently sample a random tie-breaker

$$u_i \xleftarrow{\$} \{0,1\}^\lambda$$

for statistical security parameter λ. Then, the index will sort and order records lexicographically by (k_i, u_i). We will show that the insertion order is statistically indistinguishable from a randomly chosen insertion order provided that all of the sampled tie-breakers have no collisions. This shows that, as long as the u_i values remain hidden from the adversary, the insertion order reveals nothing about the inserted keys beyond order-only leakage.

Theorem 4.1 (Statistical security of random tie-breakers). *Let $\mathcal{D}$ be a fixed distribution over the key space. Suppose, at time $i \in \{1, \ldots, n\}$, we insert $k_i \xleftarrow{\text{i.i.d.}} \mathcal{D}$ with tie-breaker $u_i \xleftarrow{\$} \{0,1\}^\lambda$. Let $\vec{\mathbf{f}}$ be the frequency vector of $k_1, \ldots, k_n$ and $\mathbf{b} = \langle b_1, \ldots, b_n \rangle$ where b_j is the insertion time of the entry at sorted position j. Then, let π be a uniformly random permutation of $\{1, \ldots, n\}$ sampled independently. Then, for any possibly unbounded adversary $\mathcal{A}$ and for $n = \text{poly}(\lambda)$,*

$$\left| \Pr\left[\mathcal{A}(\vec{\mathbf{f}}, \mathbf{b}) = 1\right] - \Pr\left[\mathcal{A}(\vec{\mathbf{f}}, \pi) = 1\right] \right| \leq \text{negl}(\lambda),$$

where $\text{negl}(\lambda)$ is a negligible function of security parameter λ.

Proof. Let E be the event that all u_i values are distinct. By a union bound, the probability of $\neg E$ (2 of n tie-breakers collide) is

$$\Pr[\neg E] \leq \binom{n}{2} 2^{-\lambda} = \text{negl}(\lambda).$$

Condition on E and consider the set $M = \{(k_i, u_i)\}_{i=1}^n$, whose elements are distinct under E. Since the pairs (k_i, u_i) are i.i.d., every assignment of these pairs to the insertion times $\{1, \ldots, n\}$ is equally likely. Thus, their insertion times form a uniformly random permutation. Also, M determines $\vec{\mathbf{f}}$. Thus, $\mathbf{b}$ and $\vec{\mathbf{f}}$ are independent conditioned on E. The real and ideal distributions can differ only when $\neg E$ occurs, which gives the stated bound. □

There are some subtle considerations with this approach. First, if the adversary can observe the tie-breakers themselves (e.g., via a side channel), then it can still recover frequency information as in the single-snapshot attack (Section 3). Second, if the distribution of insertions is not i.i.d., then the adversary can possibly still exploit

correlations between insertion order and plaintext values to recover frequency information. For example, given a key space of m unique keys, inserting all m keys in sorted order will trivially produce a distinct ciphertext insertion order compared to a workload that inserts all m keys in reverse sorted order.

Nevertheless, the TEE-hybrid construction (with randomized tie-breakers) improves upon *frequency-hiding order-preserving encryption* (FH-OPE) schemes [85], which have non-uniform insertion distributions even under i.i.d. workloads [24, 67, 150]. However, the i.i.d. assumption—across the entire lifespan of the database—is very limiting (if not prohibitive) in practice, and it is unclear if real-world workloads can satisfy this assumption.

Breaking linkability. Our attacks exploit the fact that the location of an (encrypted) search key instance can be traced between snapshots. Adding techniques like *oblivious RAM* (ORAM) [60] will break the linkability property but at the cost of additional overhead on B$^+$-tree accesses and writes. Weaker alternatives which also would solve the multi-snapshot attack are more efficient *write-only ORAM* variants (e.g., [10, 17, 25]) which only protect write leakage, or *distribution-aware* techniques which only provide guarantees for a fixed, public write distribution (e.g., [58, 63, 105]). However, the use of ORAM seems fundamentally in tension with the goal of storing records in a sorted order (as is done in a B$^+$-tree index) and it is unclear what the performance of such a scheme would be. Thus, such an approach would require design and engineering of a new "ORAM-enabled" storage engine, which would defeat the premise of reusing database components.

Disable querying on randomized encryption. For completeness, a trivial mitigation is to disable TEE-based comparison operations altogether and prevent querying on randomly encrypted data. (This is the recommended approach from Microsoft and Alibaba for customers who are concerned with frequency recovery.) This stops the attack but eliminates the desired functionality.

5 Noisy Extraction from TEE-Native EDBs

So far, we have focused on TEE-hybrid systems, where we extract frequencies at the level of individual encrypted search keys. We now turn to *TEE-native* encrypted databases where an entire B$^+$-tree-based DBMS executes inside the enclave. Prior work by Espiritu, Kamara, Moataz, and Ogier [45] proposed the Leafblower attack which recovers approximate *order-only* leakage in this setting. In this section, we will show that we can augment the Leafblower attack with our dendrochronology insight to recover approximate frequency leakage at the level of *buckets* (e.g., B$^+$-tree leaf nodes).

5.1 Additional Background

TEE-native databases. We refer the reader to [45] for a detailed discussion of a TEE-native database architecture and instead provide a minimal description here of the necessary components to understand our attack. For ease of discussion and baseline comparison, we will focus on one specific TEE-native database, SQLite running inside an Intel SGX enclave, as is done in Leafblower [45]. Here, the entire DBMS executes inside the enclave. This substantially reduces the attack surface of the system, as the adversary cannot observe any internal state of the DBMS or the enclave.

To maintain persistence and resiliency guarantees, the database writes data to disk after each operation. SQLite writes its B$^+$-trees in page-aligned files where each node of the tree fits in a page. These files are encrypted using Intel SGX's Protected File System [135] format, which encrypts the file such that, when a logical plaintext page is updated between two snapshots, we can determine which logical page (and thus, corresponding B$^+$-tree node) was updated via comparing snapshots of the encrypted index file [45].

Notation and threat model. We consider the passive *n-snapshot external-memory adversary* from [45] that obtains a sequence of point-in-time snapshots of the encrypted database files on disk. Formally, we say that after the database executes a sequence of n insertions $\langle \mathsf{op}_1, \ldots, \mathsf{op}_n \rangle$, the adversary obtains $(S_1, S_2, \ldots, S_n)$, the sequence of encrypted B$^+$-tree index files at timestamps $1, \ldots, n$:

$$\mathsf{View}_{\Pi}^{n,\mathrm{ext}}\big(\langle \mathsf{op}_1, \ldots, \mathsf{op}_n \rangle\big) := \big(S_1, S_2, \ldots, S_n\big).$$

We assume that the adversary does not have access to the file encryption key used to encrypt the index files. The adversary also does not have access to enclave internals.

The Leafblower attack. The Leafblower attack [45] recovers approximate order-only leakage of records from TEE-native databases under the n-snapshot external-memory adversary. The attack reconstructs the logical B$^+$-tree structure of the database and guesses which records are contained in each leaf node. However, the attack does not recover frequencies, as it does not learn the order of records within each leaf node.

For our purposes, the Leafblower attack enables the adversary to perform the following reconstruction tasks.[5] First, for each snapshot S_i, the adversary can determine which *physical* pages of the index file snapshot correspond to *logical* leaf nodes of T_i, the B$^+$-tree after insertion i. Then, for each S_i, it can determine the order of the logical leaf nodes in the B$^+$-tree T_i. Finally, the adversary can determine which logical leaf nodes in T_j correspond to which logical leaf nodes in T_i. We will denote this as

$$\textsc{Leafblower}(S_i) \to T_i$$

where T_i is the reconstructed logical B$^+$-tree after insertion i. Using this functionality, Leafblower recovers approximate order-only leakage, but does not recover frequencies.

5.2 The Attack

If the underlying database uses monotonically increasing record IDs as a tie-breaker, dendrochronology attacks are still possible in the TEE-native setting using a technique similar to the one we used in Section 4. Even though we cannot see individual records, the adversary can keep track of the time at which each logical leaf node was created. As we will show, this is sufficient to recover approximate frequency leakage.

Intuition: nodes as "buckets". One can view each leaf node of the B$^+$-tree as a *bucket* that contains multiple records. As inserts arrive, a bucket is filled until it overflows, at which point the storage engine allocates a new bucket on the right of the overflowing bucket and redistributes records across the two buckets. Repeating this process over time creates a sequence of buckets whose creation times are

[5] We direct the reader to [45] for a more detailed discussion of the Leafblower attack.

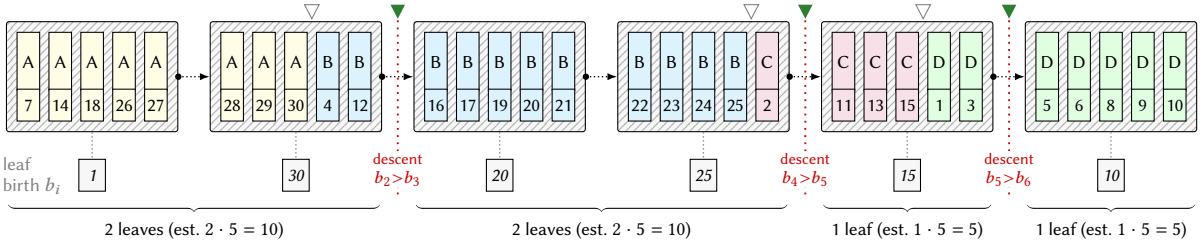

Figure 7: Example of extracting frequencies from a TEE-native database (where entire leaf pages are encrypted) using leaf birth times. Leaf contents are shown for exposition and are not visible to the adversary. Leaves split and rebalance with their neighbors when full (see [45] for a full description). Each key boundary (hollow triangle) falls inside a leaf that is split due to later inserts of the lower of the two keys, producing a leaf birth time descent to the right of the key boundary.

correlated with the insertion times of the individual records that populate them. We illustrate this in Figure 7.

The attack. Using this insight, we can obtain an *approximate* frequency vector in the following way. Let $T_n \leftarrow \textsc{Leafblower}(S_n)$ be the final index B$^+$-tree and denote its leaf nodes in left-to-right order as $B_1, \ldots, B_\ell$ where ℓ is the number of leaf nodes in T_n. For each B_i, we calculate the *leaf birth time* b_i, the timestamp of the first snapshot in which that leaf node appears:

$$b_i := \min\left\{r \in \{1, \ldots, n\} : B_i \in T_r\right\}.$$

This produces a *leaf birth time sequence*

$$\langle b_1, \ldots, b_\ell \rangle.$$

Our attack then follows as before, just over the leaf birth time sequence. We calculate descents in the leaf birth time sequence:

$$\text{descents} := \left\{\, j \in \{1, \ldots, \ell - 1\} : b_j > b_{j+1} \,\right\}.$$

We then use these descents to partition the final leaf order into segments. Writing the descents in order as $d_1 < \cdots < d_t$ and setting $d_0 = 0$ and $d_{t+1} = \ell$, we calculate the number of leaf nodes between each descent as

$$\vec{\mathbf{f}}' = \left(d_i - d_{i-1}\right)_{i \in \{1, \ldots, t+1\}}.$$

Then, given an estimate c of the number of records per leaf, we can multiply each entry of $\vec{\mathbf{f}}'$ by c to obtain an approximate frequency vector. (Equivalently, dividing each entry of $\vec{\mathbf{f}}'$ by ℓ gives an approximate normalized histogram over the plaintext keys.)

Remark on approximation. Unlike the TEE-hybrid attacks, we do not recover a true coarsening of the real frequency vector because we do not know the exact number of records in each leaf node. Instead, the adversary learns an approximation based on its knowledge of the approximate number of records in each leaf node. Nevertheless, we will show in our experimental results that this is sufficient to improve the accuracy of the adversary's auxiliary information compared to an order-only attack.

5.3 Evaluation

We extended the original attack implementation from Leafblower [45] to reconstruct an approximate frequency vector by computing the leaf birth time sequence. As in [45], we evaluated our attack against SQLite 3.53.0 [55] running under Gramine 1.9 [135].

Workloads. For comparison, we reuse workloads from Leafblower. Amazon consists of 5-star product reviews [74] with the rating histogram reported by Hu, Zhang, and Pavlou in Table 2 of [75] as auxiliary. PUDF contains approximately 700k records from the 2018 Texas Hospital Inpatient Discharges dataset [134] with corresponding histograms from the Texas Census QuickFacts [138]. For one column (stay length) where QuickFacts data is not available, we use the New York SPARCS healthcare [115] dataset as auxiliary.

We record KS_{orig}, the Kolmogorov–Smirnov distance [110] between the target histogram and the raw auxiliary prior, and KS_{adj}, which recomputes the same distance after matching the auxiliary distribution to the recovered frequency vector using Bindschaedler, Grubbs, Cash, Ristenpart, and Shmatikov's multinomial attack [16]. Lower KS values mean that the corresponding distribution more accurately aligns with the real distribution. We also measure end-to-end plaintext reconstruction with the same **% ERecov** metric from Section 4.1. Ord is the Leafblower order-only baseline; Den is with our adjusted histogram.

Interpretation. Table 3 shows two distinct effects. KS_{adj} is lower than KS_{orig} on every workload, meaning our attack allows the adversary to adjust its auxiliary distribution so it more accurately matches the real distribution. In other words, our recovered histogram is *more accurate* than the original auxiliary distribution.

However, even with more accurate auxiliary information, the downstream effects on inference are mixed. Using the adjusted auxiliary improves recovery on Amazon Software, PUDF stay length, and PUDF sex by 16.0–30.6 percentage points. However, in other workloads, the more accurate auxiliary slightly decreases recovery by up to 3.9 percentage points. This is because Leafblower recovers *approximate* order information, and so even with improved auxiliary the exact value recovery is not necessarily improved. The KS_{adj} results show this is a weakness of the Leafblower attack rather than an issue with our frequency reconstruction, and we leave it as an open problem as to whether the Leafblower attack itself can be improved to exploit more accurate auxiliary for improved recovery.

5.4 Possible Mitigations

Our attack relies on Leafblower [45] as a precondition, so we can thwart the attack by preventing Leafblower's main requirement—the adversary's ability to track writes to individual filesystem pages.

Table 3: Frequency recovery on TEE-native EDBs via our improved Leafblower attack. KS is Kolmogorov–Smirnov distance. Ord and Den use the same auxiliary distribution. Takeaway: Our frequency-adjusted histograms are more accurate than the auxiliary distribution in all cases, though improved frequency fit does not always improve inference because Leafblower's order recovery is noisy.

Dataset		Aux. Fit ↓		ERecov (%) ↑		
Workload	Auxiliary	KS_{orig}	KS_{adj}	Ord → Den		Gain
Amazon Industrial	Hu et al. [75]	0.018	**0.006**	**98.1** →	95.6	−2.5
Amazon Software	Hu et al. [75]	0.147	**0.020**	61.3 →	**87.6**	+26.3
PUDF stay length	SPARCS [115]	0.026	**0.008**	63.3 →	**93.9**	+30.6
PUDF ethnicity	Census [138]	0.040	**0.024**	**99.0** →	95.1	−3.9
PUDF race	Census [138]	0.153	**0.138**	**84.8** →	83.5	−1.3
PUDF sex	Census [138]	0.086	**0.009**	82.0 →	**98.0**	+16.0

One way to do this is with *oblivious RAM* (ORAM) [60], which has been used in TEEs to hide access patterns to memory (e.g., [1]) and in filesystem syscalls [3, 34]. Some TEE-native EDBs even use ORAM directly [30, 41, 155]. However, the overhead of ORAM [93] may be prohibitive for online transactional databases. One possible tradeoff is to use *write-only ORAM* techniques (e.g., [10, 17, 25]), which achieve greater efficiency by only protecting writes, which is sufficient to mitigate snapshot-based attacks. One advantage of the TEE-native setting is that the ORAM can be applied transparently at the TEE virtualization [82] or syscall proxy [135] layer as opposed to needing to integrate it directly with the DBMS, which may make it easier to implement. We direct the reader to the Leafblower paper for further attack-specific mitigations [45].

6 Related Work

Breaking TEE-based applications without side channels. An emerging line of work attempts to identify unintended leakage from TEE-based applications without breaking the TEE primitive itself. In addition to Leafblower [45] and Lama [126], we are aware of two works in the stronger *persistent adversary* model: Jean-Louis, Li, Ji, Malvai, Yurek, Bellemare, and Miller [78] exploit access pattern leakage in filesystem syscalls used by TEE-based cryptocurrency applications. Similarly, Zhao, Hu, Liu, and Chen [154] use a persistent adversary to cause integrity violations in the "maintenance mode" of HeDB [101].

Attacks on FH-OPE. Grubbs, Sekniqi, Bindschaedler, Naveed, and Ristenpart [67] presented the first attack against the FH-OPE construction of [85]. Their attack is limited to recovering the approximate frequencies of the highest cardinality elements. Yang and Kim [150] use a persistent adversary model that observes range queries on an FH-OPE encrypted database. The closest work to ours is that of Cao, Liu, Shen, Ye, and Ren [24], who exploit non-uniform ciphertext insertion distributions in specific FH-OPE schemes to recover approximate frequencies. Their attacks require at least 6 snapshots in order to achieve greater than 90% exact recovery on real-world datasets, whereas our TEE-hybrid attacks require at most 2 snapshots to achieve close to perfect accuracy.

Attacks in encrypted search. We have already discussed attacks on TEE-based EDBs [45, 126], attacks on FH-OPE [24, 67, 150], and attacks on other kinds of property-preserving encryption (e.g., [16, 67, 114]). Most prior cryptanalysis of EDBs focuses on *persistent adversaries* which exploit leakage that is not visible in *snapshot* models. These attacks typically leverage access patterns, search patterns, response volumes, and query or record injection. In particular, persistent model attacks against encrypted range schemes (e.g., [37, 47, 49]) are relevant as recovering record access pattern leakage from all possible queries is equivalent to recovering frequency and order leakage (e.g., [46, 48, 64, 65, 68, 84, 89–92, 102, 106–109]).

Snapshot security. Snapshot-secure encrypted data structures [4, 39, 81, 86, 118] are motivated by the same reasons as our threat model, where the attacker may obtain structured snapshots of memory or storage but may not control the execution environment long enough to observe every operation. Nevertheless, some attacks on real-world EDBs demonstrate that they leak more information in the snapshot model than originally claimed due to the presence of additional information in logs or other indexing structures [66, 69].

Other database side-channel attacks. Various side channel attacks target non-cryptographic database components [33, 43, 53, 79, 83, 125, 128, 142]. Other works recover secrets by injecting records and measuring the resulting size of (encrypted) index files [19, 54, 72]. As discussed, our attacks do not require injections by default.

7 Conclusion

"PPE attacks that rely on the static DB [e.g., snapshots] *are not applicable to enclave-enabled EDBs which store data strongly-encrypted* [e.g., using randomized encryption]*"*. This statement dates back to 2021 [8], but our investigation demonstrates this is false. Our results show that such snapshot attacks are, in fact, applicable to a wide class of TEE-based EDBs—even without attempting to break the TEE primitive itself.

The burden of legacy code. It is well-known that legacy code is often the source of security vulnerabilities, but here we show an example—TEE-based EDBs—where maintaining legacy code and reusing existing DBMS machinery is a conscious "benefit" of the design. This makes EDBs easier to build, but our results show another outcome of this bargain: the same features that make legacy storage engines practical may be surprising sources of leakage. Our proposed mitigations (e.g., randomized tie-breakers) are not a simple drop-in replacement for existing DBMS logic and do not protect against all workloads—more robust mitigations may require substantial redesign of the underlying storage engine.

One philosophical question that this work raises is: why has this seemingly simple vulnerability gone unnoticed for many years? Prior analyses of TEE-based EDBs [45, 126] modeled the TEE-based index as a sorted sequence of ciphertexts and abstracted additional practical index metadata—such as tie-breakers—out of the model. This highlights an occupational hazard of designing TEE-based applications: the promise of reusing a plaintext storage engine invites designers to treat the existing engine as a black box, when, in reality, its algorithmic invariants become critical parts of the security boundary (and surprising vectors of leakage).

There may be other standard database mechanisms that leak insertion order and allow for the same dendrochronology attack, such as multi-versioning systems (e.g., [38] in SQL Server) or metadata in other index types [117]. We leave the investigation of these orthogonal systems as an open problem.

We end with a broader lesson: secure co-design with TEEs, whether for databases or other applications, cannot treat existing algorithms as a neutral implementation detail. Such components must be carefully analyzed as part of the security boundary. We expect that as additional TEE-based applications become more mainstream, vulnerabilities like these will arise from natural consequences of the underlying (plaintext) algorithms.

Acknowledgments

We thank the CCS reviewers for their helpful feedback. We thank Seny Kamara, Tarik Moataz, and Andrew Park for helpful discussions. We thank Microsoft and Alibaba for their responses in the coordinated disclosure process. We thank the reviewers of [45] whose question about frequency recovery stuck with us until we realized that it might be possible—in even more scenarios than we imagined. We thank JB, GK, SA, and HL for reminding us to think about what we're going to leave behind. As always, we thank Casey Nelson for all of their support. Generative AI tools (Claude Code) were used to write code and for the final review of paper text. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] Shaizeen Aga and Satish Narayanasamy. 2019. InvisiPage: oblivious demand paging for secure enclaves. In *ISCA*.
[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *SIGMOD*.
[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS*.
[4] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2019. Breach-Resistant Structured Encryption. *PETS* 2019, 1 (2019).
[5] Anjuna. 2023. Demo Series: Protect Database From Exposing Data in Temporary Files. https://www.anjuna.io/blog/demo-series-protect-database-from-exposing-data-in-temporary-files
[6] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *SIGMOD*.
[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *SIGMOD*.
[8] Arvind Arasu, Raghav Kaushik, Donald Kosmann, and Ravi Ramamurthy. 2021. Integrity-based Attacks for Encrypted Databases and Implications. In *CIDR*.
[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*.
[10] Adam J. Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S. Roche. 2017. ObliviSync: Practical Oblivious File Backup and Synchronization. In *NDSS*.
[11] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*.
[12] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *TKDE* 26, 3 (2014).
[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM TOCS* 33, 3 (2015).
[14] Rudolf Bayer and Edward McCreight. 1970. Organization and maintenance of large ordered indices. In *ACM SIGFIDET*.
[15] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and Efficiently Searchable Encryption. In *CRYPTO*.
[16] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. 2018. The tao of inference in privacy-protected databases. *VLDB* 11, 11 (2018).
[17] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *CCS*.
[18] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *EUROCRYPT*.
[19] Britney Bourassa, Yan Michalevsky, and Saba Eskandarian. 2025. G-DBREACH Attacks: Algorithmic Techniques for Faster and Stronger Compression Side Channels. In *ACNS*.
[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*.
[21] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *CCS*.
[22] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX SEC*.
[23] U.S. Census Bureau. [n. d.]. Total Population. U.S. Census Bureau. https://data.census.gov/table/DECENNIALDHC2020.P1?q=All+5-digit+ZIP+Code+Tabulation+Areas+within+United+States+Populations+and+People
[24] Xinle Cao, Jian Liu, Yongsheng Shen, Xiaohua Ye, and Kui Ren. 2023. Frequency-Revealing Attacks against Frequency-Hiding Order-Preserving Encryption. *VLDB* 16, 11 (2023).
[25] Anrin Chakraborti, Chen Chen, and Radu Sion. 2017. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries. *PETS* 2017, 3 (2017).
[26] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*.
[27] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel TDX Demystified: A Top-Down Approach. *ACM Comp. Surv.* 56, 9 (2024).
[28] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. 2026. TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition. In *S&P*.
[29] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX SEC*.
[30] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*.
[31] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*.
[32] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR CHES* 2018, 2 (2018).
[33] Chen Dar, Moshik Hershcovitch, and Adam Morrison. 2023. RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time. *SIGMOD* 1, 1 (2023).
[34] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *ACM SOSP*.
[35] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. 2026. Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing. In *S&P*.
[36] Kalen Delaney. 2013. Chapter 7. Indexes: internals and management. In *Microsoft SQL Server 2012 Internals: Professional* (1 ed.).
[37] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. 2018. Practical Private Range Search in Depth. *ACM TODS* 43, 1 (2018).
[38] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*.
[39] Yang Du, Daniel Genkin, and Paul Grubbs. 2022. Snapshot-Oblivious RAMs: Sub-logarithmic Efficiency for Short Transcripts. In *CRYPTO*.
[40] F. Betül Durak, Thomas M. DuBuisson, and David Cash. 2016. What Else is Revealed by Order-Revealing Encryption?. In *CCS*.
[41] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *VLDB* 13, 2 (2019).
[42] Zachary Espiritu. 2026. Snapshots of TEE-Based Encrypted Databases Leak Plaintext Histograms via Dendrochronology. In *CCS*.
[43] Zachary Espiritu and David Cash. 2026. Plaintext Recovery Against Post-Filtering Access Control Systems. In *USENIX SEC*.
[44] Zachary Espiritu, Seny Kamara, and Tarik Moataz. 2025. Bayesian Leakage Analysis: A Framework for Analyzing Leakage in Cryptography. *CIC* 2, 1 (2025).
[45] Zachary Espiritu, Seny Kamara, Tarik Moataz, and Valentin Ogier. 2026. Leafblower: a Leakage Attack Against TEE-Based Encrypted Databases. In *S&P*.

[46] Zachary Espiritu, Seny Kamara, Tarik Moataz, and Andrew Park. 2025. PolySys: an Algebraic Leakage Attack Engine. In *USENIX SEC*.

[47] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*.

[48] Francesca Falzon, Evangelia Anna Markatou, Akshima, David Cash, Adam Rivkin, Jesse Stern, and Roberto Tamassia. 2020. Full Database Reconstruction in Two Dimensions. In *CCS*.

[49] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. 2022. Range Search over Encrypted Multi-Attribute Data. *VLDB* 16, 4 (2022).

[50] FiveThirtyEight. 2016. U.S. Births. https://github.com/fivethirtyeight/data/blob/master/births/US_births_2000-2014_SSA.csv

[51] Fortanix. 2023. Fortanix Launches Industry-First Confidential Data Search for Regulated Encrypted Data. https://www.fortanix.com/company/pr/2023/06/industry-first-confidential-data-search-for-regulated-encrypted-data

[52] Harold C. Fritts. 1976. Chapter 1: Dendrochronology and Dendroclimatology. In *Tree Rings and Climate*.

[53] Ariel Futoransky, Damián Saura, and Ariel Waissbein. 2007. The ND2DB attack: database content extraction using timing attacks on the indexing algorithms. In *WOOT*.

[54] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. 2024. Injection Attacks Against End-to-End Encrypted Applications. In *S&P*.

[55] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *VLDB* 15, 12 (2022).

[56] Jessica Gentles, Mason Fields, Garrett Goodman, and Suman Bhunia. 2025. Breaking the Vault: A Case Study of the 2022 LastPass Data Breach. arXiv:2502.04287 [cs]

[57] Peter Geoghegan. 2019. nbtree: An architectural perspective. Talk presented at PGCon 2019. https://www.pgcon.org/2019/schedule/attachments/518_nbtree-arch-pgcon.pdf

[58] Marilyn George, Seny Kamara, Tarik Moataz, and Zachary Espiritu. 2025. Structured Encryption and Distribution-Aware Leakage Suppression. In *ASIACRYPT*.

[59] Philipp Giersfeld, Benedict Schlüter, and Shweta Shinde. 2026. BreakFAST: Confused Deputy Attack on Infinity Fabric to Break AMD SEV-SNP. In *S&P*.

[60] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (1996).

[61] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM TODS* 35, 3 (2010).

[62] Alex Grinman. 2023. Inside the Enclave, Part 2. https://onefootprint.com/blog/inside-the-enclave-part-2

[63] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores. In *USENIX SEC*.

[64] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *CCS*.

[65] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *S&P*.

[66] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Why Your Encrypted Database Is Not Secure. In *HotOS*.

[67] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *S&P*.

[68] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *CCS*.

[69] Zichen Gui, Kenneth G. Paterson, and Tianxin Tang. 2023. Security Analysis of MongoDB Queryable Encryption. In *USENIX SEC*.

[70] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *EuroSec*.

[71] Silvia Heubach and Toufik Mansour. 2010. Chapter 1: Introduction. In *Comb. Comp. Words*.

[72] Mathew Hogan, Yan Michalevsky, and Saba Eskandarian. 2023. DBREACH: Stealing from Databases Using Compression Side Channels. In *S&P*.

[73] Homeland Security: Science and Technology. 2012. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. Technical Report. U.S. Department of Homeland Security.

[74] Yupeng Hou, Jiacheng Li, Xiangjun Fu, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. 2026. Bridging Language and Items for Retrieval and Recommendation: Benchmarking LLMs as Semantic Encoders. In *ACL*.

[75] Nan Hu, Jie Zhang, and Paul A. Pavlou. 2009. Overcoming the J-shaped distribution of product reviews. *Commun. ACM* 52, 10 (2009).

[76] Internal Revenue Service. 2022. SOI Tax Stats – Individual Income Tax Statistics – 2022 ZIP Code Data (SOI). https://www.irs.gov/statistics/soi-tax-stats-individual-income-tax-statistics-2022-zip-code-data-soi

[77] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *S&P*.

[78] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. 2024. SGXonerated: Finding (and Partially Fixing) Privacy Flaws in TEE-based Smart Contract Platforms Without Breaking the TEE. *PETS* 2024, 1 (2024).

[79] Qisheng Jiang and Chundong Wang. 2024. Sync+Sync: A Covert Channel Built on fsync with Storage. In *USENIX SEC*.

[80] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. 2022. SoK: Cryptanalysis of Encrypted Search with LEAKER – A framework for LEakage AttacK Evaluation on Real-world data. In *EuroS&P*.

[81] Seny Kamara and Tarik Moataz. 2023. *Design and Analysis of a Stateless Document Database Encryption Scheme*. Technical Report. MongoDB.

[82] David Kaplan. 2024. Hardware VM Isolation in the Cloud. *Comm. ACM* 67, 1 (2024).

[83] Adi Kaufman, Moshik Hershcovitch, and Adam Morrison. 2023. Prefix Siphoning: Exploiting LSM-Tree Range Filters For Information Disclosure. In *USENIX ATC*.

[84] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *CCS*.

[85] Florian Kerschbaum. 2015. Frequency-Hiding Order-Preserving Encryption. In *CCS*.

[86] Florian Kerschbaum and Anselme Tueno. 2019. An Efficiently Searchable Encrypted Data Structure for Range Queries. In *ESORICS*.

[87] Shaharyar Khan, Ilya Kabanov, Yunke Hua, and Stuart Madnick. 2023. A Systematic Analysis of the Capital One Data Breach: Critical Lessons Learned. *ACM TOPS* 26, 1 (2023).

[88] Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. 2023. Ethical Frameworks and Computer Security Trolley Problems: Foundations for Conversations. In *USENIX SEC*.

[89] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *S&P*.

[90] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution. In *S&P*.

[91] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In *S&P*.

[92] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *S&P*.

[93] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, There is an Oblivious RAM Lower Bound!. In *CRYPTO*.

[94] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX SEC*.

[95] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys*.

[96] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM TODS* 6, 4 (1981).

[97] Kevin Lewi and David J. Wu. 2016. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *CCS*.

[98] Dongjie Li, Siyi Lv, Yanyu Huang, Yijing Liu, Tong Li, Zheli Liu, and Liang Guo. 2021. Frequency-hiding order-preserving encryption with small client storage. *VLDB* 14, 13 (2021).

[99] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *S&P*.

[100] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *ACSAC*.

[101] Mingyu Li, Xuyang Zhao, Le Chen, Cheng Tan, Huorong Li, Sheng Wang, Zeyu Mi, Yubin Xia, Feifei Li, and Haibo Chen. 2023. Encrypted Databases Made Secure Yet Maintainable. In *OSDI*.

[102] Peijie Li, Huanhuan Chen, Kaitai Liang, and Evangelia Anna Markatou. 2025. Learning from Leakage: Database Reconstruction from Just a Few Multidimensional Range Queries. In *NDSS*.

[103] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*.

[104] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*.

[105] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. 2023. Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns. *SIGMOD* 1, 4 (2023).

[106] Evangelia Anna Markatou, Francesca Falzon, Zachary Espiritu, and Roberto Tamassia. 2023. Attacks on Encrypted Response-Hiding Range Search Schemes

in Multiple Dimensions. *PETS* 2023, 4 (2023).
[107] Evangelia Anna Markatou, Francesca Falzon, Roberto Tamassia, and William Schor. 2021. Reconstructing with Less: Leakage Abuse Attacks in Two Dimensions. In *CCS*.
[108] Evangelia Anna Markatou and Roberto Tamassia. 2019. Full Database Reconstruction with Access and Search Pattern Leakage. In *ISC*.
[109] Evangelia Anna Markatou and Roberto Tamassia. 2024. Reconstructing with Even Less: Amplifying Leakage and Drawing Graphs. In *CCS*.
[110] Frank J. Massey Jr. 1951. The Kolmogorov-Smirnov Test for Goodness of Fit. *J. Am. Stat. Assoc.* 46, 253 (1951).
[111] Microsoft. 2023. Loyal improves data protection, platform performance with Always Encrypted with secure enclaves for Azure SQL Database. https://www.microsoft.com/en/customers/story/1635968706412485429-loyal-partner-professional-services-azure-sql-database
[112] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*.
[113] Roy Murphy. [n. d.]. An Analysis of the Distribution of Birthdays in a Calendar Year. https://www.panix.com/~murphy/bday.html
[114] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *CCS*.
[115] New York Office of Health Services Quality and Analytics. 2024. *Hospital Inpatient Discharges (SPARCS De-Identified): 2022.* https://health.data.ny.gov/d/5dtw-tffi
[116] openGauss. [n. d.]. MOT Indexes. https://docs.opengauss.org/en/docs/5.1.0/docs/DatabaseAdministrationGuide/mot-indexes.html
[117] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996).
[118] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: an encrypted database using semantically secure encryption. *VLDB* 12, 11 (2019).
[119] PostgreSQL. 2026. `pageinspect`. https://www.postgresql.org/docs/current/pageinspect.html
[120] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *S&P*.
[121] Benedict Schlüter and Shweta Shinde. 2025. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *CCS*.
[122] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *USENIX SEC*.
[123] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2025. Heracles: Chosen Plaintext Attack on AMD SEV-SNP. In *CCS*.
[124] Benedict Schlüter, Christoph Wech, and Shweta Shinde. 2026. Fabricked: Misconfiguring Infinity Fabric to Break AMD SEV-SNP. In *USENIX SEC*.
[125] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. 2023. Practical Timing Side-Channel Attacks on Memory Compression. In *S&P*.
[126] Ryan Seah, Daren Khu, Alexander Hoover, and Ruth Ng. 2024. LAMA: Leakage Abuse Attacks Against Microsoft Always Encrypted. In *SECRYPT*.
[127] Alex Seto, Oytun Kuday Duran, Samy Amer, Jalen Chuang, Stephan van Schaik, Daniel Genkin, and Christina Garman. 2025. WireTap: Breaking Server SGX via DRAM Bus Interposition. In *CCS*.
[128] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. 2021. Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite. In *USENIX SEC*.
[129] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *AsiaCCS*.
[130] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*.
[131] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. 2025. Sigy: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals. In *AsiaCCS*.
[132] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *VLDB* 14, 6 (2021).
[133] Jakub Szymaszek. 2021. Royal Bank of Canada using Always Encrypted with secure enclaves. https://techcommunity.microsoft.com/blog/azuresqlblog/royal-bank-of-canada-using-always-encrypted-with-secure-enclaves/2260337
[134] Texas Department of State Health Services. 2018. *Hospital Inpatient Discharge Public Use Data File: 2018, First quarter.* https://www.dshs.texas.gov/center-health-statistics/texas-health-care-information-collection/
[135] Chia-che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*.
[136] U.S. Census Bureau. 2021. Frequently Occurring Surnames from the 2010 Census. https://www.census.gov/topics/population/genealogy/data/2010_surnames.html
[137] U.S. Census Bureau. 2022. Frequently Occurring Surnames from the Census 2000. https://www.census.gov/topics/population/genealogy/data/2000_surnames.html
[138] U.S. Census Bureau. 2025. *QuickFacts: Texas (Vintage Year 2024).* https://web.archive.org/web/20250412161446/https://www.census.gov/quickfacts/fact/table/TX/RHI125223#RHI125223
[139] U.S. Social Security Administration. [n. d.]. Popular Baby Names. https://www.ssa.gov/oact/babynames/limits.html
[140] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *S&P*.
[141] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *PETS* 2019, 3 (2019).
[142] Liang Wang, Paul Grubbs, Jiahui Lu, Vincent Bindschaedler, David Cash, and Thomas Ristenpart. 2017. Side-Channel Attacks on Shared Search Indexes. In *S&P*.
[143] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. 2022. Operon: an encrypted database for ownership-preserving data management. *VLDB* 15, 12 (2022).
[144] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*.
[145] Xingchen Wang and Yunlei Zhao. 2018. Order-Revealing Encryption: File-Injection Attack and Forward Security. In *ESORICS*.
[146] Yongzhi Wang, Yulong Shen, Cuicui Su, Jiawen Ma, Lingtong Liu, and Xuewen Dong. 2020. CryptSQLite: SQLite With High Data Security. *IEEE TC* 69, 5 (2020).
[147] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*.
[148] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.
[149] Yuqin Yan, Wei Huang, Ilya Grishchenko, Gururaj Saileshwar, Aastha Mehta, and David Lie. 2025. Relocate-Vote: Using Sparsity Information to Exploit Ciphertext Side-Channels. In *USENIX SEC*.
[150] JiHye Yang and Kee Sung Kim. 2024. Exploring the Security Vulnerability in Frequency-Hiding Order-Preserving Encryption. *Sec. Comm. Net.* 2024, 1 (2024).
[151] Xinying Yang, Cong Yue, Wenhui Zhang, Yang Liu, Beng Chin Ooi, and Jianjun Chen. 2024. SecuDB: An In-Enclave Privacy-Preserving and Tamper-Resistant Relational Database. *VLDB* 17, 12 (2024).
[152] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks with Ciphertext Side Channels. In *S&P*.
[153] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *USENIX SEC*.
[154] Sen Zhao, Haibo Hu, Zheli Liu, and Xiaofeng Chen. 2025. SARAs: Substitution and Replay Attacks in the Maintenance Mode of Encrypted Databases. *TDSC* 22, 6 (2025).
[155] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*.
[156] Jinwei Zhu, Kun Cheng, Jiayang Liu, and Liang Guo. 2021. Full encryption: an end to end encryption mechanism in GaussDB. *VLDB* 14, 12 (2021).

A Open Science

Our attack and experiment code is available at https://doi.org/10.5281/zenodo.22693233.

B Ethical Considerations

Our work involves vulnerabilities against real-world systems. Following the ACM CCS 2026 guidelines, we conducted a stakeholder-based ethics analysis based on guidelines from the Menlo Report [73] and Kohno, Acar, and Loh [88]. For each stakeholder, we consider the benefits and harms of this research.

End users. The primary stakeholders are *end users* whose records may be stored in encrypted databases. These users are not direct research participants, but they may be affected if frequency leakage enables stronger inference of sensitive attributes. Such harms could include privacy loss, financial loss, discrimination, or exposure of confidential medical, employment, consumer, or business data. Even absent direct impact, the leakage and corresponding attacks we

propose can violate users' privacy expectations. Nevertheless, we believe that the public interest is served by accurate claims about database confidentiality guarantees—this can incentivize database customers to use stronger protections or apply mitigations, which in turn benefit end users. We mitigate the impact of our research against end users by limiting our experiments to local systems under our control. We did not test against third-party systems and performed our end-to-end inference attacks on public datasets.

Database customers. Database customers are also affected. They may have relied on order-only leakage claims when deciding whether a system was appropriate for sensitive or regulated data. Our work benefits them by clarifying residual risk, but it may also impose costs for risk assessment or migration to other systems. It is impossible for us as researchers to coordinate disclosure with every possible database customer who uses one of the affected industry databases. However, we mitigate risk here through coordinated disclosure with specific vendors, who in turn are responsible to notify their customers of the potential harms, described below.

Database vendors. Vendors and maintainers of the identified systems may face reputational harm and engineering cost to mitigate vulnerabilities. We reduce unfair harm by describing the vulnerability as a widespread consequence of practical storage engine design, not as evidence of negligence by any specific product. We also further mitigated risks by initiating coordinated disclosure with identified industry vendors (Microsoft, Alibaba Cloud) on April 10, 2026, which we discuss below.

Microsoft acknowledged disclosure on April 10. On May 6, Microsoft claimed the AE security definition "implicitly" stated the RND scheme leaks frequencies in the snapshot model and closed our report. We disagreed with this and sent a follow-up on May 9. Microsoft sent a revised response on May 29, in which they agreed with our findings. They also agreed that re-encrypting the tie-breakers addresses the single-snapshot leakage. However, they chose not to take any actions at this time as they believe any mitigation would not materially change AE's posture against persistent server compromise, under which frequency information is leaked.

Alibaba did not initially respond, so we sent a second attempt on May 8. Alibaba acknowledged our disclosure on May 12. On May 13, they stated that they independently reproduced the attack against Always-Confidential indexes on which TEE-based comparison operations were explicitly enabled (a configuration we were unable to access). Alibaba's position is that enabling comparison operations relaxes Always-Confidential's default security guarantees, and that the resulting distributional leakage is an inherent consequence of that authorized configuration rather than a flaw in their security model. Accordingly, they chose not to take action at this time.

Prior to publication, we provided the above vendors with the opportunity to comment on near-final versions of the paper and the opportunity to request embargo. We then proceeded with publication after coordinating a release date with all vendors.

The research community. The security research community benefits from a clearer understanding of how practical implementation choices affect leakage. At the same time, attackers may benefit from publication by learning how to improve inference attacks once they already have snapshot access. We believe that this is outweighed by publication of our findings, which we hope will motivate and lead to development of better defenses in the research community.

The authors. The authors of this paper are also a stakeholder, but we do not believe it is possible for a research team to provide an unbiased analysis of the benefits and harms of their own research on themselves so we abstain from this analysis.

C Proof of Theorem 3.2 (and Injections)

Proof. Let $B_1, \ldots, B_m$ be the blocks of equal keys in the sorted encrypted key sequence (such that B_r contains all instances of the search key κ_r), so $\vec{\mathbf{f}} = (|B_1|, \ldots, |B_m|)$. Then, let $j_0 = 0$ and, for $r \in \{1, \ldots, m\}$,

$$j_r = \sum_{i=1}^{r} |B_i|,$$

so that B_r occupies the indices $j_{r-1} + 1, \ldots, j_r$ and the boundary between adjacent blocks B_r and B_{r+1} is at index j_r. It is clear that $\vec{\mathbf{f}} = \vec{\mathbf{f}}'$ if and only if descents $= \{j_1, \ldots, j_{m-1}\}$, and by the proof of Theorem 3.1, descents $\subseteq \{j_1, \ldots, j_{m-1}\}$, so

$$\vec{\mathbf{f}} = \vec{\mathbf{f}}' \iff u_{j_r} > u_{j_r+1} \quad \text{for all } r \in \{1, \ldots, m-1\}. \tag{1}$$

Then, fix $r \in \{1, \ldots, m-1\}$. By definition, j_r is the index of the last entry of B_r and $j_r + 1$ is the index of the first entry of B_{r+1}. Also, by lexicographic ordering on (k, u), u_{j_r} is the largest tie-breaker among the instances of κ_r and u_{j_r+1} is the smallest among the instances of κ_{r+1}. Since tie-breakers are strictly increasing in insertion time, the largest tie-breaker of κ_r belongs to the search key inserted at $t_{\max}(\kappa_r)$, and the smallest tie-breaker of κ_{r+1} belongs to the search key inserted at $t_{\min}(\kappa_{r+1})$. Therefore,

$$u_{j_r} > u_{j_r+1} \iff t_{\min}(\kappa_{r+1}) < t_{\max}(\kappa_r). \tag{2}$$

Combining Equation (1) and Equation (2) proves the theorem. □

Remark on injections. If the adversary has the ability to perform additional *injections* before taking the snapshot, they can trivially guarantee that the condition of Theorem 3.2 holds in the following way. The adversary inserts one record for each unique key in the key space $\mathcal{K} = \{\kappa_1 < \cdots < \kappa_m\}$ in descending order; that is, κ_m, then κ_{m-1}, and so on until κ_1.[6] Then, $t_{\max}(\kappa_r)$ is later than $t_{\min}(\kappa_{r+1})$ for every $r \in \{1, \ldots, m-1\}$. The adversary can then recover $\vec{\mathbf{f}}$ from the resulting $\vec{\mathbf{f}}'$ by subtracting one from each entry (to account for the injected records). We do not evaluate injections in this work, as our end-to-end experiments achieve near-perfect frequency recovery on real-world datasets without them.

D Other Vulnerable TEE-Hybrid Databases

Beyond the three systems we validate directly in Section 3, several other TEE-hybrid EDBs have published architectures that we believe satisfy the structural observations behind our attack. We were unable to obtain public artifacts for these systems, so we do not claim experimental confirmation. Instead, we describe below how the dendrochronology attack would manifest in each using evidence from their respective publications and public descriptions.

[6] If the storage engine assigns tie-breakers in the order they are specified within a single transaction (as is the case in Always Encrypted, HeDB, and StealthDB), then the m injections can be issued in a single transaction.

Alibaba Always-Confidential [143]. Operon/Always-Confidential stores encrypted data in a conventional DBMS and uses tree-based indexes for encrypted columns. Querying such a column requires TEE-based comparison to be explicitly enabled on it. Section 4.5 and Figure 7 in [143] show a B^+-tree lookup over encrypted indexed values, and, for a non-unique encrypted index with comparison operations enabled, the storage engine breaks ties between duplicate encrypted keys using a row identifier. This gives the necessary ingredients for our attack. (We note that, as part of our disclosure, Alibaba independently validated our attack against Always-Confidential indexes on which TEE-based comparison operations were explicitly enabled.)

GaussDB's Full Encryption [156]. GaussDB's Full Encryption implements ordered indexes on randomized encrypted columns using B^+-trees, where the encrypted search key entries are sorted according to their plaintext order. Figure 2 and Figure 3 in [156] illustrate this design. The paper does not explicitly mention tie-breaker behavior. However, prior documentation of GaussDB describes the use of tie-breakers on non-unique indexes [116] and so we believe our attacks apply.

Enclage [132]. Enclage presents a taxonomy for different design choices for TEE-hybrid databases and allows database practitioners to choose different configuration options. Section 3.3 of [132] presents a "strawman" construction which stores encrypted search keys in a B^+-tree with record IDs explicitly stored in plaintext. (This is additionally illustrated in Figure 1 of [132].) However, Enclage identifies plaintext record IDs as a potential source of *record identity leakage* to a persistent adversary. Thus, Enclage encrypts record IDs as part of the final design presented in Section 4 of [132]. Nevertheless, encrypting record IDs is not sufficient to completely stop snapshot adversary attacks as discussed in our multi-snapshot TEE-hybrid attacks of Section 4. We remark that certain configuration options for Enclage use encryption at the level of B^+-tree pages instead of encryption of individual search keys—in these cases, while the attacks of Section 4 may not apply, the TEE-native attacks of Section 5 may still apply.

Cipherbase [7]. Cipherbase is the precursor to Microsoft's Always Encrypted and is built on Microsoft SQL Server. Cipherbase supports range indexes which store encrypted search keys inside of a B^+-tree and uses an FGPA-based trusted hardware module to perform comparison operations over encrypted search keys. Based on our investigation of Always Encrypted and SQL Server's use of tie-breakers, Cipherbase's foundations in SQL Server mean it also likely exposes the dendrochronology vulnerability.

TrustedDB [11, 12]. TrustedDB uses cryptographic processors (SCPUs) with trusted hardware assumptions to support encrypted indexes and perform comparison operations. Equation 5 of [11] describes TrustedDB's per-attribute physical format, which explicitly includes the indexed value's primary key (i.e., record ID) in plaintext. Section 4.3 in [11] further discusses how TrustedDB is based on SQLite, which has strictly increasing record identifiers by default. This provides the required information for our attack.