Compromising Privacy in Precise Query Protocols

Jonathan L. Dautrich Jr.
Computer Science and Engineering
University of California, Riverside
dautricj@cs.ucr.edu

Chinya V. Ravishankar
Computer Science and Engineering
University of California, Riverside
ravi@cs.ucr.edu

ABSTRACT

Privacy and security for outsourced databases are often provided by *Precise Query Protocols* (PQPs). In a PQP, records are individually encrypted by a client and stored on a server. The client issues encrypted queries, which are run under encryption at the server, and the server returns the exact set of encrypted tuples needed to satisfy the query. We propose a general attack against the privacy of *all* PQPs that support range queries, using query results to partially order encrypted records. Existing attacks that seek to order etuples are less powerful and depend on weaknesses specific to particular PQPs. Our novel algorithm identifies permissible positions (loci) for encrypted records by organizing range query results using PQ-trees. These results can then be used to infer attribute values of encrypted records. We propose *equivocation* and *permutation entropy* as privacy metrics, and give experimental results that show PQP privacy to be easily compromised by our attack.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*

General Terms

Security, Algorithms

1. INTRODUCTION

Cloud computing is a popular paradigm that lets clients outsource data management, but many users still keep sensitive data offline due to privacy concerns [9]. In the *Database As a Service* model [15], clients store, update, and query data on *honest-but-curious* cloud servers. Such servers correctly process queries, but do not respect user data privacy. Data stored on the server are encrypted using keys known only to the client (see Figure 1). Queries are likewise encrypted, and may be issued only by the client. The server is the principal threat to privacy, having access to all encrypted records, queries, responses, and processing, so we treat the server as the primary attacker.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1597-5/13/03 ...$15.00.

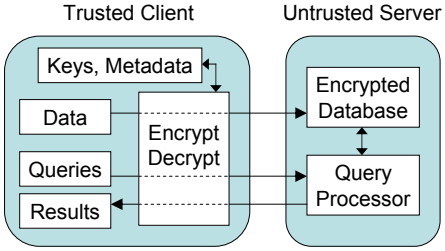

Figure 1: *Database As a Service* with encrypted data, queries, and results.

Fully secure outsourced databases require a full database scan for each query [17]. To avoid such costs, many schemes [2, 5, 8, 11, 18, 19, 27] adopt the more practical *Precise Query Protocol* (PQP) model defined in [30]. In PQPs, records are individually encrypted and stored on the server as *etuples*. The protocol is *precise* since the server returns the exact set of etuples needed to satisfy each query. PQPs are efficient as they return no spurious etuples, but their precision causes them to leak information that we can use to order the stored etuples.

We propose a novel attack on the privacy of all PQPs that support one-dimensional range queries over a *query-attribute* Q. Our attack identifies a set $\mathcal{P}$ of *permissible etuple permutations*, which are potentially-correct orderings of etuples by plaintext Q value. The permissible permutations in $\mathcal{P}$ define a partial ordering of etuples. We use the precise results returned by successive range queries to exclude permutations from $\mathcal{P}$, refining this partial ordering. Existing attacks that seek to determine such etuple orderings rely on properties specific to particular PQPs [19].

Every permutation $\pi \in \mathcal{P}$ places etuple e at some position $\pi[e]$. The set of positions for e over all $\pi \in \mathcal{P}$ yields the *permissible loci* of e. If we know the permissible loci, we can make inferences about the plaintext Q value in e. As more range queries are run, more permutations are excluded from $\mathcal{P}$, and the number of permissible loci for each etuple drops, improving inferences and further compromising privacy.

Current literature recognizes that revealing etuple order is a privacy threat. The Order-Preserving Encryption Scheme (OPES) [2], an efficient PQP that explicitly reveals etuple order, notes that privacy will be compromised if the distribution of the query-attribute is known. It is argued in [11] that schemes such as OPES should be avoided, as they allow etuples to be ordered easily. Work in [19] shows how to infer etuple order in PQPs that use prefix-preserving encryption schemes, and claims that privacy is compromised.

As in [16, 19], our goal is to enable the discovery of sensitive information associated with etuples, not necessarily to associate etu-

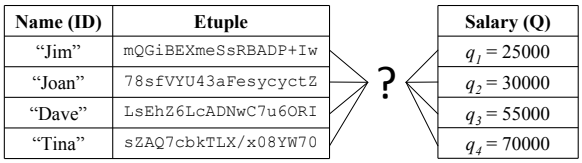

Figure 2: Employee records in a PQP. Salaries are known, but not employee-salary relationships.

ples with particular people. PQPs claim to obscure sensitive values, so revealing them clearly defeats PQP privacy guarantees. Information correlated with identity may be inferred through other attacks, or even stored in the clear.

In the example PQP database of Figure 2, etuples are employee records and range queries are issued on the salary attribute Q. Initially, we know nothing about which salary matches which etuple, so all 4 loci are permissible for each etuple. However, if we can use query result sets to exclude all permutations that assign Jim's etuple to loci 2 and 3, then only loci 1 and 4 are permissible. Consequently, we know that Jim's salary is either 25000 or 70000.

It is common to assume that attribute distributions are known [2, 7, 11, 16, 19], but our attack does not require exact knowledge of Q values or distributions. Even attribute distributions estimated from other sources, such as public Census Bureau data, suffice once we reduce the number of permissible loci sufficiently. Permissible loci can also be used in a larger attack to recover exact Q values [12].

1.1 Our Contributions

We present a novel attack on the privacy of all PQPs that support one-dimensional range queries. Our work is the first to show that etuples can be efficiently ordered in *any* such PQP. Existing attacks exploit weaknesses specific to individual PQPs, but the only requirement for our attack is the ability to observe the etuples returned by encrypted queries. In Section 3, we outline our attack and show how to use PQ-trees [6] to efficiently maintain the set of permissible permutations. Our core contribution, given in Section 4, is a novel algorithm that uses a PQ-tree to identify the permissible loci of each etuple. Query-attribute values can be inferred from these permissible loci. In Section 5, we define *equivocation* and *permutation entropy* as metrics for PQP privacy. Section 6 gives experimental results on real and synthetic datasets, showing that privacy is compromised quickly and that PQPs are highly insecure against our attack.

2. RELATED WORK

Much work exists on applying the Database As a Service model to various query types [2, 11, 15, 16, 19, 20, 23, 27]. See [25] for a survey.

2.1 Precise Query Protocol Schemes

PQPs are used for their efficiency. Table 1 shows the asymptotic costs of several PQPs. The Order Preserving Encryption Scheme (OPES) of [2] maps plaintexts to ciphertexts while preserving plaintext order and flattening the ciphertext distribution. Indexes can easily be created on the ciphertext, keys are small and nearly constant in size, and encryption costs are low. However, OPES fully reveals etuple order, making it highly vulnerable to inferences.

Prefix-preserving encryption [19] encrypts the query-attribute of each record such that if two plaintexts share a k-bit prefix, their ciphertexts share a (distinct) k-bit prefix. The prefix-preserving ciphertexts cause this scheme to leak information about etuple order more rapidly than other PQPs. PQPs such as the encrypted B$^+$-tree in [11] process queries interactively between client and server, but suffer from heavy communication loads.

Some PQPs, including MRQED [27], RASP [8], and Hidden Vector Encryption (HVE) [5], use novel encryption techniques to improve privacy. Recent work [18, 26] has used inner-product predicate encryption to implement HVE and to initially guarantee privacy of query-attribute values and query ranges. However, when the scheme is used to support one-dimensional range queries, we can still infer etuple order using our attack.

Trusted server-side hardware is used in [17] to process queries and re-encrypt etuples in order to limit the attacker's ability to make inferences. Oblivious index traversal techniques [21] are used to maintain privacy for point queries when PQPs use indexes that are visible to the server.

2.2 Imprecise Query Protocols

Many schemes sacrifice query result set precision in favor of improved privacy. In *bucketization* [15, 16] the server returns all etuples in a range of *buckets*, yielding a superset of the query result. Larger buckets improve privacy, but return more spurious etuples, raising client-side costs.

Other schemes rely on data *fragmentation*, assuming that some attributes are only sensitive when paired with others [10]. Such schemes assume non-colluding servers, high client storage capacities, or obscured table relations.

Work in [13] uses an encrypted B$^+$-tree and incorporates spurious queries, client-side caching, and node content shuffling to provide strong query access pattern privacy. Other schemes with similar goals are based on Oblivious RAM (ORAM) [28] or Private Information Retrieval (PIR) [24]. Such techniques are becoming more efficient, but still require several communication rounds per query. Work in [20] uses hierarchical predicate encryption to achieve access pattern privacy, but depends on non-colluding proxies and only supports restricted query ranges.

2.3 Prior Work on Privacy Loss

Even when the privacy of individual records is guaranteed, privacy can be compromised by careful analysis of query access patterns or indexes [29, 30]. Work in [16] discusses privacy factors of *bucketization* in terms of statistical measures such as variance and entropy, demonstrating a trade-off between privacy and efficiency.

Using relationships between encrypted records to infer plaintext information is referred to as *inference exposure* in [11]. A common technique is to exploit the fact that identical plaintexts generally produce identical ciphertexts [7, 11]. Other attacks associate frequently requested etuples with significant plaintexts, yielding probabilistic assignments of values to etuples [13].

Authors in [19] propose an attack against PQPs that use prefix-preserving encryption to support range queries. Their attack collects all pairs of etuples known to be adjacent and uses them to infer order. Our attack leads to stronger inferences as it uses everything that can be learned about etuple order from the range query results, not just what can be inferred from adjacent pairs. Further, our attack applies generally to all PQPs that support range queries.

3. ATTACK MODEL AND OUTLINE

Our attack identifies etuple orderings that are consistent with observed range query result sets. We call these orderings *permissible permutations*, and store them using a PQ-tree [6]. In Section 4, we use the PQ-tree to identify the permissible loci of each etuple, which we use to make inferences about the etuple's query-attribute values (Section 1). Notation is summarized in Table 2.

Table 1: Asymptotic costs for existing PQP schemes, with large costs highlighted. Few papers gave these costs explicitly, so the table reflects our best-effort analysis. $|D|$ is domain size, n is the etuple count, C is the result set size, N is a 512–4096 bit number, K_g and K_s are costs of group and symmetric encryption operations, respectively. δ for OPES is small with unknown relation to n.

PQP Scheme	Scheme Setup Work	Client Query Pre-Process Work	Client Storage (Bits)	Server Query Work	Query Send Size (Bits)
Order-Preserving Enc. (OPES) [2]	$O((K_s+\delta)n)$	$O(\delta)$	$O(\delta \log \lvert D\rvert + \log N)$	$O(\lvert C\rvert + \log n)$	$O(\log \lvert D\rvert)$
Prefix-Preserving Enc. [19]	$O(K_s n \log \lvert D\rvert)$	$O(K_s \log^2 \lvert D\rvert)$	$O(\log N)$	$O(\lvert C\rvert + \log n \log \lvert D\rvert)$	$O(\log^2 \lvert D\rvert)$
Encrypted B$^+$-Tree [11]	$O(K_s n)$	$O(K_s(\lvert C\rvert + \log n))$	$O(\log N)$	$O(\lvert C\rvert + \log n)$	$O((\lvert C\rvert + \log n)(\log N))$
1D MRQED [27]	$O(K_g n \log \lvert D\rvert)$	$O(K_g \log \lvert D\rvert)$	$O(\log \lvert D\rvert \log N)$	$O(K_g n \log \lvert D\rvert)$	$O((\log \lvert D\rvert)(\log N))$
Hidden Vector Enc. (HVE) [5]	$O(K_g n \lvert D\rvert)$	$O(K_g)$	$O(\lvert D\rvert \log N)$	$O(K_g n)$	$O(\log N)$
HVE with Predicate Enc. [18, 26]	$O(K_g n \lvert D\rvert)$	$O(K_g \lvert D\rvert)$	$O(\lvert D\rvert \log N)$	$O(K_g n \lvert D\rvert)$	$O(\lvert D\rvert \log N)$

Table 2: PQP and Attack Notation

e, E	Encrypted record (etuple), all etuples
r_e	Plaintext record encrypted to form e
$Q, r_e.q$	Query-attribute Q, Q value of record r_e
$[\alpha, \beta]$	Inclusive plaintext range query bounds
$E_{\alpha,\beta}$	Etuples needed to satisfy query range $[\alpha, \beta]$
π_c	Correct ordering of etuples according to Q
C	Cluster of etuples, defined by a result $E_{\alpha,\beta}$
$\mathcal{P}$	Set of permissible permutations of E
Λ_e	Permissible loci of e

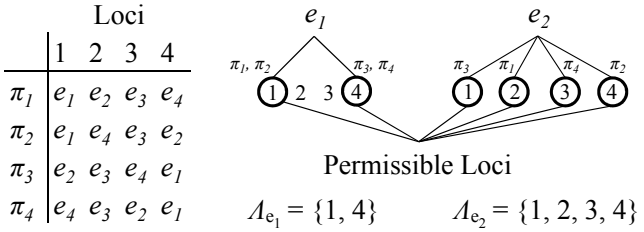

Figure 3: *Permissible loci* of etuples e_1 and e_2, given permissible permutations $\mathcal{P} = \{\pi_1, \pi_2, \pi_3, \pi_4\}$.

3.1 Precise Query Protocols (PQPs)

In a PQP, each plaintext record r is a tuple of attributes. The data owner, or *client*, encrypts each record as a single ciphertext e called an *etuple*. Let r_e denote the plaintext record that produced etuple e. The set E of all etuples is then stored on a semi-trusted, honest-but-curious server (see Section 1).

The client generates range queries over a *query-attribute* Q, encrypting the plaintext range $[\alpha, \beta]$ in each query. We let $r_e.q$ denote the Q value in record r_e. The server, without decrypting the query or any of the etuples, finds and returns the exact result set $E_{\alpha,\beta}$ satisfying the query, where:

$$E_{\alpha,\beta} = \{e \in E \mid \alpha \leq r_e.q \leq \beta\}$$

We can realize such server-oblivious querying protocols using specialized encryption techniques, ranging from order-preserving encryption to predicate encryption. We do not present details here, but see [2, 5, 8, 11, 18, 19, 27] for examples of PQPs supporting range queries.

Queries are encrypted, so they can only be generated by trusted clients, and an attacker cannot craft his own queries. Instead, he can mount the attack by observing responses to the queries issued by the client, without knowing the plaintext query ranges. Etuples may be inserted/deleted, so we let E be the subset of etuples that persist in the database across the set of issued queries. We exclude inserted/deleted etuples before mounting the attack.

3.2 Permissible Permutations and Loci

Definition 1. A *correct ordering* of E by attribute Q is a permutation $\pi = e_1 e_2 \ldots$ of E with $r_{e_1}.q \leq r_{e_2}.q \leq \cdots$.

If several etuples have the same Q value, there are multiple correct orderings. Our attack handles this case, and both our experimental datasets include repeated values. However, for ease of presentation, we introduce the attack as though only one correct ordering, π_c, exists. We can use *clusters* to learn which permutations might be π_c.

Definition 2. A *cluster* $C \subseteq E$ is a subset of etuples that are contiguous in π_c. $\mathcal{C}$ denotes a set of clusters.

Let $E_{\alpha,\beta} \subseteq E$ be the set of etuples returned by a query on any range $[\alpha, \beta]$. Since $E_{\alpha,\beta}$ contains precisely those $e \in E$ for which $\alpha \leq r_e.q \leq \beta$, etuples in $E_{\alpha,\beta}$ are contiguous in π_c. Thus, each query result set $E_{\alpha,\beta}$ is a cluster.

Definition 3. A cluster C *excludes* a permutation π of E if an etuple $e_i \notin C$ appears in π between two $e_j, e_k \in C$. Once π has been excluded, we know that $\pi \neq \pi_c$.

Definition 4. The set $\mathcal{P} = \{\pi_1, \pi_2, \ldots\}$ of *permissible permutations* consists of all permutations of E not excluded by any cluster. A permutation π is permissible if and only if for every cluster C, the etuples in C are contiguous in π.

Each permutation in $\mathcal{P}$ is potentially the correct ordering, given the observed clusters, so $\mathcal{P}$ defines a partial ordering on E. Initially, every permutation is in $\mathcal{P}$. As queries arrive, we can identify clusters and use them to exclude from $\mathcal{P}$ any permutations in which clustered etuples are not contiguous, thereby refining the partial ordering.

Consider the etuple set $E = \{I, J, K\}$. Initially, all six permutations of E are permissible. A range query returning J and K defines a cluster $C = \{J, K\}$. Permutations JIK and KIJ are excluded by C, since $I \notin C$ appears between $J, K \in C$. Only four permutations remain permissible.

Definition 5. The *locus* $\ell = \pi[e]$ of etuple e in permutation π is the position of e in π, such that e is the ℓth etuple in π. The *permissible loci* Λ_e of e are the set of loci of e across all permissible permutations $\pi \in \mathcal{P}$ (see Figure 3):

$$\Lambda_e = \{\pi[e] \mid \pi \in \mathcal{P}\}$$

Λ_e gives possible positions of e in the correct ordering π_c. Etuples in π_c are ordered by query-attribute value, so if we have even partial knowledge of the query-attribute distribution, we can use Λ_e to make inferences about the value of $r_e.q$. As more clusters

are observed, more permutations are excluded from $\mathcal{P}$, lowering $|\Lambda_e|$ and reducing uncertainty about $r_e.q$.

A cluster that excludes any permutation π must also exclude its reverse $\overline{\pi}$. Thus, given enough distinct clusters, we can exclude permutations until $\mathcal{P} = \{\pi_c, \overline{\pi_c}\}$, but no further. Therefore, we always have $|\mathcal{P}| \geq 2$.

3.3 Using PQ-Trees to Maintain $\mathcal{P}$

A *PQ-tree* [6] is a rooted, ordered tree capable of compactly representing the set of permissible permutations $\mathcal{P}$ derived from cluster set $\mathcal{C}$. In fact, PQ-trees were designed specifically to represent such permutations. Non-leaf nodes in PQ-trees are of type P or Q. Leaves represent etuples from E, so the terms *etuple* and *leaf* will be used interchangeably. The sequence of leaves reached by a pre-order traversal of a PQ-tree T forms its *frontier* $F(T)$, which defines a permutation of E.

Every P-node and Q-node must have at least two children. Children of a P-node can be arbitrarily permuted, while children of a Q-node may only be reversed. Each combination of rearranged children in T produces an *equivalent* tree $T' \equiv T$. The set $\mathcal{P}_T$ of permutations consistent with T is $\mathcal{P}_T = \{F(T') \mid T' \equiv T\}$.

All PQ-tree leaves start as children of a single P-node, forming a *universal* tree consistent with all $|E|!$ possible permutations. Using a cluster C, we can transform T into a new PQ-tree T^* through a *reduction* operation. The permutations consistent with T^* are those that are consistent with T *and* in which all etuples in C are contiguous. The reduction algorithm runs in time $O(|C|)$ [6].

After successively reducing T using each $C \in \mathcal{C}$, T precisely represents $\mathcal{P}$, giving $\mathcal{P}_T = \mathcal{P}$. With enough distinct clusters, we can reduce T until all its leaves are children of a single Q-node, at which point $\mathcal{P}_T = \{\pi_c, \overline{\pi_c}\}$.

Let $n = |E|$ be the number of leaves (etuples) in T. Since every non-leaf node in T has at least two children, the number of nodes m in the tree is at most $2n - 1$. The height of T ranges from 1 when all etuples are children of a single P or Q-node, to $n - 1$, such as when T is left-deep.

3.4 Characteristic Examples

We now work through a few simple examples to demonstrate our attack. In each example, we apply a set of clusters $\mathcal{C}$ to the etuple set $E = \{I, J, K, L, M, N\}$, letting $\pi_c = IJKLMN$. We then describe the set of permissible permutations $\mathcal{P}$ consistent with $\mathcal{C}$, show the corresponding PQ-tree, and identify the permissible loci of several etuples. PQ-tree diagrams represent P-nodes using circles and Q-nodes using rectangles. Etuples are represented by their labels $I \cdots N$.

By Definition 4, a permutation π is in $\mathcal{P}$ if and only if for every cluster $C \in \mathcal{C}$, etuples in C are contiguous in π. It is helpful to think of each etuple as a point on a line, where point e has value $r_e.q$. Etuples in a cluster may appear in any order, as long as they are together on the line and have no other etuples between them.

As more clusters are added, the PQ-tree's structure can become quite complex, so we cannot provide examples for all cases here. For a more thorough demonstration of PQ-trees, see [6]. If $\mathcal{C} = \emptyset$, all permutations are permissible. In this case, all etuples are children of a single P node, and every etuple has all 6 possible permissible loci.

Example 1. (Disjoint Clusters) Let $C_1 = \{I, J, K\}$, and let $C_2 = \{L, M\}$, as in Figure 4. Permutations in $\mathcal{P}$ must not intersperse etuples in C_1, C_2, and $\{N\}$. However, the sets themselves and the elements within each set may appear in any relative order. There are 3! ways to permute $\{C_1, C_2, \{N\}\}$, 3! ways to permute C_1, 2 for C_2, and 1 for $\{N\}$. The PQ-tree representing $\mathcal{P}$ is given in Figure 5. For each $e \in E$, we can still find a $\pi \in \mathcal{P}$ that assigns e to any one of the 6 loci, so all etuples have all 6 permissible loci.

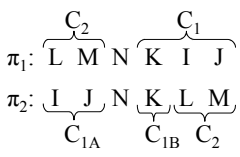

Figure 4: Permutation π_1 is permitted but π_2 is excluded as C_1 is split by $N \notin C_1$ in π_2.

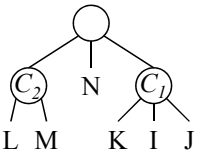

Figure 5: PQ-tree with frontier $LMNKIJ$ reduced using clusters $\{I, J, K\}$, $\{L, M\}$.

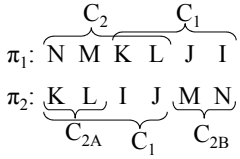

Figure 6: Permutation π_1 is permitted but π_2 is excluded as C_2 is split by $I, J \notin C_2$ in π_2.

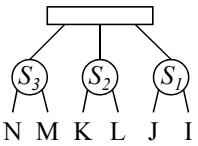

Figure 7: PQ-tree with frontier $NMKLJI$ reduced using $\{I, J, K, L\}$, $\{K, L, M, N\}$.

Example 2. (Cluster Overlap) Let $C_1 = \{I, J, K, L\}$ and $C_2 = \{K, L, M, N\}$ as in Figure 6. We define sets $S_1 = \{I, J\}$, $S_2 = \{K, L\} = C_1 \cap C_2$, and $S_3 = \{M, N\}$. Considering etuples as points on a line, we know that etuples in C_1 appear together, as do etuples in C_2. To meet both of these conditions, etuples in $C_1 \cap C_2$ must together fall between the remaining etuples. That is, etuples in S_2 must be together, and must separate the etuples in S_1 from those in S_3.

Thus, S_1, S_2, and S_3 form clusters of their own, and they appear in order $S_1S_2S_3$ or $S_3S_2S_1$. We have 2 ways to permute elements within each of S_1, S_2, and S_3. The PQ-tree is given in Figure 7, where we use a Q node to represent the fact that the sets S_1, S_2, S_3 can be in one of two orders. Etuples I, J, M, N are restricted to permissible loci $\Lambda_e = \{1, 2, 5, 6\}$, while K and L are restricted to $\Lambda_e = \{3, 4\}$.

Example 3. (All 2-Clusters) Figure 8 shows all possible size-two clusters. Applying arguments from Example 2 to each pair of intersecting clusters, we get $\mathcal{P} = \{\pi_c, \overline{\pi_c}\}$. The PQ-tree is given in Figure 9. Each etuple has 2 permissible loci. I and N each have $\Lambda_e = \{1, 6\}$, J and M have $\Lambda_e = \{2, 5\}$, and K and N have $\Lambda_e = \{3, 4\}$.

4. IDENTIFYING PERMISSIBLE LOCI

We give a novel algorithm for identifying permissible loci Λ_e of every etuple $e \in E$ given a PQ-tree T. The algorithm runs in time $O(n^2 \log n)$ and requires $O(n \log n)$ space, where $n = |E|$, and includes a dynamic programming solution for a series of

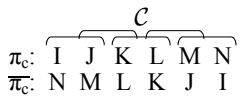

Figure 8: The clusters in $\mathcal{C}$ permit *only* π_c and $\overline{\pi_c}$.

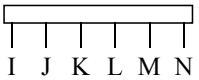

Figure 9: PQ-tree fully reduced using all size-two clusters.

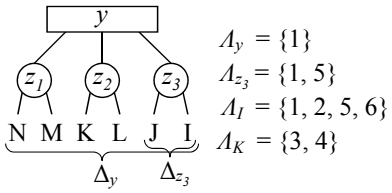

Figure 10: Labeled PQ-tree (Example 4).

related subset sum problems that we must solve for each P node. Partial results for each solution are cached in a depth-first manner to exploit problem similarities. We also give a variation called κ-*pruning*, which finds Λ_e for etuples with $|\Lambda_e| < \kappa$ in $O(n\kappa \log \kappa)$ time and $O(\kappa \log n)$ space. Key notation is summarized in Table 3.

4.1 Algorithm Outline and Terminology

Let y be a node in PQ-tree T. Our goal is to identify the permissible loci of each etuple (leaf) in T.

Definition 6. The *etuple descendants* Δ_y of y are the etuples (leaves) descended from y in T. If y is a leaf, $\Delta_y = \{y\}$.

Definition 7. The *spread* σ_y is the number of etuple descendants of y, $\sigma_y = |\Delta_y|$. We can pre-compute spreads for all nodes in time $O(n)$.

Definition 8. The symbol η represents an *offset*, which denotes the total number of etuples descended from all siblings that precede a given node in a PQ-tree.

Definition 9. The *locus* of a node y in PQ-tree T' is the position in the frontier $F(T')$ at which etuples in Δ_y begin to appear. The *permissible loci* Λ_y of y are the set of all such loci of y across all $T' \equiv T$. If y is the root, $\Lambda_y = \{1\}$. Since each such frontier is a permissible permutation of etuples (Section 3.3), this definition generalizes Definition 5 from etuples to any node. For brevity, we often refer to Λ_y as simply the *loci* of y.

Let $z_1, \ldots, z_c$ be the $c \geq 0$ children of y. The locus of z_i in T' is offset from the locus of y by the spreads of all children of y that precede z_i in T'. The subsets of children that may precede z_i in any tree $T' \equiv T$ are determined by y's node type (P or Q). Thus, given Λ_y, y's type, and the spreads $\sigma_{z_1}, \ldots, \sigma_{z_c}$ of each of y's children, we can identify $\Lambda_{z_1}, \ldots, \Lambda_{z_c}$. We apply this technique recursively to identify the permissible loci of all nodes in T, including its leaves.

Example 4. In Figure 10, y is the root of a PQ-tree, so no leaves can precede Δ_y, and $\Lambda_y = \{1\}$. Since y is a Q node, its children can be reversed, so exactly 0 or 4 of the leaves in Δ_y must precede Δ_{z_3}. Thus, the permissible loci Λ_{z_3} are offset from Λ_y by $\eta_1 = 0$ and $\eta_2 = 4$, giving $\Lambda_{z_3} = \{1, 5\}$. Similarly, $\eta_3 = 0$ or $\eta_4 = 1$ of the leaves in Δ_{z_3} precede Δ_I for each locus in Λ_{z_3}, so $\Lambda_I = \{1, 2, 5, 6\}$.

4.2 Identifying Loci for Children of Q-Nodes

Let y be a Q-node in T. We let $\overleftarrow{\eta_{z_i}}$ be the number of etuples in Δ_y that precede Δ_{z_i} in T, and $\overrightarrow{\eta_{z_i}}$ the number that precede Δ_{z_i} in the equivalent tree where y's children are reversed. Thus, $\overleftarrow{\eta_{z_i}} = \sum_{j=1}^{i-1} \sigma_{z_j}$ and $\overrightarrow{\eta_{z_i}} = \sum_{j=i+1}^{c} \sigma_{z_j}$. Λ_{z_i} are offset from Λ_y by either $\overleftarrow{\eta_{z_i}}$ or $\overrightarrow{\eta_{z_i}}$, as in Figure 11:

$$\Lambda_{z_i} = \{\ell + \overleftarrow{\eta_{z_i}} \mid \ell \in \Lambda_y\} \cup \{\ell + \overrightarrow{\eta_{z_i}} \mid \ell \in \Lambda_y\} \qquad (1)$$

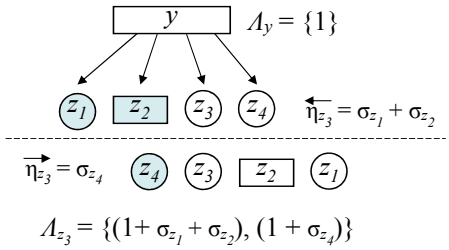

Figure 11: The loci Λ_{z_3} are offset from Λ_y by $\overleftarrow{\eta_{z_3}}$ when y's children are ordered left-to-right, and by $\overrightarrow{\eta_{z_3}}$ when ordered right-to-left.

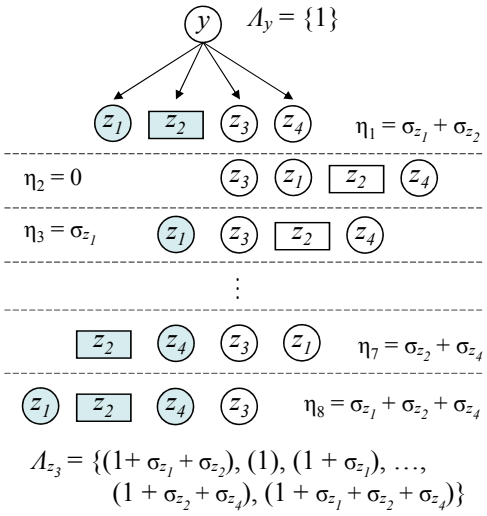

Figure 12: For P-node y, each subset of children to the left of z_3 yield an offset η of Λ_{z_3} from Λ_y.

We call this addition/union operation an *expansion* of Λ_y to Λ_{z_i}, since $|\Lambda_{z_i}| \geq |\Lambda_y|$. Since $|\Lambda_y| \in O(n)$ for all $y \in T$, each expansion takes $O(n)$ time. We perform one such expansion per child, so the per-child cost is $O(n)$. Pseudocode is given in Algorithm 1, Lines 12–23. For space efficiency, we perform the expansion for the child with the largest spread last (see Section 4.4).

4.3 Identifying Loci for Children of P-Nodes

If y is a P-node in T, any permutation of y's children $z_1, \ldots, z_c$ yields an equivalent PQ-tree. Since a child z_i may be preceded by any subset of the other $c-1$ children, the number of possible offsets for the loci Λ_{z_i} from Λ_y may be large. In contrast, Q-node children may only be reversed, so there are at most two offsets for each child. Thus, computing loci for P-node children is more challenging.

As Figure 12 shows, the sum of spreads of every possible subset of the other $c-1$ children is a valid offset η of Λ_{z_i} from Λ_y. To test whether a given value of η is an offset, we must solve a *subset-sum* problem, where the target sum is η and the list of integers is the multiset $\{\sigma_{z_j} \mid j \neq i\}$ of the spreads of the $c-1$ children.

4.3.1 Each Child Considered Separately

The general subset-sum problem is NP-complete, but we know that the sum of all child spreads never exceeds n. Using this fact, $\sum_{j=1}^{c} \sigma_{z_j} \leq n$, we can compute all the offsets of any Λ_{z_i} from Λ_y in time $O(nc)$ using the standard pseudo-polynomial-time dynamic

Table 3: Notation for Node y of PQ-Tree T

E, n	Set/number of etuples (leaves) in T
m	Total number of nodes (P, Q, and leaf) in T
$\mathcal{P}_T$	Set of permutations of E consistent with T
C	*Cluster* of etuples defined by a query result
Δ_y	The set of etuples descended from y
σ_y	The *spread* of y, $\sigma_y \equiv \lvert\Delta_y\rvert$
Λ_y	The *permissible loci* of y (starting loci of Δ_y)
Φ_y	A set of *intermediate loci* used in computation
η	An *offset*; a number of etuples preceding a node

programming algorithm for enumerating subset sums [14]. The dynamic program is based on the insight that we can choose each child $z_j, j \neq i$ to precede or succeed z_i *independently*. Thus, when computing subset sums, we can independently add in or not add in each spread $\sigma_{z_j}, j \neq i$, in any order.

Once we compute the possible offsets, we must add them to the loci in Λ_y as we did for Q-nodes in Equation 1. We can combine both steps by initializing the algorithm with Λ_y, yielding the following recurrence for Λ_{z_i}, where Φ_j represents the intermediate loci set obtained after considering the first j children:

$$\Lambda_{z_i} = \Phi_c, \text{ where} \tag{2}$$

$$\Phi_j = \begin{cases} \Lambda_y, & j = 0 \\ \Phi_{j-1}, & j = i \\ \Phi_{j-1} \cup \{\ell + \sigma_{z_j} \mid \ell \in \Phi_{j-1}\}, & \textit{otherwise} \end{cases}$$

We make no change to the intermediate loci when $j = i$, since z_i cannot precede itself. As in Equation 1, we call each addition/union operation an *expansion* with $O(n)$ cost. Identifying Λ_{z_i} for a single child z_i using Equation 2 requires $c-1$ expansions, so the per-child cost is $O(nc)$.

Example 5. Let y be a P-node with $\Lambda_y = \{1\}$ and 4 children as in Figure 12. Let $\sigma_{z_1} = 2, \sigma_{z_2} = 3, \sigma_{z_4} = 2$. We show how to find Λ_{z_3} using Equation 2 with $i = 3$. First, we have $\Phi_0 = \Lambda_y = \{1\}$. We expand with σ_{z_1} to get $\Phi_1 = \{1\} \cup \{1+2\} = \{1, 3\}$, then expand with σ_{z_2} to get $\Phi_2 = \{1, 3\} \cup \{1+3, 3+3\} = \{1, 3, 4, 6\}$. $\Phi_3 = \Phi_2$ since we skip over σ_{z_3}, and we expand with σ_{z_4} to get $\Phi_4 = \{1, 3, 4, 6\} \cup \{1+2, 3+2, 4+2, 6+2\} = \{1, 3, 4, 5, 6, 8\}$.

4.3.2 All Children Considered Together

If z_i, z_k are children of y, a direct application of Equation 2 identifies Λ_{z_i} and Λ_{z_k} by successively expanding Λ_y with each spread in $\{\sigma_{z_j} \mid j \neq i\}$ and $\{\sigma_{z_j} \mid j \neq k\}$, respectively. Thus, both Λ_{z_i} and Λ_{z_k} require expansions that use the *shared* spreads $\{\sigma_{z_j} \mid j \neq i, k\}$. We can reduce the per-child cost of our algorithm from $O(nc)$ to $O(n \log c)$ by limiting the number of expansions performed with shared spreads.

Since y is a P-node, all child orders are legal. Thus, when we identify any Λ_{z_i} using Equation 2, we can change the order in which we consider spreads for expansion, as long as we skip over σ_{z_i}. By manipulating the spread order, we can avoid unnecessarily repeating expansions with shared spreads when identifying both Λ_{z_i} and $\Lambda_{z_k}, k \neq i$.

For example, we can first expand Λ_y using all the shared spreads $\{\sigma_{z_j} \mid j \neq i, k\}$ to get the intermediate loci set Φ_{z_i,z_k}. We then expand Φ_{z_i,z_k} using σ_{z_k} to get Λ_{z_i}, and expand Φ_{z_i,z_k} using σ_{z_i} to get Λ_{z_k}, reducing the number of expansions from $2c - 2$ to c. We can apply this principle recursively to efficiently identify the loci of every child of y.

Algorithm 1 Identifying Λ_e for all $e \in E$ descended from node y. Children of y are $z_1, \ldots, z_c$.

```
1:  procedure TraverseNode(y, Λ_y)
2:      if y ∈ E then
3:          report Λ_e = Λ_y for etuple e = y
4:      else if y is a Q-node then
5:          QNode(y, Λ_y)
6:      else
7:          sort y's children s.t. σ_{z_1} ≤ σ_{z_2} ≤ ··· ≤ σ_{z_c}
8:          PNode(y, Λ_y, [1, ..., c])
9:      end if
10: end procedure
11:
12: procedure QNode(y, Λ_y)
13:     max ← index of child z_max of y with max σ_{z_max}
14:     for i ← 1 ... c do
15:         if i ≠ max then
16:             Λ_{z_i} ← Expand(Λ_y, Σ_{j=1}^{i-1} σ_{z_j}, Σ_{j=i+1}^{c} σ_{z_j})
17:             TraverseNode(z_i, Λ_{z_i})
18:         end if
19:     end for
20:     Λ_{z_max} ← Expand(Λ_y, Σ_{j=1}^{max-1} σ_{z_j}, Σ_{j=max+1}^{c} σ_{z_j})
21:     Destroy(Λ_y)
22:     TraverseNode(z_max, Λ_{z_max})
23: end procedure
24:
25: procedure PNode(y, Φ, S)
26:     if |S| = 1 then TraverseNode(z_{S(1)}, Φ)
27:     else
28:         Φ' ← copy of Φ
29:         mid ← ⌊|S|/2⌋
30:         for i ← (mid + 1) ... |S| do
31:             Φ' ← Expand(Φ', 0, σ_{z_{S(i)}})
32:         end for
33:         PNode(y, Φ', [S(1), ..., S(mid)])
34:         Destroy(Φ')
35:         for i ← 1 ... mid do
36:             Φ ← Expand(Φ, 0, σ_{z_{S(i)}})
37:         end for
38:         PNode(y, Φ, [S(mid + 1), ..., S(|S|)])
39:     end if
40: end procedure
41:
42: function Expand(Φ, η_1, η_2)
43:     return {ℓ + η_1 | ℓ ∈ Φ} ∪ {ℓ + η_2 | ℓ ∈ Φ}
44: end function
```

We use a depth-first divide-and-conquer approach. Expansions using the shared spreads from the second half of the children, given by $\sigma_{z_{(c/2)+1}}, \ldots, \sigma_{z_c}$, are common to identifying loci for each of the first half of the children $\Lambda_{z_1}, \ldots, \Lambda_{z_{c/2}}$, and vice-versa. Identifying loci for each fourth of the children we use spreads from the other three fourths, etc.

Example 6. Let y be a PNode with children $z_1, \ldots, z_8$, as in Figure 13. Let $\Phi_{z_i,\ldots,z_k}$, $i \leq k$, represent the intermediate loci after expanding Λ_y using all shared spreads common to identifying $\Lambda_{z_i}, \ldots, \Lambda_{z_k}$. Our goal is to identify all loci $\Lambda_{z_1}, \ldots, \Lambda_{z_8}$. We first identify and cache Φ_{z_1,z_2,z_3,z_4} by expanding Λ_y using spreads $\sigma_{z_5}, \sigma_{z_6}, \sigma_{z_7}, \sigma_{z_8}$. We then identify and cache Φ_{z_1,z_2} by expanding Φ_{z_1,z_2,z_3,z_4} using $\sigma_{z_3}, \sigma_{z_4}$. Finally, we identify Λ_{z_1} by expanding Φ_{z_1,z_2} using σ_{z_2}. We then backtrack and again expand the

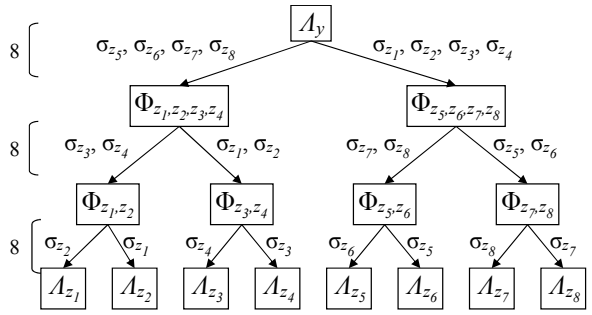

Figure 13: Identifying permissible loci for children of P-node y. Each arrow indicates expansions using the labeled spreads. Eight expansions are performed at each of three levels. (Example 6)

cached Φ_{z_1,z_2}, this time with σ_{z_1}, to get Λ_{z_2}. We backtrack again to identify Φ_{z_3,z_4} by expanding Φ_{z_1,z_2,z_3,z_4} using $\sigma_{z_1}, \sigma_{z_2}$, and so on until we identify Λ_{z_8}. In all, we perform only $8 \log_2 8 = 24$ expansions, instead of $8 \cdot (8-1) = 56$, as we would if we identified loci for each child separately.

Pseudocode is given in Algorithm 1, Lines 7, 8, and 25–40. The children are initially sorted by spread for space efficiency (see Section 4.4). S is a sequence of child indexes, initialized to $[1, \ldots, c]$ (Line 8). If $|S| = 1$, the intermediate loci Φ are the loci of a particular child (Line 26), and we can recursively find the loci of that child's children. Otherwise, we split S in half, and save a copy of Φ. We then expand the copy of Φ using the *second* half of the spreads $\sigma_{z_{S(\lfloor |S|/2 \rfloor + 1)}}, \ldots, \sigma_{z_{S(|S|)}}$, and recursively call PNODE with the expanded Φ and the first half of the spreads, given by $\sigma_{z_{S(1)}}, \ldots, \sigma_{z_{S(\lfloor |S|/2 \rfloor)}}$ (Lines 28–33). We then repeat the process, expanding Φ using the *first* half of the spreads, and recursively calling PNODE using the second half of the spreads (Lines 35–38).

The recursion has $O(\log c)$ levels, and we expand using each spread at most once for each level, so the total cost is $O(nc \log c)$, with a per-child cost of only $O(n \log c)$, down from $O(nc)$ when the loci of each child are identified separately (Section 4.3.1).

4.4 Analysis and Space-Time Tradeoff

Algorithm 1 combines our methods for P and Q-node children using a depth-first approach. Once we identify the permissible loci for a node in T, we immediately start identifying the permissible loci of its children. Algorithm 1 identifies loci for $O(n)$ nodes and generates $O(n)$ sets of intermediate loci via recursive calls to PNODE, with $O(n)$ loci in each set.

There is a tradeoff between improved speed if we cache more loci and reduced space if we cache fewer. One possible extreme is to store all $O(n)$ loci sets, using $O(n^2)$ space. However, since we can easily have $n \geq 10^6$, storing $O(n^2)$ items in memory is unacceptable. (Even storing the permissible loci for every etuple takes $O(n^2)$ space, but we assume that the attacker logs any loci of long-term interest to secondary storage.)

Another extreme is to store only one set of loci at a time, and apply Equations 1 and 2 directly. Since we do not cache any intermediate loci, we must start from the root every time we want to identify Λ_e for a different e. For each node y on the path from the root to e, this algorithm takes time $O(nc)$, where y has c children. Thus, identifying Λ_e requires $O(n^2)$ time for each etuple, but only $O(n)$ total space. Repeating for all n etuples takes time $O(n^3)$.

Algorithm 1 is a compromise between these extremes. It caches intermediate loci, but discards them as soon as possible (Lines 21 and 34). By carefully structuring our traversal of T, we can find all Λ_e in time $O(n^2 \log n)$, using only $O(n \log n)$ space. For example, we can discard the loci of a Q-node as soon as we identify the loci of its last child. Since T can have height $O(n)$, a naïve traversal still requires $O(n^2)$ space, so we must carefully select the order in which we perform expansions.

THEOREM 1. *Algorithm 1 takes time $O(n^2 \log n)$.*

PROOF. Per-child time is $O(n)$ for children of Q-nodes (Section 4.2) and $O(n \log c)$ for children of P-nodes (Section 4.3.2). The per-child time for sorting children of P-nodes (Line 7) is only $O(\log c)$. Thus, the worst-case per-child time in Algorithm 1 is $O(n \log c) \subset O(n \log n)$. Since T has n leaves, and each non-leaf node has at least two children, there are $O(n)$ nodes, and thus $O(n)$ children, in T. With $O(n)$ children, and time $O(n \log n)$ per child, the total time for Algorithm 1 is in $O(n^2 \log n)$. □

The parameters in Algorithm 1 are pass-by-reference. Before we prove that Algorithm 1 requires $O(n \log n)$ space, we must introduce the following definitions.

Definition 10. A *caching call* is a procedure call to QNODE or PNODE for which we are caching a set of loci (Λ_y in QNODE, Φ' in PNODE). Each call is a caching call until its cached loci are destroyed (Lines 21, 34), except for PNODE calls with $|S| = 1$, which are not caching calls.

Definition 11. The *coverage* of a call is the number of etuples for which Λ_e is reported (Line 3) by the call and all its descendant calls. Thus, the coverage of a call is the sum of coverages of its sub-calls. A TRAVERSENODE or QNODE call has coverage σ_y, and a PNODE call has coverage $\sum_{i=1}^{|S|} \sigma_{z_{S(i)}}$.

LEMMA 1. *Calls to* QNODE *and* PNODE *in Algorithm 1 always make the sub-call with the largest coverage last.*

PROOF. For calls to QNODE, all sub-calls are calls to TRAVERSENODE on the children $z_1, \ldots, z_c$. Thus, by Definition 11, the coverage of a sub-call for child z_i is the spread σ_{z_i}. Since Algorithm 1 explicitly makes the call for the child with the largest spread last, it makes the sub-call with the largest coverage last.

For calls to PNODE, if $|S| = 1$, the sub-call with largest coverage is the last and only sub-call. Otherwise, PNODE makes two sub-calls to PNODE. The first sub-call has coverage given by the sum $\sum_{j=1}^{\lfloor |S|/2 \rfloor} \sigma_{z_{S(j)}}$, and the second by $\sum_{j=\lfloor |S|/2 \rfloor + 1}^{|S|} \sigma_{z_{S(j)}}$. The second sub-call's coverage sums at least as many spreads as the first, since $\lfloor |S|/2 \rfloor \leq |S|/2$. Further, the indexes in S are always in increasing order, and the children are sorted such that for any two children z_i, z_k, with $i < k$, we have that $\sigma_{z_i} \leq \sigma_{z_k}$ (Line 7). Thus, $\sum_{j=1}^{\lfloor |S|/2 \rfloor} \sigma_{z_{S(j)}} \leq \sum_{j=\lfloor |S|/2 \rfloor + 1}^{|S|} \sigma_{z_{S(j)}}$, so the last (second) sub-call has the largest coverage. □

LEMMA 2. *The coverage of each caching call is at least twice that of its active sub-call.*

PROOF. Let ψ be a caching call with active sub-call λ and last sub-call ω. By Lemma 1, ω is the sub-call with the largest coverage. By Definition 10, ψ must be a call to PNODE or QNODE, and $\lambda \neq \omega$, since cached loci are destroyed before sub-call ω is made. The coverage of ψ is at least that of λ and ω combined, and since the coverage of ω is at least that of λ, the coverage of ψ is at least twice that of λ. □

THEOREM 2. *Algorithm 1 requires $O(n \log n)$ space.*

PROOF. Let e be a leaf for which we are reporting Λ_e (Line 3), and let χ be the number of currently cached loci sets. Since loci are only cached by a caching call, χ is also the number of caching calls in the current call stack.

The coverage of the root's TRAVERSENODE call is n. By Definition 11, the coverage of any call is at least as large as the coverage of each of its sub-calls. By Lemma 2, the coverage of the ith deepest caching call is at least twice that of the $(i+1)$st deepest caching call. Thus, the coverage of the deepest call is at most $n/2^\chi$. The deepest call is the TRAVERSENODE call reporting Λ_e, which has coverage $\sigma_e = 1$ according to Definition 11. Thus, we have:

$$\frac{n}{2^\chi} \geq 1 \;\rightarrow\; 2^\chi \leq n \;\rightarrow\; \chi \leq \log_2 n \tag{3}$$

Each of the χ cached loci sets consumes $O(n)$ space, so Algorithm 1 requires $O(n \log n)$ space. □

4.5 The κ-Pruning Variant

Etuples with fewer permissible loci (smaller $|\Lambda_e|$) yield more information (Section 3.2). Thus, we may want to identify Λ_e for only those etuples with $|\Lambda_e| < \kappa$, for some threshold κ. We can prune the call tree in Algorithm 1 to find such loci in $O(\kappa \log n)$ space and $O(n\kappa \log \kappa)$ time. We refer to the resulting algorithm as the *κ-pruning* variant.

If z_i is a child of y, we expand Λ_y to get Λ_{z_i}, so $|\Lambda_{z_i}| \geq |\Lambda_y|$. Thus, if $|\Lambda_y| \geq \kappa$, all etuple descendants of y will have $|\Lambda_e| \geq \kappa$, and we can *prune* y, skipping the TRAVERSENODE calls for y and all its descendants. Since we need only traverse nodes with $|\Lambda_y| < \kappa$, we need only store loci sets with at most $O(\kappa)$ loci, so κ-pruning has space complexity $O(\kappa \log n)$. When y is a P-node, we may be able to use the following theorem to prune y even if $|\Lambda_y| < \kappa$.

THEOREM 3. *If y is a P-node with children $z_1, \ldots, z_c$, then for every child z_i, $|\Lambda_{z_i}| \geq |\Lambda_y| + c - 1$.*

PROOF. Let m_j be the maximum value in Φ_j in Equation 2. For each expansion $(j \neq i)$, σ_{z_j} is added to each element in Φ_{j-1}, including m_{j-1}, and the results are placed in Φ_j. Thus, $m_j \geq \sigma_{z_j} + m_{j-1}$. Since $\sigma_{z_j} \geq 1$, $m_j > m_{j-1}$ and thus $m_j \notin \Phi_{j-1}$. Since $\Phi_{j-1} \subset \Phi_j$, and Φ_j has at least one element (m_j) that is not in Φ_{j-1}, we know that $|\Phi_j| \geq |\Phi_{j-1}| + 1$. We perform $c-1$ expansions going from Φ_0 to Φ_c, so $|\Lambda_{z_i}| = |\Phi_c| \geq |\Phi_0| + (c-1) = |\Lambda_y| + (c-1)$, and thus $|\Lambda_{z_i}| \geq |\Lambda_y| + c - 1$. □

By Theorem 3, we know that if y is a P-node, and if $|\Lambda_y| + c - 1 \geq \kappa$, then for every child z_i of y, $|\Lambda_{z_i}| \geq \kappa$. Thus, we can prune y if it has at least $c \geq \kappa - |\Lambda_y| + 1$ children. Since we always have $|\Lambda_y| \geq 1$, we can always prune y if $c \geq \kappa$.

Algorithm 1 requires $O(1)$ expansions per child for a Q-node, and $O(\log c)$ per child for a P-node. Since we need only traverse P-nodes with $c < \kappa$, we need at most $O(\log \kappa)$ expansions per child. Loci sets now contain at most $O(\kappa)$ loci, so each expansion takes time $O(\kappa)$. In pathological cases, we still traverse $O(n)$ children, so the total time for κ-pruning is $O(n\kappa \log \kappa)$. In practice, κ-pruning runs much faster than this asymptotic bound.

The following theorem will be used in Section 6.

THEOREM 4. *If at least $\kappa - 1$ etuples appear in none of the clusters in $\mathcal{C}$, then every etuple has $|\Lambda_e| \geq \kappa$, for $1 < \kappa < n$.*

PROOF. Let y be the root of T, with $|\Lambda_y| = 1$. Recall that y starts out as a P-node with all etuples as its children. If $\kappa > 1$ and at least $\kappa - 1$ etuples do not appear in any cluster, then those $\kappa - 1$ etuples must still be children of y, and y must still be a P-node with at least κ children. Therefore, by Theorem 3, every child z_i of y has $|\Lambda_{z_i}| \geq |\Lambda_y| + \kappa - 1 = \kappa$. Thus, since all etuples are descendants of y, every etuple also has $|\Lambda_e| \geq \kappa$. □

5. MEASURING PRIVACY LOSS

As the number of permissible loci $|\Lambda_e|$ becomes smaller, more information is revealed about the query-attribute of e, reducing privacy. *Equivocation* captures this measure of progress toward compromising PQP privacy.

Definition 12. Etuple e has *equivocation* $\varepsilon_e = |\Lambda_e|$.

We can compute ε_e using Algorithm 1. Since $\overline{\pi_c} \in \mathcal{P}$ if $\pi_c \in \mathcal{P}$, we can have $\varepsilon_e = 1$ only if e is the center etuple in π_c. Otherwise, $\varepsilon_e \geq 2$. When $\varepsilon_e \leq 2$, we have learned all that we can about e using clusters, and e's privacy has clearly been compromised. Most clients will not accept the privacy compromise of *any* of their etuples, so we can state:

Definition 13. The *privacy of a PQP is compromised* if $\varepsilon_e \leq 2$ for any $e \in E$.

Having $\varepsilon_e \leq 2$ for some $e \in E$ is sufficient, but not necessary, to compromise privacy. In requiring all etuples to have equivocation at least 3, we propose a notion of privacy similar to ℓ-diversity [22], where each entity must be associated with at least ℓ sensitive values. Here, these values are loci. In Section 6, we demonstrate that PQPs are insecure by showing that at least one etuple's equivocation quickly drops below 3.

5.1 Alternate and Related Metrics

We could also measure progress toward compromising privacy in terms of uncertainty about which permutation is the correct ordering π_c. Each $\pi \in \mathcal{P}$ has equal likelihood of being π_c, and each $\pi \notin \mathcal{P}$ has likelihood zero, so we can measure this uncertainty using *permutation entropy* [3].

Definition 14. The *permutation entropy* of E is $\log_2 |\mathcal{P}|$.

Permutation entropy is straightforward to compute using a PQ-tree T. Since every tree $T' \equiv T$ represents a unique permutation $\pi \in \mathcal{P}_T$, there are $|\mathcal{P}_T|$ trees equivalent to T. Let $f(y_i)$ be the number of ways to rearrange the children of node y_i in T. Every combination of valid child arrangements yields an equivalent tree, so $|\mathcal{P}_T| = f(y_1) \cdots f(y_m)$. Thus $\log_2 |\mathcal{P}_T| = \log_2 f(y_1) + \cdots + \log_2 f(y_m)$ gives the permutation entropy, which we can compute in time $O(m) = O(n)$.

If y_i is a leaf node, $f(y_i) = 1$. If y_i is a Q-node, $f(y_i) = 2$, as y_i's children can only be in forward or reverse order. If y_i is a P-node with c_i children, then $f(y_i) = c_i!$, as the children can be arbitrarily permuted.

We give experimental results measuring permutation entropy in Section 6. Permutation entropy adequately measures uncertainty about π_c, but it fails to capture the idea that $\mathcal{P}$ may give more information about some etuples than others. Thus, equivocation is generally preferable.

Another alternative is to extend Algorithm 1 to count the number of permissible permutations that assign each etuple to each of its permissible loci. Intuitively, loci deemed permissible by more permutations are more likely to be correct. We could then merge this information with knowledge of the query-attribute distribution to obtain a precise metric for the uncertainty about the query-attribute value of each etuple. Unfortunately, this modification to Algorithm 1 raises its costs to $O(n^3/\log n)$ space, and $O(n^4)$ time, leaving equivocation as a better choice when n is large. Due to space constraints, we omit details for this modification of Algorithm 1.

Work in [11] uses a metric akin to permutation entropy to analyze attacks based on repeated ciphertexts. Averaging equivocations also resembles work in [11], and counting etuples with $\varepsilon_e < \kappa$ relates to *confidential intervals* in [29].

6. EXPERIMENTS AND EVALUATION

We conducted experiments to study how quickly the privacy of Precise Query Protocols (PQPs) is compromised. In each experiment, we generate a set E of n etuples, and create an initial PQ-tree T with all n etuples as leaves of a single P node. We then generate a series of random range queries, obtain the cluster C of etuples returned by each query, and use each C to *reduce* T (see Section 3.3). We use the algorithms described in Section 4 to identify permissible loci and compute equivocations when needed. When averages are reported, they are computed by averaging results from 10 sets of queries issued on a single dataset.

We ran experiments using two datasets. In the *Random* dataset, each etuple is given a query-attribute value sampled uniformly with replacement from the domain $D = \mathbb{Z}_{10^8}$. For our real-world *Salary* dataset, we used a set of 162591 federal employee salaries [1] with values from $D = \mathbb{Z}_{373071}$. Only a tenth of the salaries are distinct. Over 600 have minimum value 0, while only one has maximum value. In practice, this property could be used to distinguish low and high-salary etuples, and thus to distinguish the correct ordering from its reverse. The most frequent salary appears over 7000 times.

The integer center of each query range is sampled from the uniform distribution $U(0, |D|)$. Integer query widths are sampled either from $U(0, |D|)$ or from a Gaussian distribution. We refer to such queries as *Uniform* and *Gaussian*, respectively. The Gaussian distribution is given by $N(10^5, 5 \times 10^4)$ for the Random dataset, and $N(2 \times 10^4, 10^4)$ for the Salary dataset. Uniform query widths tend to be large, while Gaussian widths are smaller. The Gaussian queries used for the Salary dataset are larger, relative to $|D|$, than for the Random dataset, to ensure that at least one query spans each pair of adjacent etuples. In the Salary dataset, the maximum separation between subsequent query-attribute values is nearly 25000, despite its small domain size. We use the following terms:

- *Query Count*: The number of queries issued so far.
- *Total Return Count*: The total number of etuples returned by queries issued so far.
- *Distinct Return Count*: The number of distinct etuples returned so far.
- *Privacy Compromise*: The event of at least one etuple reaching equivocation $\varepsilon_e \leq 2$, as in Definition 13. Recall that this condition is sufficient, but not necessary, for PQP privacy to be compromised.

6.1 Progress Before Privacy Compromise

Query Count and *Total Return Count* both measure query processing work done by the server. We primarily use *Total Return Count*, as it is less sensitive to the distribution of query widths. Figures 14 and 15 show equivalent results under both metrics for the Random dataset.

Figure 14 gives the average *Query Count* before *Privacy Compromise* occurs. These numbers are strikingly low. Privacy is compromised sooner for the larger, Uniform queries since large queries are more likely to intersect with others, and thus exclude more permutations.

In most cases, for any etuple to have $\varepsilon_e \leq 2$, the root of T must be a Q node, which will not happen until all etuples are linked together or *covered* by overlapping clusters. Further, the clusters must be dense enough that the intersection of at least one pair of clusters contains only one etuple. For a given query width distribution, reaching sufficient coverage requires a constant number of queries, while reaching sufficient density requires a number of

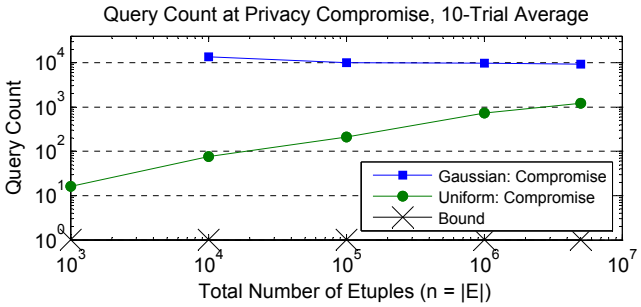

Figure 14: *Query Count* **before** *Privacy Compromise*, **Random dataset.**

Student Version of MATLAB

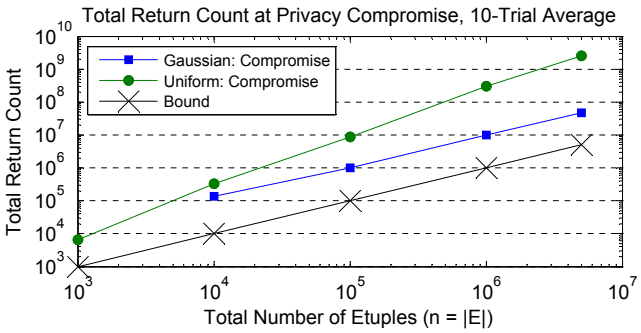

Figure 15: *Total Return Count* **before** *Privacy Compromise*, **Random dataset.**

Student Version of MATLAB

queries that increases when etuples are more closely spaced along the domain (larger n).

The larger, Uniform queries reach sufficient coverage before sufficient density, so as n increases, we need more of them in order to reach *Privacy Compromise*. However, the Gaussian queries reach sufficient density before sufficient coverage, so we need nearly the same number of them for all n. In fact, as n increases, the number of Gaussian queries needed drops slightly, as the regions where query ranges overlap are more likely to contain etuples, and thus to yield intersecting clusters.

Figure 14 shows that issuing even 100 queries is risky. In principle, even a single query can cause *Privacy Compromise* if it returns all but one etuple, as captured by curve *Bound*. All etuples returned by such a query are contiguous, so the remaining etuple e must be assigned to locus 1 or n in the correct ordering. That is, e has $\Lambda_e = \{1, n\}$, and $\varepsilon_e = 2$.

Figure 15 measures against *Total Return Count* instead of *Query Count*. It shows that for small queries, on average, *Privacy Compromise* occurs after the *Total Return Count* reaches roughly 10 times the database size ($10n$). *Privacy Compromise* is just as quick for the Salary dataset ($n = 162591$), averaging 13.2 queries or 1.017×10^6 etuples returned (Uniform), and averaging 100.9 queries or 1.019×10^6 etuples (Gaussian).

While larger queries exclude more permutations, they exclude fewer permutations than several small queries with the same total size. Thus, privacy is compromised sooner for the smaller, Gaussian queries in terms of the *Total Return Count*, since the Gaussian queries exclude more permutations *per etuple returned*. This contrasts with the fact that privacy is compromised sooner for Uniform queries in terms of *Query Count* (Figure 14).

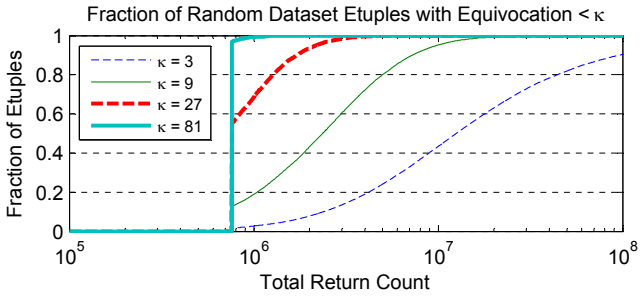

Figure 16: Random dataset, Gaussian widths: $N(10^5, 5\times10^4)$.

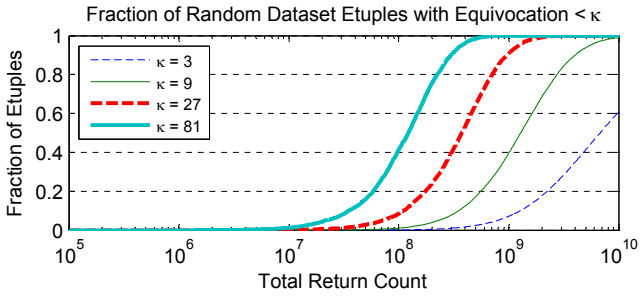

Figure 17: Random dataset, Uniform widths: $U(0, 10^8)$.

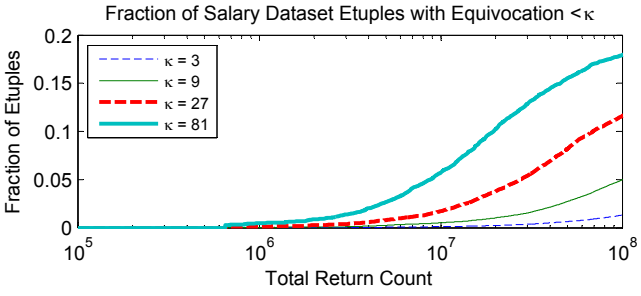

Figure 18: Salary dataset, Gaussian widths: $N(2\times10^4, 10^4)$.

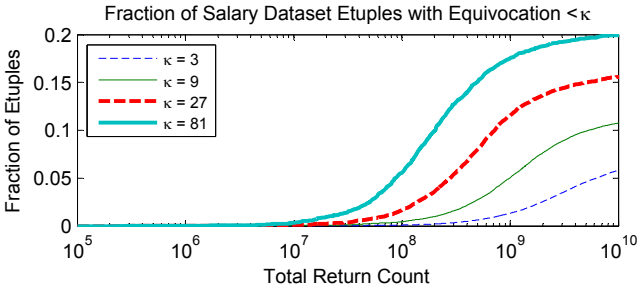

Figure 19: Salary dataset, Uniform widths: $U(0, 373071)$.

6.2 Higher Thresholds and Larger κ

Student Version of MATLAB

If value is distributed across many records, we may permit many etuples to have small equivocations. In other cases, it is unacceptable for any etuple to have even a moderate equivocation, such as when an attacker is guessing a passcode and can afford several attempts. Figures 16-19 illustrate the rate of privacy loss in such cases by plotting the fraction of etuples with equivocation $\varepsilon_e < \kappa$, for various κ.

We explain the threshold phenomenon in Figure 16 by considering the *Distinct Return Count*, which is the number of distinct etuples returned so far. If the *Distinct Return Count* is at most $(n - \kappa + 1)$, then at least $\kappa - 1$ etuples have not been returned, and thus do not appear in any cluster. By Theorem 4 (Section 4.5), every etuple $e \in E$ then has equivocation $\varepsilon_e \geq \kappa$, for $1 < \kappa < n$. Thus, no etuples have $\varepsilon_e < \kappa$ until the *Distinct Return Count* exceeds $n - k + 1$.

This threshold is reached with only a few of the large, Uniform queries, but requires many of the small, Gaussian queries. The numerous Gaussian queries exclude many permutations, such that many etuples have equivocations only slightly larger than κ. When the threshold is reached, many such etuples drop to $\varepsilon_e < \kappa$ together, yielding the phenomenon in Figure 16. With the larger, Uniform queries, fewer permutations are excluded before the threshold is reached, so the trend is more gradual (Figure 17).

This same threshold phenomenon appears for Gaussian queries in the Salary dataset (Figure 18), though it is almost undetectable since the queries are larger relative to the domain. The Salary dataset contains many indistinguishable etuples (etuples with duplicate values), so only a few etuples can ever reach low equivocations (Figures 18-19).

6.3 Permutation Entropy

Figure 20 plots permutation entropy against the *Total Return Count* for the experiments in Figures 16–19. Permutation entropy is independent of κ and captures overall progress toward identifying the correct ordering. In these experiments, permutation entropy remains high even after equivocation drops below 3 for several etuples, at which point privacy is compromised. Thus it appears that equivocation is a more reliable privacy metric. Permutation entropy stays higher for the Salary dataset because of the large number of indistinguishable etuples with duplicate query-attribute values.

Student Version of MATLAB

Permutation entropy is useful when the query distribution is so skewed that many etuples are never returned and all equivocations remain large. Privacy may still be eroding, as etuples that *are* returned become relatively ordered, allowing the attacker to make limited inferences. In such cases, permutation entropy is an effective and efficient privacy metric, whereas equivocation is expensive for large κ and gives no indication of privacy loss for small κ.

6.4 Effects of Indexes on PQP Privacy

We have shown that the privacy of any PQP can be compromised quickly as queries are issued, regardless of the PQP's encryption and querying mechanisms. In practice, *Privacy Compromise* oc-

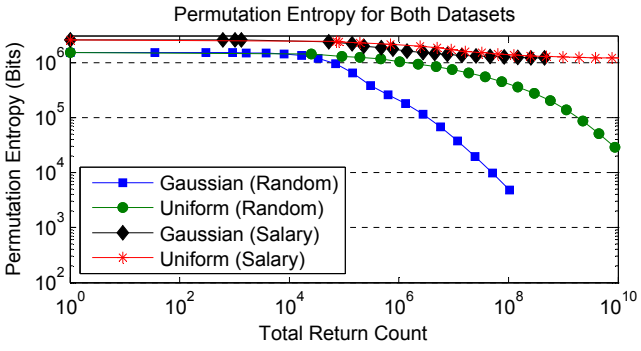

Figure 20: Drop rates of permutation entropy for Gaussian and Uniform query width distributions on the Random and Salary datasets.

Student Version of MA

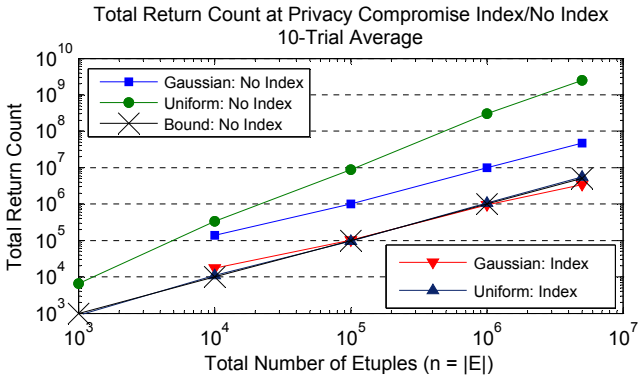

Figure 21: Effects of a binary range tree index on the average *Total Return Count* reached before *Privacy Compromise*.

curs even sooner, as most PQPs use a server-side index that leaks additional information.

For example, consider a balanced, binary range tree, with nodes that hold encrypted query-attribute ranges. The children of a single parent node hold disjoint ranges covering the parent's range. Each leaf represents a single query-attribute value, and points to all etuples with that value. The server receives encrypted query range tokens from the client and uses them to search the tree by homomorphically checking whether query and node ranges overlap, all without learning the plaintext ranges. Similar indexes can be found in [8] and [19].

In such an index, etuples descended from each node form a cluster. These clusters, along with those obtained from query result sets, can be used to reduce a PQ-tree. In Figure 21, we see that including such an index dramatically reduces the average *Total Return Count* reached before *Privacy Compromise*.

6.5 Consequences and Alternatives

The Figures above show that in a PQP that supports range queries, *Privacy Compromise* can occur quickly. Faced with this reality, clients have two options: abandon PQPs, or assume that query-attribute distributions are hidden.

6.5.1 Assuming Attribute Distributions are Hidden

The client may choose to make the dangerous assumption that the attacker will never learn the query-attribute distribution. In this case, our work shows that the client should operate under the assumption that the attacker knows the correct etuple ordering. PQPs like OPES [2] are designed for this scenario, and claim decent privacy properties [4] and excellent efficiency. Thus it is hard to see much advantage to using a more complex PQP, which will likely be less efficient (see Table 1).

6.5.2 Abandoning PQPs Altogether

If the distribution is known, the client must find alternatives to PQPs. One option is to use partitioning schemes [15, 16], which make privacy guarantees in terms of entropy and indistinguishability, at the cost of spurious results.

An alternative is to periodically re-encrypt etuples using a new key. The attacker cannot correlate old and re-encrypted etuples, so he must then restart his attack. As Figure 15 shows, we must re-encrypt *at least* one etuple for every 10–100 returned to retain even a basic level of privacy.

Re-encryption, also called *node swapping* or *shuffling*, is already used to support privacy-preserving point queries in work on oblivious index traversal techniques [13, 21], and Oblivious RAM [28]. Oblivious index traversals generally use spurious queries and a tunable constant number of re-encryptions per etuple returned in order to achieve probabilistic privacy guarantees. Existing oblivious RAM schemes require at least $\log n$ re-encryptions per etuple returned, and achieve provable access pattern indistinguishability by requesting spurious items and frequently re-encrypting recently requested items. Such techniques are promising, but have not yet been optimized for range queries.

7. CONCLUSION

We have presented an attack that can be used to infer attribute values of encrypted records in any Precise Query Protocol (PQP) that supports one-dimensional range queries. We mounted the attack using PQ-trees and a novel algorithm for identifying permissible loci. Experimental results demonstrate that our attack requires us to observe only 10^4 queries to compromise privacy for a database of over 10^6 records, indicating that PQPs are highly insecure when query-attribute distributions are known. Future research on privacy-preserving range queries should investigate efficient alternatives to PQPs.

We will explore additional privacy metrics for PQPs in future work. We are also interested in finding an analog to our attack for multi-dimensional queries, where returned etuples are contiguous along multiple attributes at once.

8. ACKNOWLEDGEMENTS

This work was supported in part by grant N00014-07-C-0311 from the Office of Naval Research and by the National Physical Science Consortium Graduate Fellowship.

9. REFERENCES

[1] Salaries of federal employees located in the District of Columbia. Available: `http://php.app.com/fed_employees11/search.php`, 2011. Source: U.S. Office of Personnel Management.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. ACM SIGMOD*, pages 563–574, 2004.

[3] C. Bandt and B. Pompe. Permutation entropy: A natural complexity measure for time series. *Phys. Rev. Lett.*, 88:174102, Apr 2002.

[4] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proc. EUROCRYPT*, pages 224–241, 2009.

[5] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proc. TCC*, pages 535–554, 2007.

[6] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13(3):335–379, 1976.

[7] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Trans. Inf. Syst. Secur.*, 8(1):119–152, 2005.

[8] K. Chen, R. Kavuluru, and S. Guo. RASP: efficient multidimensional range query on attack-resilient encrypted databases. In *Proc. ACM CODASPY*, pages 249–260, 2011.

[9] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud:

outsourcing computation without outsourcing control. In *Proc. ACM CCSW*, pages 85–90, 2009.

[10] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. *Proc. ESORICS*, pages 440–455, 2009.

[11] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. ACM CCS*, pages 93–102, 2003.

[12] J. Dautrich and C. Ravishankar. Security limitations of using secret sharing for data outsourcing. In *Proc. DBSec*, 2012.

[13] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. ICDCS*, 2011.

[14] B. Faaland. Solution of the value-independent knapsack problem by partitioning. *Operations Research*, 21(1):332–337, 1973.

[15] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. ACM SIGMOD*, pages 216–227, 2002.

[16] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, pages 1–26, 2011.

[17] M. Kantarcıoglu and C. Clifton. Security issues in querying encrypted data. In *Proc. DBSec*, pages 325–337, 2005.

[18] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proc. EUROCRYPT*, pages 146–162, 2008.

[19] J. Li and E. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *Proc. DBSec*, pages 69–83, 2005.

[20] M. Li, S. Yu, N. Cao, and W. Lou. Authorized private keyword search over encrypted personal health records in cloud computing. In *Proc. ICDCS*, 2011.

[21] P. Lin and K. Candan. Hiding tree structured data and queries from untrusted data stores. *Information Systems Security*, 14(4):10, 2005.

[22] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1):3–es, 2007.

[23] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *Proc. DBSec*, pages 89–103, 2006.

[24] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Proc. FC*, 2011.

[25] P. Samarati and S. De Capitani di Vimercati. Data protection in outsourcing scenarios: issues and directions. In *Proc. ASIACCS*, pages 1–14, 2010.

[26] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. In *Proc. TCC*, pages 457–473, 2009.

[27] E. Shi, J. Bethencourt, T.-H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proc. IEEE S&P*, pages 350–364, 2007.

[28] E. Shi, H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.

[29] J. Wang and X. Du. A secure multi-dimensional partition based index in DAS. In *Proc. APWeb*, pages 319–330, 2008.

[30] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *Proc. ESORICS*, pages 479–495, 2006.