The Strength of Weak Randomization: Efficiently Searchable Encryption with Minimal Leakage

David Pouliot, Scott Griffy, and Charles V. Wright

Portland State University {dpouliot,scog,cvwright}@cs.pdx.edu

November 10, 2017

Abstract

Efficiently searchable and easily deployable encryption schemes enable an untrusted, legacy service such as a relational database engine to perform searches over encrypted data. The ease with which such schemes can be deployed on top of existing services makes them especially appealing in operational environments where encryption is needed but it is not feasible to replace large infrastructure components like databases or document management systems. Unfortunately all previously known approaches for efficiently searchable encryption are vulnerable to inference attacks where an adversary can use knowledge of the distribution of the data to recover the plaintext with high probability.

In this paper, we present the first efficiently searchable, easily deployable database encryption scheme that is provably secure against inference attacks even when used with real, low-entropy data. Ours is also the only efficiently searchable construction that provides *any* provable security for protecting multiple related attributes (columns) in the same database. Using this ESE construction as a building block, we give an efficient construction for performing range queries over encrypted data.

We implemented our constructions in Haskell and used them to query encrypted databases of up to 10 million records. In experiments with a local Postgres database and with a Google Cloud Platform database, the response time for our encrypted queries is not excessively slower than for plaintext queries. With the use of parallel query processing, our encrypted queries can achieve similar and in some cases superior performance to queries on the plaintext.

1 Introduction

Many organizations today are moving to the cloud, shipping their critical data to servers over which they have little control. Encrypting data before uploading it into the cloud protects against theft or accidental disclosure, but standard encryption mechanisms also prevent the cloud service from performing any useful computation on the client's behalf. One of the most desirable applications for encrypted data is search.

The problem of searching on encrypted data involves inherent trade-offs between security, performance, and utility. Here utility includes the expressiveness of queries supported. For example, a construction that supports a large subset of Structured Query Language (SQL) is more useful than one that supports only exact-match queries. Another important aspect of utility is the ease or difficulty of deploying a construction for use in the real world.

Homomorphic encryption [15] and oblivious RAM [17, 34] offer very strong guarantees of security, but their practical performance is limited compared to other, more efficient technqiues like symmetric searchable encryption (SSE) [33, 16, 13, 11]. Recent SSE schemes support rich queries [26] and achieve fast performance even on very large data sets [11]. However, their lack of backwards compatibility with existing services makes them difficult to deploy in real operational environments [20].

Another parallel line of work has investigated "efficiently searchable" and "efficiently deployable" schemes that enable an untrusted server to efficiently index and search on encrypted data [6, 4, 7]. The idea is to trade off a little security in exchange for potentially faster performance and (most importantly) the ability to deploy encryption *immediately* to protect data stored on existing cloud services. This approach also frees the encryption developer from needing to worry about issues of availability, redundancy, performance, scaling, etc., because the underlying cloud service already takes care of them. The CryptDB system of Popa et al [31] first demonstrated the potential of this approach for protecting outsourced relational databases. By layering a collection of encryption techniques, including efficiently searchable and order-preserving encryption, CryptDB supports most SQL queries used by real applications. They achieved near-native performance on queries from a standard database benchmark, storing only ciphertext in a standard MySQL database. Similar efforts from industry include SAP's SEEED project [19] and Microsoft's *Always Encrypted* feature in SQL Server 2016 [1].

Unfortunately for most real data, efficiently searchable constructions like deterministic and order-preserving encryption trade off more than just a little security. Inference attacks have recently been demonstrated that enable an adversary to recover some or all of the plaintext records if it has some auxiliary information about their statistical distribution [10, 29, 32, 14, 21, 37, 38]. Given the power of these attacks, it is reasonable to question whether security is feasible at all for efficiently searchable and easily deployable encryption. Much to our surprise, it turns out that the answer is "yes," it is possible to construct efficiently searchable schemes that achieve provable security against inference attacks.

1.1 Contributions

In this paper, we present the first efficiently searchable, easily deployable database encryption scheme that is provably secure against inference attacks even when used with real, low-entropy data. Ours is also the only efficiently searchable construction that provides *any* provable security for protecting multiple related attributes (columns) in the same database. Using this ESE construction as a building block, we give an efficient construction for performing range queries over encrypted data in an untrusted SQL database. The security of our schemes is tunable with a single parameter, allowing database owners to choose the most appropriate balance of security versus runtime performance and space overhead for the demands of their individual applications.

Our core technique is a generalization of a "folklore" encryption technique that we call *weakly randomized encryption* (WRE). WRE is a sort of middle ground between deterministic encryption (DET) and conventional, strongly randomized encryption. DET enables efficient, logarithmic-time search because it allows a legacy server to create an index from only ciphertexts, but on the other hand, it provides very little security for real data. Conventional (strongly) randomized encryption prevents the adversary learning even a single bit about the plaintext [18], but in doing so, it also precludes the possibility of efficient search. In a weakly randomized encryption, only a few bits of randomness sampled from a low-entropy distribution are used in each encryption.

Our analysis shows that this is sufficient to protect against inference attacks if we choose the distribution carefully. The trade off is that, in order to perform a WRE encryption, one must know the probability distribution of the plaintexts. We believe it is not unreasonable to ask that the data owner must know his data at least as well as the attacker does.

Our constructions are compatible with standard SQL relational databases. They can be deployed immediately on popular cloud service platforms including Google Cloud SQL[1] and Amazon Relational Database Service[2]. They are efficiently scalable up to databases containing millions of records. In our experiments, we instantiated an encrypted database of 10 million records on Google Cloud SQL and queried it from a residential cable modem connection. We performed equality queries and range queries returning up to 10,000 records. Our encrypted database, including its server-generated indices, requires less than twice the space required for the plaintext DB. Query response time with our construction depends on the choice of security parameter, but with support for parallel query processing, we can handle many encrypted queries faster than the plaintext.

Our threat model assumes a passive attacker: we give the adversary access to only the encrypted data and a source of auxiliary information. We assume he does not have access to the encrypted queries, the access patterns or return results. We acknowledge that the adversary in this model is relatively weak compared to the standard adversaries for SSE or ORAM. However, we stress that attackers who can obtain offline access to the encrypted database, e.g. by SQL injection or by stealing a backup hard drive, are an important real-world threat. All previous easily deployable, efficiently searchable schemes fail to achieve even this modest level of security.

The paper is organized as follows. We review related work in Section 2. We introduce our notion of security for WRE against inference attacks in Section 3. In Section 4 we present the generic template for a weakly randomized encryption, and then we give sequentially stronger variations on this idea, leading up to our most secure construction, WRE with Poisson salt allocation. We discuss the problem of protecting multiple related database columns against inference attack in Section 5. We show how WRE can be used to support range queries in Section 6. We evaluate the performance of our new constructions experimentally with real databases in Section 7.

2 Related Work

An *efficiently searchable encryption* scheme is one that reveals some function of the plaintext in order to allow logarithmic search time using a legacy database or information retrieval system. Bellare, Boldyreva, and O'Neill [6] and Amanatidis, Boldyreva and O'Neill [4] proposed and analyzed deterministic encryption (DET) schemes that are efficiently searchable. Those schemes are provably secure only when the plaintext database has high min-entropy [6]. An "easily deployable" efficiently searchable encryption is one that retains (enough of) the expected format of the plaintext to enable use with an existing cloud service [23] [27].

Order-preserving encryption was first described in 2004 by Agrawal et al., who proposed a method to encrypt data so that the resulting ciphertexts retain the same ordering as the plaintext [3]. OPE was first studied formally by Boldyreva, Chenette, Lee, and O'Neill [7]. Boldyreva, Chenette, and O'Neill introduced the related notion of efficiently orderable encryption [8], in which

[1]`https://cloud.google.com/sql/`
[2]`https://aws.amazon.com/rds/`

a public, efficient function can be used to compare the ciphertexts. The similar notion of order-revealing encryption was proposed by Boneh et al. [9], and recent works give efficient symmetric schemes for ORE [12, 28]. The term "property-preserving encryption" is sometimes used to encapsulate both OPE/ORE and ESE [30].

Deterministic ESE constructions were shown to be insecure against inference attacks [29], [10], [32]. Even more powerful attacks have been demonstrated against OPE and ORE [29] [14] [22]. A more recent analysis shows that OPE/ORE and deterministic encryption allow the adversary to recover almost the entire plaintext database, with high probability, for almost all types of data [38].

Techniques for "bucketizing" search tokens, for example with hash collisions, to reduce information leakage have been proposed in encrypted search schemes since their inception [33, 16, 6, 27]. Another line of work also uses bucketization to enable range queries over encrypted data without the use of ORE [25, 24, 36]. While we do not employ bucketization in this work, we believe that it could be combined with our techniques to provide even stronger security.

3 Security Definitions

Our security definitions are closely modeled after the standard notion of security against a *ciphertext-only attack*, also sometimes called a *known ciphertext attack*. We extend these in a straightforward way to capture the idea that, in the cryptanalysis of an encrypted database, it is not the actual bits of the ciphertext that are useful to the adversary, but the leakage.

The first definition, *IND-LOA*, essentially states that the adversary can't learn anything from observing the leakage, i.e. that the leakage does not reveal anything about the plaintext.

The second definition, ϵ*-IND-LOA*, is weaker than the first. It allows the adversary to learn something from observing the leakage, but limits his advantage to no more than some small ϵ. Note that we do not require that ϵ be negligible. The idea is to admit schemes that fall short of the stronger definition, but that still provide some quantifiable level of security which may be sufficient for practical use in (some) real applications.

Adversarial Model. Our model assumes that an attacker obtains all the ciphertext records from an encrypted database. It also has auxiliary information including the alphabet of plaintext values $\mathcal{M}$ and the probability distribution over $\mathcal{M}$ from which the database was generated. It is allowed to make one query to an encryption oracle, which does not return the ciphertext (as in the standard ciphertext-only attack) but the leakage associated with the ciphertext. The adversary's task is then to use the observed leakage, together with his auxiliary information, to guess which plaintext was encrypted by the oracle.

Leakage. In an encrypted database, the leakage can come in a few different forms. For example, deterministic encryption reveals the number of times each record occurs in each column of the database, and order-revealing encryption also reveals the number of records greater than and less than each encrypted record. In a database table having more than one encrypted column (ie, in all real databases), the leakage also includes all information the adversary can observe about the relationship between columns. For example with DET, this includes how often each ciphertext a in column 1 occurs together with each ciphertext b in column 2, and so forth for all pairs, triples, etc. of columns in the table.

3.1 Security Game

Let db be the plaintext records of the database. We assume that the records in db are an independent and identically distributed sample from probability distribution P_M over a finite alphabet of plaintexts $\mathcal{M}$. Let edb be the encrypted records of the database. Let Aux include the plaintext values and the plaintext frequencies. Let $\mathcal{A}$ be an adversary who has access to edb and Aux. Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme with message space $\mathcal{M}$ and key space $\mathcal{K}$.

Indistinguishably Experiment $PrivK_{\mathcal{A},\Pi}$

1. Let P_M be a probability distribution over $\mathcal{M}$, and let $db \overset{\$}{\leftarrow} P_M$.
2. Let $sk \leftarrow \mathsf{Gen}(\mathcal{K})$.
3. Let $edb \leftarrow \mathsf{Enc}(sk, db)$.
4. $\mathcal{A}$ gets P_M and edb
5. $\mathcal{A}$ chooses a pair of messages $m_0, m_1 \in \mathcal{M}$
6. A uniform bit $b \in 0, 1$ is chosen. Ciphertext $c \leftarrow \mathsf{Enc}(sk, m_b)$ is computed and the leakage $\ell = \mathcal{L}(c, edb)$ is given to $\mathcal{A}$.
7. $\mathcal{A}$ outputs a bit b'
8. The output of the experiment is 1 if $b' = b$, and 0 otherwise. We write $PrivK_{\mathcal{A},\Pi} = 1$ if the output of the experiment is 1, and in this case we say that $\mathcal{A}$ succeeds.

Definition 1 (Indistinguishability under Leakage-Only Attack (IND-LOA))**.** *We say that the encryption scheme* Π *with security parameters* λ *and* k *has IND-LOA security if, for all probabilistic polynomial time adversaries* $\mathcal{A}$*,*

$$Pr[PrivK_{\mathcal{A},\Pi}] \leq \frac{1}{2} + \mathsf{negl}(k) + negl(\lambda).$$

Definition 2 (ϵ-Indistinguishability under Leakage-Only Attack (ϵ-IND-LOA))**.** *We say that the encryption scheme* Π *has* ϵ*-IND-LOA security if, for all probabilistic polynomial time adversaries* $\mathcal{A}$*,*

$$Pr[PrivK_{\mathcal{A},\Pi}] \leq \frac{1}{2} + \epsilon$$

Definition 3 (Statistical Distance)**.** *The* statistical distance Δ *between two random variables* X, Y *over a common domain* ω *is defined as:*

$$\Delta(X, Y) = \frac{1}{2} \sum_{\alpha \in \omega} \left| Pr(X = \alpha) - Pr(Y = \alpha) \right|$$

Two random variables X, Y are said to be ϵ-close if the statistical distance between them is at most ϵ. Variables X, Y are called *statistically indistinguishable* if $\epsilon = negl(\alpha)$ with security parameter α.

Theorem 3.1 (ϵ-IND-LOA) Bound). *The adversary's success in distinguishing m_0 and m_1 in the IND-LOA game is bounded by the statistical distance of the leakage for the two plaintexts. Let M be the random variable for the plaintext chosen by the oracle, and let $\mathcal{L}$ be the leakage associated with its ciphertext. Then the encryption is ϵ-IND-LOA secure with*

$$\epsilon = \frac{1}{2}\Delta\big(Pr(\mathcal{L}|M = m_0), Pr(\mathcal{L}|M = m_1)\big)$$

Note that Theorem 3.1 holds even for encryptions that allow the adversary to guess with nearly-perfect accuracy, i.e. $\epsilon \approx \frac{1}{2}$. Examples of such schemes include DET and OPE [38]. The goal in this work is to show the existence of efficiently deployable schemes that give an ϵ-IND-LOA bound with ϵ much closer to zero.

4 Weakly Randomized Encryption

In this section we formalize and extend a "folklore" technique that we call *weakly randomized encryption* (WRE). The idea is that we can reduce the vulnerability of deterministic encryption to frequency analysis and other leakage abuse attacks by adding a small amount of randomness to the encryption. In Algorithm 1 we show how the WRE encryption, decryption, and search functions can be constructed from a DET scheme with encryption function E and decryption function D.

The WRE encryption takes as input: a symmetric key K; a plaintext m; the probability distribution P_M of the plaintexts; and a deterministic encryption E that leaks nothing except equality of the plaintexts.

Next the encryption algorithm calls the *getSalts* subroutine to pseudorandomly generate a probability distribution P_S over a set of salts S. The *getSalts* subroutine can use knowledge of the plaintext distribution to choose a salt distribution that makes the ciphertexts close to uniform. (We give a handful of candidate algorithms for *getSalts* below, and we evaluate their security in the following section.) A salt $s \in S$ is chosen at random according to P_S and is pre-pended to the message. Finally, the salt and plaintext are encrypted with the Deterministic encryption algorithm.

Decrypting simply uses the deterministic decryption D to recover the salt and plaintext, then discards the salt, and returns the plaintext.

To search, the search term is encrypted with all the possible salts creating a query that searches for all of these encrypted keywords separated by *or* clauses. WRE retains the sub-linear search time of deterministic encryption, with a linear (in n where n is the number of salts) increase in overhead. To retrieve all records equal to m, the client first computes all possible encryptions of m and then requests all encrypted records equal to c_1 or c_2 ... or c_n. Because the number of unique ciphertexts for each plaintext is small, WRE allows the server to build useful indexes on the encrypted data, just as with DET. To perform the search for each c_n, the server simply consults its index and returns the list of matching records.

An alternative construction uses a keyed PRF instead of a deterministic encryption algorithm. Since the PRF is assumed to be one way, the output of the PRF is only used for creating search tokens and the plaintext is also encrypted with a randomized encryption algorithm. Decrypting ignores the search tokens and simply decrypts the ciphertext.

The WRE-2 encryption takes as input: a symmetric key K; a plaintext m; the probability distribution P_M of the plaintexts; a randomized encryption RE that leaks nothing; and a PRF that leaks nothing except equality. The first steps in encryption are the same. The difference is a

Algorithm 1 WRE-1

```
1:  function $\mathcal{E}(K, m, P_M)$
2:      $k_1, k_2 \leftarrow Expand(K)$
3:      $S, P_S \leftarrow getSalts(P_M, m, k_1)$
4:      $s \xleftarrow{\$} P_S$
5:      $c \leftarrow E(k_2, s||m)$ return $c$
6:  function $\mathcal{D}(K, c)$
7:      $k_1, k_2 \leftarrow Expand(K)$
8:      $s||m \leftarrow D(k_2, c)$ return $m$
9:  function $Search(K, m, P_M)$
10:     $k_1, k_2 \leftarrow Expand(K)$
11:     $S, P_S \leftarrow getSalts(P_M, m, k_1)$
12:     $l \leftarrow |S|$
13:     $for\ i = 1, .., l\ do$
14:         $c_i = E(k_2, S_i \mid\mid m)$
15:     return $Query(c_1\ or\ c_2 ..\ or\ c_l)$
```

search tag is created using the PRF and salt, and the plaintext is encrypted with the randomized encryption algorithm. The search algorithm uses the PRF to obtain all possible search tags instead of the encryption algorithm.

The improvement in security, if any, of WRE over deterministic encryption is not immediately clear. We show in the following section that the security of weakly randomized encryption against inference attack is completely dependent on (i) the number of salts allocated to each plaintext and (ii) the probability distribution from which the salts are sampled. Surprisingly, our analysis also shows that, with a carefully chosen *getSalts* algorithm, we can construct a weakly randomized encryption that leaks virtually no information about the plaintext when the distribution P_M is known. In Sections 4.1 and 4.2 below, we begin by introducing some simple "strawman" salt algorithms. After that, we provide algorithms with increasing levels of provable security, Remainder salts, Uniform Random Salts, and Poisson Random Salts.

4.1 Fixed Salts Method

We refer to the "folklore" version of weakly randomized encryption as the "fixed salts" method, because it always uses a constant number of salts for every plaintext, regardless of the frequency of the plaintext. We label the security parameter of this scheme as N, the number of unique salts per plaintext.

Security. If a plaintext m occurs in the unencrypted database with frequency p, then with fixed salts, each of m's N ciphertexts will occur in the EDB with frequency $\frac{p}{N}$. Intuitively, the fixed salt method improves on the security of deterministic encryption because it reduces the differences in the plaintext frequencies.

Using the analysis of from Baigneres, Junod, and Vaudenay (Thm. 6 in [5]), we could show that the fixed salts method increases the number of records that can be safely stored in the EDB, because it decreases the difference in frequency of the observed events (that is, the ciphertexts). However, here we do not give a full analysis of the fixed salt method's security, because we have

Algorithm 2 WRE-2

```
function $\mathcal{E}(K, m, P_M, E, F)$
    $k_1, k_2, k_3 \leftarrow Expand(K)$
    $S, P_S \leftarrow getSalts(P_M, m, k_1)$
    $s \xleftarrow{\$} P_S$
    $t \leftarrow F(k_2, s||m)$
    $c \leftarrow E(k_3, m)$ return $c, t$
function $\mathcal{D}(K, c, D)$
    $k_1, k_2, k_3 \leftarrow Expand(K)$
    $m \leftarrow D(k_3, c)$ return $m$
function $Search(K, m, P_M, F)$
    $k_1, k_2, k_3 \leftarrow Expand(K)$
    $S, P_S \leftarrow getSalts(P_M, m, k_1)$
    $l \leftarrow |S|$
    $for\ i = 1, .., l\ do$
        $t_i = F(k_2, S_i \mid\mid m)$
    return $Query(t_1\ or\ t_2 ..\ or\ t_l)$
```

Algorithm 3 Fixed Salts Method

```
function GetSalts-Fixed($P_M$, $m$, $k$)
    $S \leftarrow [1, N]$
    $P_S \leftarrow DiscreteUniform(S)$
    return $S, P_S$
```

limited space, and because it suffers from a few important limitations.

Limitations. First, the overall improvement to security is small. For large databases, the adversary can still guess the plaintext with very high accuracy. Second, the fixed salt WRE is not very efficient. In order to achieve any reasonable security for a database of moderate size, it needs a large number of salts, making query processing unnecessarily intensive, especially for low-frequency plaintexts. We could potentially improve both of these aspects if we used fewer salts for low-frequency plaintexts. We formalize this idea in the next section.

4.2 Proportional Salts Method

The fixed salts method can be improved by taking into account the frequencies of the plaintexts in the database. Intuitively, we would like each ciphertext to occur with roughly the same frequency, regardless of the plaintext. In the proportional salts method, we allocate a different number of salts to each plaintext, in proportion to its frequency in the plaintext data. Let the total number of ciphertexts be N_T. Then for a plaintext m with frequency $P_M(m)$, we use $N_m \approx P_M(m) \cdot N_T$ salts.

Unlike the fixed salts method, proportional salt allocation requires that the data owner must know the plaintext distribution P_M in order to encrypt a message. It also requires some extra work to calculate the number of salts needed for each plaintext. These downsides are offset by the increase in security.

Algorithm 4 Proportional Salt Allocation

```
1: function GetSalts-Proportional($P_M$, $m$, $k$)
2:     $N_m \leftarrow P_M(m) \cdot N_T$
3:     $S \leftarrow [1, N_m]$
4:     $P_S \leftarrow DiscreteUniform(S)$
5:     return $S, P_S$
```

Security. With proportional salt allocation, for any two plaintexts $m_0, m_1 \in \mathcal{M}$, their ciphertexts will appear in the EDB with approximately the same frequency:

$$\frac{P_M(m_0)}{N_{m_0}} \approx \frac{P_M(m_1)}{N_{m_1}}$$
$$\frac{P_M(m_0)}{N_T P_M(m_0)} \approx \frac{P_M(m_1)}{N_T P_M(m_1)}$$
$$\frac{\sim 1}{N_T} \approx \frac{\sim 1}{N_T}$$

Limitations. One limitation of proportional salts stems from the fact that we must allocate an integer number of salts for each plaintext. This gives rise to an aliasing problem, where in certain situations using more salts can actually reduce the security. For example, consider an example database column with $P_M(m_1) = 0.7$ and $P_M(m_2) = 0.3$. If we encrypt this database with $N_T = 10$, we will end up with 7 salts for plaintext m_1 and 3 salts for plaintext m_2. This gives us perfect security: every ciphertext has frequency 0.1, independent of the particular plaintext.

However, if we increase the total number of salts to 12, then we will have 8 ciphertexts for plaintext m_1, each with frequency 0.0875, and 4 ciphertexts for plaintext m_2, each with frequency 0.075. Given sufficiently many encrypted records, the adversary will be able to distinguish the plaintexts.

In general, it may not be possible to find a single value of N_T that makes all plaintexts indistinguishable without requiring an impractical number of ciphertexts. In an extreme case, we could allocate more ciphertexts than we expect to have records in the database, ie set $N_T \geq n$, but this is almost certainly not practical unless the database is very small.

In the following sections we address this problem in multiple ways. With Remainder Salts (§4.3), we remove the leakage for most of the plaintexts in the EDB, at the cost of allowing the adversary to recover some small fraction of plaintexts. With Random Salts (§4.4.1), rather than choose salts from a uniform distribution, salts are chosen from a non-uniform distribution causing the frequencies of all ciphertexts to appear within a random range. With the Poisson Salt technique (§4.4.2), we create similar non-uniform salt distributions, but do so with Poisson processes to improve security.

4.3 Remainder Salts

With the Remainder Salts method, the idea is to encrypt as much of the database as possible with ciphertexts that have statistically identical frequencies.

This extension biases our random number generator to choose a remainder salt less often than other salts, allowing these other salts to be chosen with any weight, allowing their frequency to be completely arbitrary values. This allows us to set the frequencies of all ciphertexts that are not encrypted with a remainder salt to the exact uniform value, removing all frequency analysis. Ciphertexts that are encrypted with a remainder salt are much easier to distinguish, but because there's only one remainder salt for each unique plaintext, we leak less information for the same number of salts compared to proportional salts.

Because we can set the average frequency of non-remainder salt ciphertexts to any value, we set them all to the same value. This shared frequency will be equal to $1/N_T$, where N_T is the total number of non-remainder salts.

Algorithm 5 Remainder Salt Method

```
function GetSalts-Remainder($P_M$, $m$, $k$)
    $I \leftarrow P_M(m) \cdot N_T$
    $N_r \leftarrow \lfloor I \rfloor$
    $N_m \leftarrow N_r + 1$
    $S \leftarrow [1, N_m]$
    for each $s$ in $S$ do
        if $s < N_m$ then
            $P_S(s) = 1/N_T$
        else
            $P_S(s) = (1/N_T) * (I - N_r)$
    return $S, P_S$
```

Limitations. The remainder salts method leaks the last ciphertext (remainder salt), which later methods will fix.

4.4 Random Salt Frequencies

In our final WRE technique, the distribution of the salts or each plaintext m is generated pseudorandomly from a secret key and m. The number of salts allocated to each plaintext is also randomized.

4.4.1 Uniform Random Frequencies

In the first version of this technique (Algorithm 6), we choose the frequency for each salt from a random number, uniformly distributed on the range between zero and some maximum, p_{max}. For a plaintext m with frequency $P_M(m)$ in the plaintext distribution, we allocate salts to m until the sum of the ciphertext frequencies is within p_{max} of $P_M(m)$. At that point, we allocate one final salt and assign it whatever remaining probability mass is left.

This ensures that all ciphertext frequencies fall within the same range, $[0, p_{max}]$, regardless of the plaintext. Moreover, except for the last ciphertext allocated to each plaintext, all ciphertext frequencies, for all plaintexts, are drawn from the same Uniform distribution. Therefore the adversary learns nothing about the plaintext by observing the ciphertext frequency.

However, because the frequencies for the final salt for each plaintext are not drawn from the same distribution as the rest, it is possible that the adversary might be able to leverage this difference to learn something about the plaintext. Next we show how we can remove this limitation by sampling from a Poisson process to generate our salt frequencies.

Algorithm 6 Uniform Random Salt Frequencies

```
1: function GetSalts-Uniform($P_M$, $m$, $k$)
2:     $s = 1$
3:     $total = 0$
4:     $U = Uniform(0, p_{max})$
5:     $R = CSPRNG(k, m)$
6:     while $total < P_M(m) - p_{max}$ do
7:         $weight[s] \leftarrow Sample(U, R)$
8:         $total \leftarrow total + weight[s]$
9:         $s \leftarrow s + 1$
10:    $weight[s] \leftarrow P_M(m) - total$
11:    $N_m = s$
12:    $S = [1, N_m]$
13:    for $s \in S$ do
14:        $P_S(s) \leftarrow \frac{weight[s]}{total}$
15:    return $S, P_S$
```

4.4.2 Poisson Random Frequencies

A Poisson process is a simple stochastic process often used to model the arrival of events in a system, for example the occurrence of earthquakes in a geographical region, or the arrival of buses at a bus stop. In a Poisson process with rate parameter λ, the times between arrival events, called the "interarrival times," are independent and identically distributed, and they follow an Exponential distribution with parameter λ. The number of arrivals in an interval of length t is independent of the events in all intervals before and after, and it is Poisson distributed with expected value λt.

In the Poisson version of WRE, we have one global parameter, the base rate λ. On expectation, this method will generate about $\lambda + |\mathcal{M}|$ ciphertexts in total across all plaintexts. To allocate salts for plaintext $m \in \mathcal{M}$ and to assign their relative weights, we sample arrivals in the interval $[0, P_M(m)]$ from a Poisson process with rate λ. Let the number of arrival events in the interval be N, and let their times be denoted $a_1, \ldots, a_N$. Additionally, we define $a_0 = 0$ and $a_{N+1} = P_M(m)$. The interarrival times are $x_i = a_i - a_{i-1}$ for $i \in 1, \ldots, N+1$.

Based on the outcome of this experiment, we allocate $N + 1$ salts to plaintext m, and when we encrypt m, we choose salt i with probability $\frac{x_i}{P_M(m)}$. The resulting ciphertext will then have frequency equal to x_i in the encrypted database. Also note that N has a Poisson disribution, thus on average we will allocate about $\lambda \cdot P_M(m) + 1$ salts to plaintext m.

In the end, the Poisson approach looks very similar to the Uniform random method, but instead of sampling from a Uniform distribution to choose the weight of each salt, we sample instead from an Exponential distribution. The pseudocode for our Poisson method's algorithm is shown below in Algorithm 7.

Security. Our analysis shows how the unique properties of the Poisson process make it ideally suited for use in weakly randomized encryption. Most critically, the Poisson process guarantees that, subject to one constraint on λ, all ciphertext frequencies for all plaintexts are pseudorandom samples from indistinguishable Exponential distributions. Therefore a computationally bounded adversary learns nothing about the plaintext from the frequencies of the ciphertexts.

Algorithm 7 Poisson Salt Distributions

```
1: function GetSalts-Poisson($P_M$, $m$, $k$)
2:     $s = 0$
3:     $E = Exponential(\lambda)$
4:     $R = CSPRNG(k, m)$
5:     $lastArrival = 0$
6:     $total = 0$
7:     $prevTotal = 0$
8:     while $total < P_M(m)$ do
9:         $s = s + 1$
10:        $weight[s] \leftarrow Sample(E, R)$
11:        $prevTotal = total$
12:        $total = total + weight[s]$
13:    $total = P_M(m)$
14:    $weight[s] \leftarrow total - prevTotal$
15:    $N_m = s$
16:    $S = [1, N_m]$
17:    for $s \in S$ do
18:        $P_S(s) \leftarrow \frac{weight[s]}{total}$
19:    return $S, P_S$
```

One technical limitation of the Poisson approach is that the frequency of the first salt for each plaintext is not drawn from the same Exponential distribution as the others. To see why this is so, notice that the Poisson process may generate zero arrival events in the interval $[0, P_M(m)]$. This occurs whenever the first arrival time from the Poisson process occurs after the end of the interval; in other words, when the first interarrival time is greater than $P_M(m)$. Then we have only a single salt, and hence a single ciphertext that appears in the encrypted database with the same probability as the plaintext, $P_M(m)$. Therefore the distribution of the first ciphertext frequency is not in fact an Exponential; all the probability mass that the Exponential would assign to values greater than $P_M(m)$ is instead lumped onto the point $P_M(m)$. We call this distribution a "capped Exponential" with parameters λ and $\tau = P_M(m)$. Figure 1 illustrates the difference between the capped and regular Exponential distributions.

Recall from Theorem 3.1 that the adversary's chance of success in the IND-LOA game is bounded by the statistical distance between the distribution for his observed leakage given plaintext m_0 and the leakage distribution given m_1. The statistical distance between the standard Exponential(λ) and the capped Exponential with λ and τ is defined as the difference in their probabilities. This is equal to the area under the Exponential curve at τ. From the definition of the Exponential distribution, this quantity is

$$Pr(X > \tau) = e^{-\lambda\tau}$$

By increasing λ relative to $P_M(m)$, we can make this statistical distance arbitrarily small. We are now ready to state our first result.

Theorem 4.1 (Single-Column Security for Poisson WRE)**.** *For a single database column with*

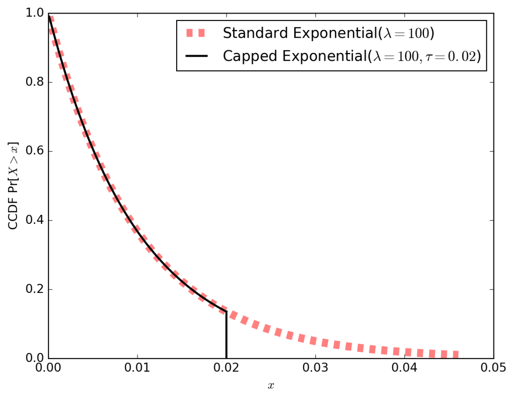

Figure 1: Complementary cumulative distribution for capped versus standard Exponentials

distribution P_M over the set of plaintexts $\mathcal{M}$, the weakly randomized encryption scheme with Poisson salt frequencies is IND-LOA secure.

Proof Sketch. The adversary can achieve success rate of 1/2 by random guessing. Analysis of the ciphertext frequencies gives him an advantage bounded by the statistical distance between the capped and standard Exponential distributions, which is $\mathsf{negl}(\lambda)$. By attempting to break the CSPRNG, he gains an additional advantage at most $\mathsf{negl}(k)$. All together, his chance of success in the game is

$$Pr[PrivK_{\mathcal{A},\Pi}] = \frac{1}{2} + \mathsf{negl}(\lambda) + \mathsf{negl}(k)$$

as desired.

5 Security for Multiple Columns

So far our solutions provide security against frequency analysis for single columns only. A harder challenge arises from cross column attacks, or those attacks using co-occurrence frequencies. If column 1 contains plaintexts $m1_i$ with $i \in \{1..n\}$ and and column 2 contains plaintexts $m2_j$ with $j \in \{1..q\}$, then a co-occurrence from column 1 and 2 is $(m1_i, m2_j)$. The frequency of this co-occurrence is how often they appear together.

This attack uses the relationships of data across columns. For example, a database column containing income might have a uniform distribution of numbers. If the table had a second column of education level, it is easy to see that there probably a high correlation between the data from the income column eduction column. While the data in the income column is uniformly distributed and by itself is difficult to attack, using the data in the Education column and the relationship between data of the two columns might be enough for an attacker to learn the contents from the uniformly distributed income column.

Thus preventing co-occurrence attacks requires that our ciphertexts not only achieve indistinguishablity in their column, but also across columns with their co-occurrence frequencies.

5.1 Encrypting Columns Independently

The simplest approach for handling multiple columns in a database table is to encrypt each column independently. This hides all frequency information for each column in isolation, but leaves open the possibility that the adversary could analyze ciphertext co-occurrence frequencies to learn about the plaintext.

Theorem 5.1. *Suppose a Poisson weakly randomized encryption scheme* Π *with parameter* λ *is used to encrypt a database of* d *columns and* n *rows with plaintext distribution* P_M*. Let* $f(m_{a0}, m_{b0}, \ldots, m_{k0})$ *be the smallest non-zero frequency of a plaintext tuple for* k *plaintexts* $(m_{a0}, m_{b0}, \ldots, m_{k0})$ *and let* $f(m_{a1}, m_{b1}, \ldots, m_{k1})$ *be the largest frequency for some* k*-tuple of plaintexts* $(m_{a1}, m_{b1}, \ldots, m_{k1})$ *over the same subset of columns. Let* $0 < \tau < 1$*, let*

$$p_0 = f(m_{a0}, m_{b0}, \ldots, m_{k0}) \cdot \tau^k$$

and let

$$p_1 = f(m_{a1}, m_{b1}, \ldots, m_{k1}) \cdot \tau^k.$$

If Π *is IND-LOA secure for each column, then it is also* ϵ*-IND-LOA secure against attacks that leverage any subset of the* d *columns, with*

$$\epsilon = \Delta(Binom(n, p_0), Binom(n, p_1))$$

with probability $1 - e^{-\lambda\tau}$*.*

The security bound from Theorem 5.1 depends strongly on the distribution P_M of the plaintexts. In the worst case scenario for an arbitrary dataset, the statistical distance could be very large (ie, close to 1), so the scheme would provide essentially no real security unless λ is very large and τ is very small.

However with many real world datasets, the epsilon may be reasonable. For example, we computed the bound from Theorem 5.1 for the Texas Health Care Data set [2] used in the NKW attacks [29]. Table 1 shows the statistical distance that we get for the data distributions from the Texas data for databases from 10,000 up to 10 million records, with τ from 0.01 to 0.00001.

τ	Size of Database			
	10,000	100,000	1,000,000	10,000,000
0.001	0.010501	0.095620	0.500062	0.500062
0.0001	0.000383	0.003814	0.036854	0.267487
0.00001	0.000024	0.000236	0.002358	0.023080

Table 1: Statistical Distance Examples from Texas Health Care Data

While these values are not negligible, some of them allow the adversary an advantage of less than 0.001. Although this is a much weaker guarantee than is typically used for cryptographic security, e.g. for a block cipher, it is many times stronger than the security that DET or ORE can achieve for even a single column.

5.2 Multicolumn Poisson Salt Allocation

Since multicolumn security is case by case, depending on the actual dataset, the database size and τ parameter, we looked for a stronger algorithm that achieves multicolumn security for any dataset. We achieved this by treating the co-occurrence of plaintexts as one plaintext and by requiring that each co-occurrence plaintext use a unique set of salts. That is, if a salt occurs in set_i, it does not occur in any other salt set.

Our method to accomplish the uniqueness of each set of salts is to take the value of the PRF for the plaintext row and prepend it to the chosen salt. This prevents us from having to either store or calculate the appropriate salts to eliminate duplicate use.

Algorithm 8 Poisson Multicolumn Encryption

```
1: function $\mathcal{E}(P_M, (m_1, m_2, ...m_i), F, E, K)$
2:     $k_1, k_2, k_3 \leftarrow Expand(K)$
3:     $S, P_S \leftarrow getSalts(P_M, m(m_1, m_2, ...m_i), k_1)$
4:     $s \xleftarrow{\$} P_S$
5:     $h \leftarrow F(k_1, (m_1, m_2, ...m_i))$
6:     $t \leftarrow F(k_2, h||s||m_1), F(k_2, h||s||m_2),$
7:             $..., F(k_2, h||s||m_i)$
8:     $c \leftarrow E(k_3, m_1), E(k_3, m_2), ..., E(k_3, m_i)$
9: return $c, t$
```

Theorem 5.2. *Let $(m_a, m_b, ..., m_z)$ be a plaintext tuple representing one record with z columns. If each plaintext tuple is assigned a Poisson Salt Distribution with λ parameter that achieves IND-LOA, and each set of salts for each plaintext tuple is unique, then each ciphertext tuple $(c_a, c_b, ..., c_z)$ and every sub co-occurrence of $(c_a, c_b, ..., c_z)$, is IND-LOA.*

Proof. We already know that the ciphertext tuple $(c_a, c_b, ..., c_z)$ has perfect secrecy by construction from our single column Theorem 4.1.

Since each ciphertext tuple gets a unique set of salts, then $(c_a, c_b, ..., c_z)$ will have the same frequency as any sub co-occurrence of $(c_a, c_b, ..., c_z)$. Since the frequencies of $(c_a, c_b, ..., c_z)$ are indistinguishable and are the same as the frequencies of its sub co-occurrences, the frequencies of all its sub co-occurrences are indistinguishable. □

The downside to using this technique is in order to achieve $IND-LOA$ security for multiple columns, we must set the security parameter λ large enough such that $\frac{1}{\lambda}$ is smaller than the frequency of the least frequent plaintext record. This means we will require more salts overall; therefore our queries must be more complex, and the database must do more work to process them. Also, when the number of columns is large and the set of possible plaintexts in each column is not small, even just computing the total number of salts for a given plaintext in a given column may be computationally intensive.

However for many datasets, this technique is still viable in practice. For example, looking at the Texas Health care database, there are several columns that contain only a small set of plaintexts; Table 2 gives some examples. Many of these are highly correlated with other columns. For such low-entropy datasets, this approach gives a large increase in security versus encrypting each column independntly, and we can calculate the entire set of salts with a relatively small amount of computation.

Column	Unique Plaintexts
Sex_Code	4
Race	6
Ethnicity	3
Spec_Unit	13
Pat_State	8

Table 2: Texas Health Care Columns

6 Efficient Range Queries with WRE

In this section we propose a scheme for range queries using our multi-column secure WRE from Section 5. The construction is practical, and it is easily deployed with existing databases with practical space requirements and reasonable performance. Unlike ORE, our scheme does not leak the order of the plaintexts.

The primary concept for this scheme involves breaking apart, or *partitioning*, each plaintext number into multiple *units*, and then encrypting each unit with WRE. Comparisons on the encrypted ciphertext units then enable range queries and similar queries provided by order revealing encryption.

For example, partitioning the number 64 into two base-10 units U_1 and U_2 results in the pair $(U_1 = 6, U_2 = 4)$. Partitioning 64 into two base-10 units gives $(U_1 = 4, U_2 = 0)$. Table 3 shows how we can construct a plaintext range query over the original space of two-digit numbers by combining equality queries over the two new units U_1 and U_2.

$$\begin{array}{c}
\big(\quad (U_1 = 1)\ \text{AND}\ \ (U_2 = 8)\quad\big)\\
\text{OR}\\
\big(\quad (U_1 = 1)\ \text{AND}\ \ (U_2 = 9)\quad\big)\\
\text{OR}\\
(U_1 = 2)\\
\text{OR}\\
\big(\quad (U_1 = 3)\ \text{AND}\ \ (U_2 = 0)\quad\big)\\
\text{OR}\\
\big(\quad (U_1 = 3)\ \text{AND}\ \ (U_2 = 1)\quad\big)
\end{array}$$

Table 3: Plaintext Range Query: 18 to 31
2 Units, Base 10, no Salts

To perform the same range query over the encrypted database, we simply replace the equality clause for each plaintext value with a clause that represents all the possible ciphertexts for the given plaintext. Table 4 shows the structure of the encrypted version of the range query from Table 3, where each plaintext gets two salts, s_1 and s_2.

$$
\begin{array}{lllll}
\Big(& (& (U_1 = E(s_1||1)) \text{ or } (U_1 = E(s_2||1)) &) & \text{AND} \\
 & (& (U_2 = E(s_1||8)) \text{ or } (U_2 = E(s_2||8)) &) & \Big) \\
 & & \text{OR} & & \\
\Big(& (& (U_1 = E(s_1||1)) \text{ or } (U_1 = E(s_2||1)) &) & \text{AND} \\
 & (& (U_2 = E(s_1||9)) \text{ or } (U_2 = E(s_2||9)) &) & \Big) \\
 & & \text{OR} & & \\
\Big(& (& (U_1 = E(s_1||2)) \text{ or } (U_1 = E(s_2||2)) &) & \\
 & & \text{OR} & & \\
\Big(& (& (U_1 = E(s_1||3)) \text{ or } (U_1 = E(s_2||3)) &) & \text{AND} \\
 & (& (U_2 = E(s_1||0)) \text{ or } (U_2 = E(s_2||0)) &) & \Big) \\
 & & \text{OR} & & \\
\Big(& (& (U_1 = E(s_1||3)) \text{ or } (U_1 = E(s_2||3)) &) & \text{AND} \\
 & (& (U_2 = E(s_1||1)) \text{ or } (U_2 = E(s_2||1)) &) & \Big)
\end{array}
$$

Table 4: Encrypted Range Query: 18 to 31
2 Units, Base 10, 2 Salts

The encryption function, shown in Algorithm 9 takes as input the message m, a WRE encryption function E, and the base b for partitioning. To encrypt, the message m is split into a list of n base-b units. Each unit is stored as a column in the encrypted database, so the DBMS can create indexes to enable fast search on each of the units. Each unit is encrypted with the WRE scheme and inserted into the database.

Decrypting reverses the process. First we collect the columns that comprise the units of the original plaintext column, then we decrypt each unit, and finally we append the decrypted plaintexts together.

Algorithm 9 Range-Query-WRE

```
function $\mathcal{E}_K(m, E, b)$
    $u_1, \ldots, u_n \leftarrow divide(m, b)$
    for $i = 1, \ldots, n$ do
        $S, P_S \leftarrow getSalts(P_{U_i}, u_i, k)$
        $s_i \xleftarrow{\$} P_S$
        $c_i \leftarrow E(s_i||u_i)$
    $pad(c)$
return $c$
function $\mathcal{D}_K((c_1, \ldots, c_n), D, b)$
    $unpad(c)$
    $m \leftarrow \emptyset$
    for $i = 1, \ldots, n$ do
        $s_i||m_i \leftarrow D(c_i)$
        $m \leftarrow m||m_i$
    return $m$
```

The process for transforming a range query over the single plaintext column into a query over the new "unit" ciphertext columns is described more formally in Algorithm 10 in Appendix B. The WRE range query is comprised of a set of equality queries separated by conjunctions and disjunctions. If our logarithmic base for the partitioning is b and we have n units per plaintext, then a naive implementation of the query algorithm would in a worst case scenario perform b^n equality queries. However searching on all units is not necessary. Our algorithm has a worst case complexity of $2bn + b$. The key insight is that we only need to get all possible unit combinations on the highest and lowest unit values for each column. For example, Table 3 and Table 4 illustrate a query for the range $[18, 31]$ using base $b = 10$ and $n = 2$ units.

Security. The WRE range query scheme encrypts the database column for each "unit" independently. Therefore its security is exactly the same as the approach for encrypting multiple columns discussed in Section 5.1. As our data analysis in that section showed, the adversary's ability to distinguish plaintexts depends heavily on the distribution of the data. In the real world, relationships between the columns are often weak enough that the adversary's advantage can be kept below an acceptable threshold by increasing the security parameter of the WRE for each column.

For an example specific to range queries, we encrypted the SPARTA [35] datasets (see §7), and we computed the statistical distance for all of the range query columns. Figure 5 shows the worst-case statistical distance for SPARTA databases from 10,000 up to 10 million records, with the partition base $b = 16$ and τ parameters from 0.01 to 0.00001.

τ	Size of Database			
	10,000	100,000	1,000,000	10,000,000
0.001	0.003329	0.032200	0.234379	0.626847
0.0001	0.000033	0.000334	0.003329	0.032200
0.00001	3.341e-07	0.000003	0.000033	0.000334

Table 5: Statistical distance for range query columns in SPARTA data.

Clearly, if a data owner uses too small of a λ to encrypt too large of a database, then the adversary can use the relationships between the unit columns to guess many of the plaintexts. For example, with $\lambda = 1000$ and $n = 10$ million, by Theorem 3.1 we allow the adversary to achieve a success rate of $\frac{1}{2} + \frac{1}{2} \cdot 0.63$, or about 81%. On the other hand, for a database of 100 thousand records, the same security parameter limits the adversary to less than 52%.

We do not mean to claim that any of the statistical distances in Table 5 are "secure." What constitutes an acceptable level of security can vary greatly from application to application. In some cases, limiting the adversary to 52% is sufficient, while in others, allowing 50.001% could have serious negative consequences.

Unlike earlier schemes, we give the data owner the ability to predict the adversary's success rate and make an informed decision. Before encrypting a given database, the data owner can also run this analysis for several possible candidate bases, e.g. base-8, base-10, base-16. Then they can pick the base that yields a set of unit columns whose frequencies leak the least.

7 Performance Evaluation

We implemented several flavors of weakly randomized encryption, including the fixed salts method, uniform random frequencies, and Poisson salt allocation in the Haskell programming language.

We also implemented our WRE-based approach for range queries, using Poisson salt allocation to encrypt each column. To evaluate the performance of our prototype on realistic data and queries at a variety of scales, we used the SPARTA [35] framework from MIT-LL.

The SPARTA test framework includes a data generator and a query generator. The data generator builds artificial data sets with realistic statistics based on the US Census and Project Gutenberg. The query generator creates queries for this test database based on the desired query types and number of return results.

7.1 Experimental Setup

We used the database generator to generate databases with 100,000 records, 1 million records and 10 million records. We generated over 1,000 queries for each database consisting of a mix of equality and range queries that returned results sizes between 1 and 10,000 records.

We tested the performance of the fixed salt method with 100 and 1,000 salts, and we tested Poisson salt allocation using λ of 100, and 1,000.

The partitioning base for the WRE-ORE setup was $b = 16$ bits for integer types . Even though the fixed salts method does not provide adequate security, it provides a nice comparison for benchmarking versus the Poisson approach, which uses varying numbers of salts for each plaintext.

We performed the tests with two different network scenarios. In the first scenario, the client and the database server are located on the same local network via a 1 Gbps Ethernet switch. In the second scenario, we use Google Cloud Platform to host the database server, and the client connects to the server over the Internet via a 30 Mbps residential connection.

The LAN test utilized an older 12 core (dual Xeon E5645 CPU) server with 64GB of RAM, an array of 10k RPM hard drives, and Postgres 9.6 as the DMBS. The cloud test utilized a Postgres 9.6 SAAS instance with 8 cores, 16GB of RAM, and solid-state disk storage. While the cloud specs are lower than our LAN test, the higher cores and more ram only helped the performance on parallel query tests.

7.2 Experimental Results

The following table shows the total ciphertext expansion from encrypting the SPARTA-generated databases. We encrypted the columns `fname`, `lname`, `ssn`, `city`, and `zip` with WRE, and we encrypted the columns `income`, `last_updated`, and `foo` for range queries.[3]

Encryption Type	DB Size	DB + Indexes Size
100k Plaintext	112 MB	136 MB
100k Encrypted	156 MB	244 MB
1M Plaintext	1116 MB	1365 MB
1M Encrypted	1558 MB	2447 MB
10M Plaintext	11 GB	13 GB
10M Encrypted	15 GB	24 GB

Table 6: Ciphertext Expansion

[3]The SPARTA column `foo` is simply a random 64 bit integer.

We performed two variations of each SPARTA-generated query. The first variation takes the form *SELECT ID from main where ....* Since column `ID` is the primary key, these queries only require that the DMBS must scan the indexes. The second variation takes the form *SELECT * from main where ....* This selects the entire record, and thus requires retrieving the records from storage.

Since all the test variations showed very similar performance trends compared to the plaintext queries, and due to space constraints, we only show tests results for the 10 million record database on the LAN. Interestingly, the experiments on Google Cloud Platform showed slightly better performance than the experiments over the LAN. We believe this was due to hardware differences, as the Google Cloud platform provided SSD drives and our LAN test used a server with conventional magnetic hard drives.

As expected, as the number of salts increases, the performance decreases. The Fixed Salt 1000 performs worse than the Poisson $\lambda = 1000$ tests and Poisson $\lambda = 1000$ performs slightly worse than Poisson $\lambda = 100$. This isn't a surprise, since the Fixed Salt technique applies 1000 salts to each plaintext while the $\lambda = 1000$ results in $\lambda + |M|$ salts.

Figures 6 and 7 show that WRE equality query tests perform very close and in some cases perform better than plaintext. The range queries do sometimes show a significant performance penalty as figures 2 and 4 show. This is due to the query structure and complexity. The equality queries are very simple and much shorter compared to the range queries.

Parallel Query Processing. A unique aspect of our range queries comes from their structure which is "embarrassingly parallel." We conducted range query tests separating each of the outer "or" clauses into separate queries, running them in parallel. The plaintext query in this test is run sequentially. Figures 3 and 5 show the results from range queries using the parallel technique. We do not show any graphs for parallel equality queries since those queries do not contain the same structure to use in parallel. Range queries receive a substantial benefit from utilizing parallel queries.

Query Complexity. While the equality performance results showed near linear trends in the size of the return results, the range queries do not. This is due to another dimension affecting the performance, the complexity of the query. Figure 8 shows the query performance time against the count of outer "or" clauses in each query. These results indicate that the query complexity contributes more to performance than the size of return results.

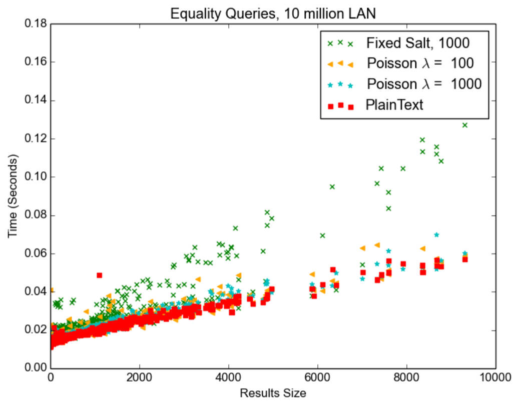

Figure 6: SELECT ID Equality Queries - 10 Million Record Record LAN Test

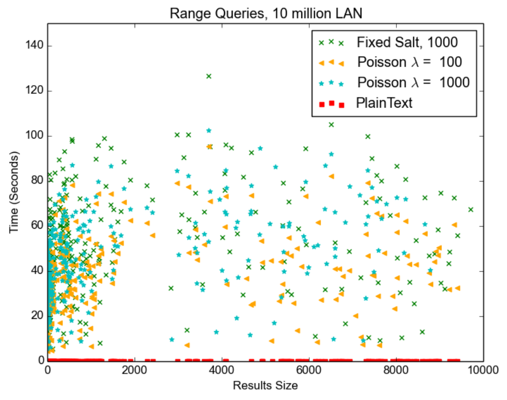

Figure 2: SELECT ID Range Queries - 10 Million Record LAN Test

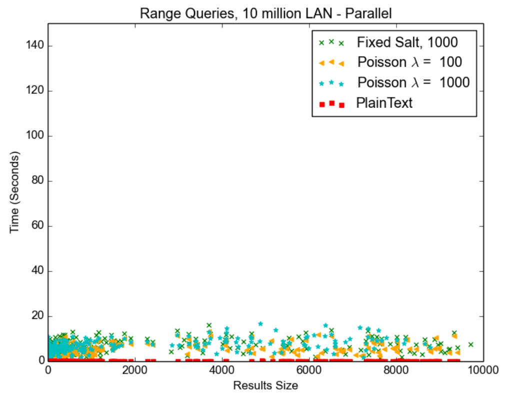

Figure 4: PARALLEL: SELECT ID Range Queries - 10 Million Record LAN Test

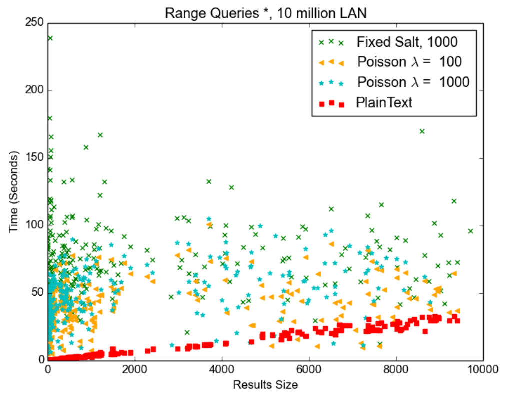

Figure 3: SELECT * Range Queries - 10 Million Record LAN Test

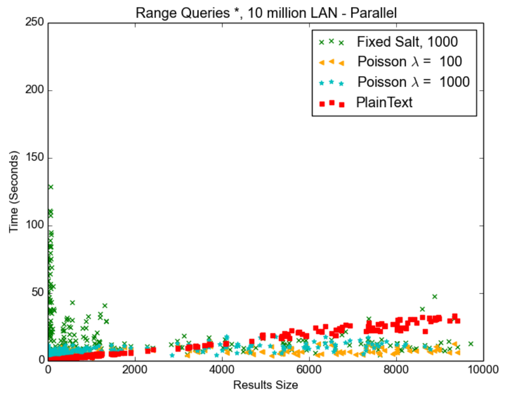

Figure 5: PARALLEL: SELECT * Range Queries - 10 Million Record LAN Test

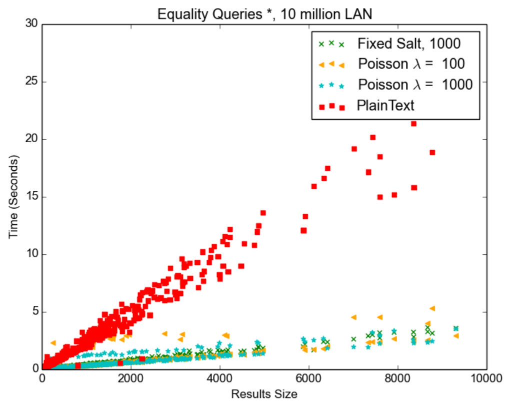

Figure 7: SELECT * Equality Queries - 10 Million Record Record LAN Test

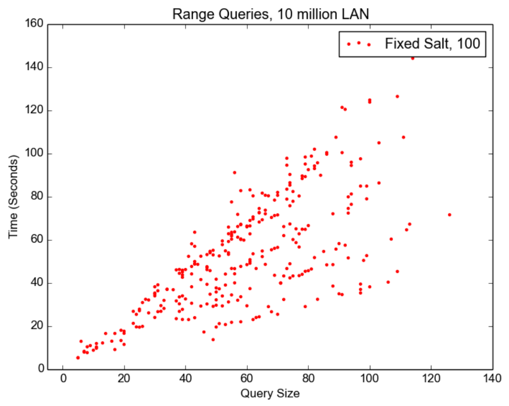

Figure 8: SELECT * Range Queries - Time vs Query Complexity - 10 Million Record LAN Test

References

[1] Always encrypted (database engine). `https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine`. Accessed: 2017-10-31.

[2] Hospital Discharge Data Public Use Data File. `http://www.dshs.state.tx.us/THCIC/Hospitals/Download.shtm`.

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

[4] Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. Provably-Secure Schemes for Basic Query Support in Outsourced Databases. In Steve Barker and Gail-Joon Ahn, editors, *DBSec*, volume 4602 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 2007.

[5] Thomas Baignères, Pascal Junod, and Serge Vaudenay. How far can we go beyond linear cryptanalysis? pages 432–450, 2004.

[6] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and Efficiently Searchable Encryption. In *CRYPTO*, pages 535–552, 2007.

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

[8] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *CRYPTO*, pages 578–595, 2011.

[9] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 563–594. Springer, 2015.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM Conference on Computer and Communications Security*, pages 668–679. ACM, 2015.

[11] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*. The Internet Society, 2014.

[12] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. Practical order-revealing encryption with limited leakage. In *International Conference on Fast Software Encryption*, pages 474–493. Springer, 2016.

[13] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS*, pages 79–88, 2006.

[14] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What else is revealed by order-revealing encryption? In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1155–1166, New York, NY, USA, 2016. ACM.

[15] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–169, 2009.

[16] Eu-Jin Goh. Secure Indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[17] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Alfred V. Aho, editor, *STOC*, pages 182–194. ACM, 1987.

[18] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[19] Patrick Grofig, Martin Hrterich, Isabelle Hang, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schrpfer, and Walter Tighzert. Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data. In Stefan Katzenbeisser, Volkmar Lotz, and Edgar R. Weippl, editors, *Sicherheit*, volume 228 of *LNI*, pages 115–125. GI, 2014.

[20] Paul Grubbs. On deploying property-preserving encryption. Real World Cryptography, 2016.

[21] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. *IACR Cryptology ePrint Archive*, 2016:920, 2016.

[22] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 655–672, 2017.

[23] Warren He, Devdatta Akhawe, Sumeet Jain, Elaine Shi, and Dawn Xiaodong Song. ShadowCrypt: Encrypted Web Applications for Everyone. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM Conference on Computer and Communications Security*, pages 1028–1039. ACM, 2014.

[24] Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(3):333–358, 2012.

[25] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 720–731. VLDB Endowment, 2004.

[26] Seny Kamara. Encrypted search. *ACM Crossroads*, 21(3):30–34, 2015.

[27] Billy Lau, Simon P. Chung, Chengyu Song, Yeongjin Jang, Wenke Lee, and Alexandra Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield-A System's Approach to Data Privacy on Public Cloud. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security Symposium*, pages 33–48. USENIX Association, 2014.

[28] Kevin Lewi and David J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM Conference on Computer and Communications Security*, pages 1167–1178. ACM, 2016.

[29] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM Conference on Computer and Communications Security*, pages 644–655. ACM, 2015.

[30] Omkant Pandey and Yannis Rouselakis. Property preserving symmetric encryption. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 375–391. Springer, 2012.

[31] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In Ted Wobber and Peter Druschel, editors, *SOSP*, pages 85–100. ACM, 2011.

[32] David Pouliot and Charles V. Wright. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM Conference on Computer and Communications Security*, pages 1341–1352. ACM, 2016.

[33] D. Song, D. Wagner, and A. Perrig. Practical Techniques for Searching on Encrypted Data. In *S&P*, pages 44–55, 2000.

[34] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, pages 253–267. IEEE Computer Society, 2013.

[35] Mayank Varia, Benjamin Price, Nicholas Hwang, Ariel Hamlin, Jonathan Herzog, Jill Poland, Michael Reschly, Sophia Yakoubov, and Robert K. Cunningham. Automated Assessment of Secure Search Systems. *SIGOPS Oper. Syst. Rev.*, 49(1):22–30, January 2015.

[36] Jieping Wang and Xiaoyong Du. A secure multi-dimensional partition based index in das. In Yanchun Zhang, Ge Yu, Elisa Bertino, and Guandong Xu, editors, *APWeb*, volume 4976 of *Lecture Notes in Computer Science*, pages 319–330. Springer, 2008.

[37] Liang Wang, Paul Grubbs, Jiahui Lu, Vincent Bindschaedler, David Cash, and Thomas Ristenpart. Side-channel attacks on shared search indexes. In *IEEE Symposium on Security and Privacy*, pages 673–692. IEEE Computer Society, 2017.

[38] Charles V. Wright and David Pouliot. Early detection and analysis of leakage abuse vulnerabilities. Cryptology ePrint Archive, Report 2017/1052, 2017. `http://eprint.iacr.org/2017/1052`.

A Proof of Theorem 5.1

Proof. Let $f(x_a, x_b, ..., x_z)$ be a co-occurrence from a row in a database table containing each column from the row. If there are k columns in the table, We can can choose any $i \in \{2..(k-1)\}$ columns from the row creating a different co-occurrence. All of the $\{2..(k-1)\}$ column co-occurrences we call sub co-occurences of $(x_a, x_b, ..., x_z)$. We can also substitute c or m for x representing ciphertext or plaintexts.

Since ciphertext frequencies take on random frequencies from the Poisson process, the leakage with ciphertext co-occurrence frequencies does not come from the co-occurrence frequency of the ciphertext. The leakage instead is a result of the ratio of the ciphertext co-occurrence frequency and the single column co-occurrence frequencies.

$$\frac{f(c_a, c_b, .., c_z)}{f(c_a) \cdot f(c_b), ..., \cdot f(c_z)}$$

If $enc(m_a, m_b, ..., m_z) = (c_a, c_b, ..., c_z)$ then we expect that as the database gets large:

$$\frac{f(m_a, m_b, ..., m_z)}{f(m_a) \cdot f(m_b) \cdot, ..., \cdot f(m_z)} = \frac{f(c_a, c_b, ..., c_z)}{f(c_a) \cdot f(c_b) \cdot, ..., \cdot f(c_z)}$$

Thus:

$$\begin{aligned} &f(c_a, c_b, .., c_z) \\ &= \frac{f(m_a, m_b, ..., m_z)}{f(m_a) \cdot f(m_b) \cdot, ..., \cdot f(m_z)} \cdot f(c_a) \cdot f(c_b) \cdot, ..., \cdot f(c_z) \\ &= f(m_a, m_b, ..., m_z) \cdot \frac{f(c_a)}{f(m_a)} \cdot \frac{f(c_b)}{f(m_b)} \cdot ... \cdot \frac{f(c_z)}{f(m_z)} \end{aligned}$$

The value $\frac{f(c_i)}{f(m_i)}$ is equal to the salt frequency i. The salt frequencies are chosen from the exponential distribution with parameter λ. Section 4.4.2, Poisson Random Frequencies shows us that $Pr(X > \tau) = e^{-\lambda\tau}$. Since we are single column IND-LOA secure across all columns, we know that this probability is negligible. Thus we can use τ as the highest probable salt frequency.

Knowing the highest probable salt frequency is important, because as τ gets smaller, the distance between $f(m_a, m_b, ..., m_z) \cdot \tau^k$ and $f(m_{a2}, m_{b2}, ..., m_{z2}) \cdot \tau^k$ gets smaller. Thus using the highest probable salt frequency of τ will give us the worst case distance between $f(m_a, m_b, ..., m_z) \cdot \tau^k$ and $f(m_{a2}, m_{b2}, ..., m_{z2}) \cdot \tau^k$.

Thus distinguishing between

$$\big(\mathcal{E}(m_a), \mathcal{E}(m_b), ..., \mathcal{E}(m_z)\big) \text{ and}$$

$$\big(\mathcal{E}(m_{a2}), \mathcal{E}(m_{b2}), ..., \mathcal{E}(m_{z2})\big)$$

is bounded by the statistical distance (from Definition 3):

$$\Delta\big(f(m_a, m_b, ..., m_z) \cdot \tau^k,\\ f(m_{a2}, m_{b2}, ..., m_{z2}) \cdot \tau^k\big)$$

If we let $p_1 = f(m_a, m_b, ..., m_z) \cdot \tau^k$ and $p_2 = f(m_{a2}, m_{b2}, ..., m_{z2}) \cdot \tau^k$ then $\Delta(p_1, p_2)$ is:

$$\Delta = \frac{1}{2}\sum_{k=1}^{n}\left|\binom{n}{k}p_1^k \cdot (1-p_1)^{n-k} - \binom{n}{k}p_2^k \cdot (1-p_2)^{n-k}\right|$$

$\square$

B ORE-WRE Search Algorithm

Algorithm 10 ORE-WRE Query Expansion

```
1: Let $L$ be the smaller term in the range query
2: Let $R$ be the larger term in the range query
3: Let $d$ be the unit base (ex 10, 16, 2)
4: Let $U$ be a function that divides search terms into lists of tuples of (unitID, unit)
5: Let $S$ be the salt generation algorithm, its input is a tuple of (unitID, unit), and its output is
   a conjunction of tuples of (salt, unitID, unit) for all possible salts for the (unitID, unit) tuple.
6: Let $And$ be a function whose input is a list and outputs a Disjunction of all elements in the
   list.
7: Let $Or$ be a function whose input is a list and outputs a Conjunction of all elements in the list.
8: Replicate :: Int $\rightarrow a \rightarrow [a]$. Let (Replicate $d$ $x$) be a function that creates a list of length $d$ of
   duplicate $x$ items.
9: zip :: $[a] \rightarrow [b] \rightarrow [(a,b)]$. zip takes two lists and returns a list of correspending pairs. If one
   input list is short, excess elements of the longer list are discarded.
10: let ++ be list concatenation
11: snoc :: $[a] \rightarrow a \rightarrow [a]$. snoc appends an element to the end of a list
12: map :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$. Map applies a function to each value in a list and returns a new
    list.
13: Query find all x such that $L \le x \le R$
14: function QUERY($L$, $R$, $U$, $S$)
15:     Let $[(ul_1, l_1)..(ul_i, l_i)] \leftarrow U(L)$
16:     Let $[(ur_1, r_1)..(ur_i, r_i)] \leftarrow U(R)$
17:     Let $x$ = first unit position where $l_i \neq r_i$
18:     Let $same = [(ul_1, l_1)..(ul_{(x-1)}, l_{(x-1)})]$
19:     Let $q_1$ = QBuild($HI, [r_{x+1}..r_i], same$)
20:     Let $q_2$ = QBuildMid($ul_x, l_x, r_x, same$)
21:     Let $q_3$ = QBuild($LO, [l_{x+1}..l_i], same$)
        return $Or$ ($q_1$ ++ $q_2$ ++ $q_3$)
22: function QBUILD($id, [(u_x, p_x)..(u_i, p_i)], same$)
```

```
23:     **if** length $[(u_x, p_x)..(u_i, p_i)] == 0$ **then**
24:         **return** And (S (same))
25:     **if** $id == HI$ **then**
26:         Let range = $[0..(p_x - 1)]$
27:     **else**
28:         Let range = $[(p_x + 1)..(d - 1)]$
29:     Let temp1 = (zip (replicate $d$ $u_x$) range)
30:     Let temp2 = map (snoc same) temp1
31:     Let query = map And (map (map S) t2)
32:     Let r =
33:         (QBuild $[(u_{x+1}, p_{x+1})..(u_i, p_i)], same)$
34: **return** query ++ r
35: **function** QBuildMid$(ul_x, l_x, r_x, same)$
36:     Let Z = zip (replicate $d$ $ul_x$) $[(l_x + 1)..(r_x - 1)]$
37:     Let $X$ = map (snoc same) Z
38: **return** map $Or$ $X$
```