Spring 2019

Intrusion-tolerant Order-preserving Encryption

John Huson

Intrusion-tolerant Order-preserving Encryption

John Benjamin Huson

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2019

FACULTY COMMITTEE:

Committee Chair: Dr. Xunhua Wang

Committee Members/ Readers:

Dr. Brett C. Tjaden

Dr. M. Hossain Heydari

Dedication

This work is dedicated to those who contribute to and maintain the privacy of others.

Acknowledgments

I would like to acknowledge Dr. Xunhua Wang for his support and insight throughout this project. Without his guidance, knowledge and drive for concrete examples this work would not have reached its full potential.

I would like to thank my committee members Dr. Mohammed Heydari and Dr. Brett Tjaden for serving. I would also like to acknowledge the James Madison University Computer Science Department, including all of the professors and students I have had the pleasure to work with over the past few years.

Table of Contents

List of Tables

List of Equations

List of Figures

Abstract

Traditional encryption schemes such as AES and RSA aim to achieve the highest level of security, often indistinguishable security under the adaptive chosen-ciphertext attack. Ciphertexts generated by such encryption schemes do not leak useful information. As a result, such ciphertexts do not support efficient searchability nor range queries.

Order-preserving encryption is a relatively new encryption paradigm that allows for efficient queries on ciphertexts. In order-preserving encryption, the data-encrypting key is a long-term symmetric key that needs to stay online for insertion, query and deletion operations, making it an attractive target for attacks.

In this thesis, an intrusion-tolerant order-preserving encryption system was developed to support range queries on encrypted data. Within this system, the long-term symmetric key is shared among multiple (say n) servers and is never reconstructed in full, at any single point. An adversary who has compromised less than a threshold number (say $t : t <= n$) of said servers will not be able to reconstruct the shared key, and therefore will not be able to decrypt stolen ciphertexts. This system is robust in that only a threshold number of servers are required for insertions into or range queries over the ciphertexts within the database. In this thesis, a prototype implementation was developed to show the feasibility of this design. This system can be used to enhance the security of range-queryable encrypted data stored in the cloud.

Chapter 1

Introduction

Problem Statement

In order to preserve data confidentiality on an outsourced, potentially hostile, database system it is common practice to employ strong encryption on the data stored. Strong encryption of data hinders the ability to perform searching, as well as range queries on said data. The cryptographic trait of indistinguishably from that of random data obliterates the ability for an outsourced database to perform searching, comparison, as well as range queries on the stored ciphertext.

In the past few decades there has been significant interest and development in searchable encryption [25] [8] , order preserving encryption (OPE) [1] [6] [7] , order preserving tagging (OPT) [22] [23] [15] and order revealing encryption (ORE) [9] [11] [16]. Constructions therein attempt to provide data confidentiality as well as the ability for an existing database system to perform efficient range queries on stored ciphertext values alone.

Within all of these constructions, OPE, ORE and OPT, it is assumed that the data owner, the client, is solely trusted with knowledge of the encryption key which is used to encrypt and decrypt the records to and from the database. In the case of ORE this encryption key is used by the client to generate tags, or tokens, by which the database system will be able to ascertain the ordering of the underlying data. This simplistic key assumption is problematic for three reasons:

1. Inherent risk of key compromise and exposure
2. Lack of fault tolerance when the key is lost, or the client is not available
3. Difficulty related to key rotation, and key management.

As enumerated, these problems are not acceptable in distributed database systems where fault tolerance and intrusion tolerance are requirements. Within this work we will validate the applicability of secret sharing by means of a distributed pseudo-random function for

existing OPE, ORE and OPT constructions. We will also provide a practical implementation of an intrusion tolerant order preserving encryption (itOPE) scheme which employs a $(t, n) - threshold$ based general access structure for key sequence sharing across multiple participants.

Background OPE, OPT, ORE

OPE variants have been in use for centuries. One-pad code books, by definition, preserve the ordering of plaintext values within resulting ciphertext through the ordered bijection of the code book's structure. Intuitively this construction causes a key double in size of the plaintext domain. Modern OPE constructions attempt to reduce the key size, while maintaining the same result, an ordered bijection from plaintext to ciphertext.

Order Preserving Encryption

Modern OPE has it's roots in the database community. Within the early exploration of order preserving encryption schemes a few were notable. Ozsoyoglu et al. proposed two schemes, the summation of random numbers, as well as the computation of increasing polynomial functions. [20] Explained in depth in Chapter 2, Ozsoyoglu proposed taking the sum of m random numbers where m is the value being encrypted. Though this solution is clever, it is shown by Agrawal et al. [1] that the underlying ciphertext which is computed still retains density characteristics of the plaintext. The second solution, computation of increasing polynomial function, approaches the problem from the same direction. By computing a sequence of increasing polynomial functions one is able to create ciphertext that retains order, but unfortunately also suffers from the density leakage as Agrawal proves.

The first attempt at an OPE scheme with limited leakage was shown by Agrawal et al. [1]. Therein the author's key motivation was to develop a bijection which retained the order of the underlying plaintext, *and* was as close to a uniform distribution as possible. Within the OPE Scheme (OPES) [1] construction, detailed in Chapter 2, the author's construction relies on mathematical transformations of the plaintext domain and ciphertext range in order to "flatten" the data. This was a major improvement over existing schemes which relied on summation of random numbers, and increasing polynomial function techniques. The OPES scheme ciphertext values do not retain the distribution shape of the under-

lying plaintext data as do the summation of random numbers and increasing polynomial techniques.

Another technique known at the time for retention of partial ordering of ciphertext data was a bucketing scheme, wherein the plaintext domain is broken into buckets of fixed space which retain order. The client encrypts the plaintext data with strong encryption, and then tells the server in which bucket to store the ciphertext. On the surface this scheme looks promising, although Agrawal [1] points out that the balance between false positive inclusion when a bucket is too large, and tight estimation exposure when the bucket is too small is catastrophically problematic. [1]

The intuition of OPES, which is expressed in more depth in Chapter 2 is to create a function $F(K, m) \rightarrow ct$ wherein $m_1 < m_i < m_M$ and correspondingly $ct_1 < ct_i < ct_N$ where N is the ciphertext range of numbers in which the plaintext domain M is to be mapped. The details of this scheme are outlined in Chapter 2, but at a high level the process consists of three phases:

1. Modeling Phase
2. Flattening Phase
3. Transformation Phase

Within the "Modeling Phase" the plaintext is bucketed such that the overall densities of each bucket's members are linear up to an arbitrary threshold. By creating bucket boundaries that cause linear densities of the members of each bucket, the author shows that a simple quadratic transformation in the "Flattening Phase" of the density of each bucket will cause the resultant flattened data to maintain a constant, or uniform, distribution. After both plaintext domain and ciphertext range has been flattened a scaling process is used in order to transform the flattened plaintext values to corresponding ciphertext values. The result of the OPES construction provides an algorithm by which the distribution shape of the underlying plaintext data is obliterated in the resulting ciphertext.

As seen, this transformation uses the data itself to generate the applicable mappings and coefficients to "encrypt" the data, which is drastically different than traditional cryptography which derives randomness from a random key. This scheme is akin to simulation of randomness through the transformation process. Within this construction the "key" consists of the bucket boundaries, quadratic coefficients, and scaling factors used within the algorithm.

Following Agrawal's contributions, Boldyreva et al. [6] [7] continued this line of research by contributing the first cryptographic treatment of OPE. Boldyreva formally defines the security notions involved with OPE by loosening the strict definitions of Indistiguishability of Chosen Plaintext Attack (IND-CPA) to more closely resemble the best possible security an OPE could ever achieve. Much like Bellare et al. [4] did for weaking the security definition for distinct chosen plaintext for deterministic encryption mechanisms by defining IND-DCPA, Boldyreva defined Indistinguishability of Ordered Plaintext Attack (IND-OCPA) [6]. Within this new, weakened definition the relative ordering of the plaintext is the only leakage. Within the same work, it was also proven that no OPE scheme is actually capable of providing this security guarantee without a ciphertext range which is exponentially larger in size to the plaintext domain [6].

With this fatal proof that no OPE scheme is capable of proving IND-OCPA, Boldyreva decided to provide a construction in which the security was derived on the security of pseudo-random functions used within her scheme. Another security definition related to the security of the order preserving pseudo-random functions was created: Pseudo-random order-preserving function against chosen-ciphertext attack (POPF-CCA). With this notation, the authors decided to use said notation for the security definition of their OPE construction. [6]

Unfortunately this is a drastically weaker security definition than the ideal IND-OCPA, "requiring that no adversary can distinguish between oracle access to the encryption algorithm of the scheme or corresponding 'ideal' object," [6] the ideal object being a truely random order preserving function. This means that the best security their construction could possibly achieve hinges on the order preserving pseudo-random function chosen. Details of Boldyreva's construction can be found in Chapter 2.

The overwhelming willingness of the database community [21] to quickly develop and deploy solutions based on Boldyreva's symmetric deterministic order preserving scheme gave pause to cryptographers, as the basic security premise completely revolved around the security of the pseudo-random order preserving functions. Although potential for leakage other than order was warned in [**?**], a definitive assertion as to how much additional leakage was unclear until Boldyreva's follow-up work [7].

It turns out, based on Boldyreva's follow-up paper in 2011 [7], that much is leaked from their construction. The assumptions of security based on ROPF-CCA needed to be further

explored, such as if ROPF is even one-way. Boldyreva states the definition of Window One-Wayness (WOW) as the ability for an adversary, given a set of ciphertexts of uniformly random messages, to determine the plaintext interval in which the plaintext lies. Using the construction defined in [6] it is shown that *upwards of half of the most significant plaintext bits are leaked*, and information about the relative distance between ciphertexts are also leaked from their construction.

Order Preserving Tagging and Interactive Protocols

Given the drawbacks within [1] [6] and [7] Popa et al. [22] decided to attack the problem from another direction. Instead of attempting to create another OPE scheme, of which is impossible to achieve IND-OCPA as proven by Boldyreva, Popa's solution was to create a construction which consisted of an interactive protocol, and resulting encoding scheme which could be used by an existing database to ascertain ordering of values. Popa's construction [22] consists of a binary tree data structure comprised of encrypted values using strong encryption. Insertions, deletions and comparison operations are controlled directly by the client through an interactive protocol, wherein the client requests a particular node in the tree by the path to said node. The client would then decrypt the value of the node with the secret key locally, and perform the comparison with the value for which the client is range searching. The particular path values would form the basis of the tagging scheme which would be used by a database to index the order relationship between the elements.

Since strong encryption is performed by the client, and all order relations are ascertained by the client through the interactive protocol, only the ordering of ciphertexts are known by the server within the encrypted binary tree. Detailed in Chapter 2, it is shown that this construction is capable of providing ideal security.

Order Revealing Encryption

ORE is not to be confused with OPE, or OPT. ORE provides a mechanism by which ordering relationships can be realized within ciphertexts, but does not provide the ability to decrypt the ciphertexts back to their plaintext values.

The study of ORE was first entertained by Boneh [9], where a new theoretical construction was created that utilized multi-linear maps for a semantically secure ORE. Unfortunately this was not a very practical solution as it was highly complex and computationally

infeasible.

Practical order revealing constructions based on secure pseudo-random functions were first explored by Chenette et al. [11] and Lewi et al. [16] in 2016. Primary to all of these constructions is the use of modular blinding of the particular bit string elements within a plaintext with a strong pseudo-random function. In order to compare ciphertexts the implementer would walk through each bit, or block in [16] of a ciphertext and each bit, or block, of the plaintext and perform a modular addition to un-blind comparison value at that location in the bit string.

These ORE schemes provide clear equality leakage for individual bits within ciphertext values, and do not comply with the ideal security proposed by Boldyreva through IND-OCPA. This said, they do provide limited leakage utilizing a non-interactive solution, which makes these schemes more attractive to practitioners.

Background Secret Sharing

Key Decomposition

Secret sharing is a mechanism by which multiple suspicious parties can work collaboratively in order to perform actions with a particular secret key without recomposition of said secret key. Shamir and Blakely both independently developed constructions in 1979 [24] [5] which provide the ability to "divide D [secret key] into n pieces [. . .] in such a way that: 1.) knowledge of any k or more D_i pieces makes D easily computable; 2.) knowledge of any $[t]$-1 or fewer D_i pieces leaves D completely undetermined." [24]

Shamir's construction is based on interpolation of Lagrange polynomials [24]. By decomposing the secret key into distinct values of a polynomial function, k participants can work together to reconstruct the secret key. Detailed examples are given in Chapter 2.

Blakely's scheme relies on intersection points of hyper planes, following a similar intuition of Shamir's construction. Given k participants who have knowledge of particular intersecting planes, the collective will be able to reconstitute the secret key by working together to find the intersection of all of their planes.

In order to use the Shamir construction without any one particular participant being able to reconstruct the secret key, most implementations make use of the multiplicative homomorphism which is a common homomorphism within schemes that abide by the Diffie-

Hellman Assumption. $Enc(m) = \sum_{i=1}^{k} m^{D_i} = m^{\sum i=1^{k} D_i}$ in which each participant D_i uses their own key share to compute a partial result, and the final result consists of the sum of the partial participant computations. Though this works well in a public key setting, using this mechanism with a strong pseudo-random function is infeasible without key reconstruction.

General Access Structures

Though the Shamir construction works very well for (k, n) threshold schemes, it was shown in 1989 by Ito et al. [13] that this construction is limited, and does not work for generic general access structures. Given a set of persons P, within Shamir's construction only a subset $P' \subset P$ can construct the secret key if $|P'| >= k$, meaning that the only access structure supported would be $P' \subset P : |P'| >= k$.

Ito stipulates that by assigning several shadows of a (k, n) threshold scheme to each participant it is possible to create a threshold sharing scheme which is much more akin to generic general access structures. With Ito's scheme, you can create various access structures based on boolean ORs-of-ANDs which is more flexible than the existing [24] [5] threshold schemes.

Computation of Block Ciphers

Due to the apparent benefit of algebraic homomorphisms needed within existing secret sharing schemes as defined by Shamir, Blakely and Ito for shared computation of primitives that abide by the Diffie-Hellman Assumption, Naor et al [19] and Brickell et al. [10] began the study of the perceived inability of utilizing secret sharing for shared block cipher computation. Brickell shows that, through a new paradigm of Sequence Sharing, by using general access structures as defined by Ito [13], one can perform shared block cipher computation by nesting encryptions with derived keys in order through $\binom{n}{t-1}$ encryptions using $\binom{n}{t-1}$ keys required from t participants. Brickell shows that by performing $Enc(k_{\binom{n}{t-1}}, Enc(k_i, \cdots Enc(k_1, m)))$ for $i = \binom{n}{t-1} \rightarrow 1$ an Access Structure given multiple shadows, or shares, of a key are able to encrypt, and also decrypt a particular value without needing to rely on algebraic homomorphisms, but rather use existing block ciphers which by design do not have such algebraic traits.

In a follow-up work by Martin et al. [18] it is shown that an XOR based block cipher sharing construction is possible using Naor's distributed pseudo-random function. Within

this construction instead of following the Brickell shared block cipher mechanism, it is shown instead that a participant can send a random nonce out to other participants, of which would then encrypt this random nonce with their key share. The originating participant, with the plaintext value, would then blind the plaintext value with a sequence of XOR operations:

$Enc(m) = (m \oplus Enc(k_1, r) \oplus Enc(k_i, r) \oplus Enc(k_{\binom{n}{t-1}}, r), r)$

Within the scheme defined by Martin the plaintext value is not leaked to any other participants, and when the combination of the encrypted values comes back from each participant using their own key shares no information is leaked. This blinding mechanism prevents participants from gaining any more information about the particular plaintext.

Computation of Pseudo Random Functions

Agrawal et al. [2] follows up on distributed symmetric-key encryption by proposing a distributed pseudo-random function construction that improves on Naor's construction. Within the construction Agrawal proposes a simple Access Structure based on a (t, n) threshold scheme. The key distribution algorithm is performed in the setup of the algorithm, and creates $\binom{d:=n}{n-t+1}$ random numbers for distribution to hosts. Given that there is an ORs-of-ANDs access structure $D_1, \ldots, D_i, \cdots, D_d$ then the i-th random number is given to all participants in D_i. The encryption and decryption is very similar to the constructions defined in [19] [17] and [18], with the improvement being a mechanism for ciphertext integrity checking. Within Agrawal's construction it is shown that by having the DPRF compute a commitment value which is $(r \oplus m)$ where r is a random nonce and m is the plaintext value, integrity of the ciphertext can be checked upon decryption.

Overview

OPE, OPT and ORE schemes have been well defined and extensively studied over the past several years. All of the modern cryptographic constructions [1] [6] [7] [22] [23] [15] [9] [11] [16] rely primarily on the data owner, or client (proxy), to provide encryption key security, and fault tolerance. As stated in the Problem Statement, this means that the data owner, or client, is responsible to protect against secret key compromise, as well as maintaining fault tolerance for when systems are offline.

Within intrusion tolerant design it is important to never fully reconstruct the encryption

key which is distributed among the participants. Much as Wu et al. [26] shows how a web server is constructed to provide intrusion tolerance, this work will focus on how to build an intrusion tolerant OPE scheme. By the nature of a threshold based secret sharing scheme, this work will show how fault tolerance and intrusion tolerance within OPE is achieved.

Chapter 2

Background and Related Work

Order Preserving Encryption from Antiquity

Order preserving encryption is not a recent development. Implementation of code books through a "one-part" substitution cipher is a clear example of an order preserving encryption scheme. The general construction of a code-book encryption scheme takes an enumerated domain of plaintext values, and performs a direct one-to-one mapping to an enumerated range of ciphertext values. Typically one would construct two-parts of a code-book, an encipherment part and a corresponding decipherment part. With these two parts Alice would be able to create a ciphertext from the ordered encipherment "part" of the code-book and Bob would be able to decipher said ciphertext with the ordered decipherment "part". Ultimately each part would be ordered such that either party would be able to quickly search the plaintext domain or the ciphertext range to yield the particular encryption or decryption mapping. A practical design "improvement" over a two-part code-book scheme was to have both parts collapsed into one book as it is much easier to distribute a one-part code-book in the field of battle as opposed to a potentially more secure two-part code book. This one-part code-book was developed in such a way that the plaintext and ciphertext mapping retained order.

Within this construction Alice and Bob would exchange a book of ordered plaintext to ordered ciphertext mappings as the key for future messages. In order to encipher a message Alice would search through the ordered plaintext values within the code book, and substitute the mapped ciphertext values in the enciphered message to Bob. Similarly, when Bob receives the ciphertext, Bob will perform an ordered search on the ciphertext values in the code-book from the message, and map them back to their respective plaintext values. The ordered nature of the plaintext to ciphertext mapping was critical for the speed of encryption and decryption. If the code words were not ordered, Alice would need to, at the worst case, scan the entirety of the code-book to encrypt a message if they were not ordered.

Though at the time computers were not in use, Alice and Bob would be able to manually perform message enciphering in $O(logn)$ complexity using a binary search method through the code-book searching for the word to encipher or decipher, as opposed to an $O(n)$ complexity from an unordered code-book where n is the number of code words in the book. Take for example the following one-part code-book:

Table 2.1: Example One-Part Code-Book

Plaintext	Ciphertext
Alpha	101
Bravo	102
Charlie	103

> The vocabulary terms would be arranged in alphabetical order, so that they could be readily found when enciphering messages; [...] In the other section, the code groups would be rearranged in straight alphabetical (or numeric) order, so as to be readily found when deciphering [...]. [12]

As we explore the more modern constructions of order preserving encryption using mathematical discoveries and algorithms to eliminate the need for a dictionary mapping "code-book" we should not loose sight of the fact that the spirit of all surveyed order preserving encryption primitives enumerated herein have the following properties in common with that of a one-part code-book:

1. Mapping function from a plaintext domain to a ciphertext range
2. Retaining order of underlying plaintext in the resulting ciphertext
3. Given a plaintext and ciphertext pairing, an attacker has the ability to infer knowledge of other ciphertext values

Order Preserving Encryption from Database Community

The first modern order preserving encipherment and decipherment scheme was developed within the database community in 2004 by Agrawal et al. The problem, expressed by Agrawal in [1], is that even though database systems offer access control as a mechanism to restrict access to sensitive data, these access control implementations are insufficient. The insufficiency lies in the fact that the implemented database access controls are only

specifically protecting database Application Programming Interfaces (APIs) which are easily bypassed by an attacker who gains access to the raw database files or memory.

Data confidentiality therefore, must be preserved in the internal memory and on disk representation of the data directly within the raw database files and memory itself in order to prevent unauthorized data accesses that may happen outside of the purview of the database access control mechanism. A potential solution, traditional data cryptography, unfortunately does not solve the problem without creating untenable performance degradation. Using cryptography one attempting to perform a range query on a set of data would have to iterate over the entirety of the database, decrypting every value, in order to know the ordering of the underlying data set. This full table scan and decryption makes even simple queries painfully slow. Even employing keyword searchable encryption [25][8], or deterministic encryption techniques do not remedy this situation as the underlying ciphertext does not retain order even if it may retain exact keyword matches. Due to this apparent shortcoming it is not possible to organize ciphertext within an efficient data-structure such as a B-tree in a database. [1]

Order Preserving Encryption Intuition

The intuition of Agrawal's OPES scheme [1] is expressed as "Generate $|P|$ unique values from a user-specified target distribution $[R]$ and sort them into a table T. The encrypted value c_i of p_i is then given by $c_i = T[i]$. $[\dots]$ Here T is the encryption key that must be kept secret." [1] It should be noted that this is strikingly similar to the construction behind one-part code-book schemes of antiquity, where the one-part code book is the mapping table T. Agrawal notes there are clear shortcomings within this simplified intuitive construction, namely: the encryption key size (which is 2x the size of the database); and the scheme is problematic to update because in order to update the mapping table one would have to re-encrypt all values greater than the value inserted, much like a node insertion in a linked list.

Based on the flaws listed from the intuition, it is evident that in order to have an efficient scheme, said scheme would need to remove the need for such a large encryption key to be practical, and remove the need for re-encryption of values on insertion. Agrawal mentions in related work three schemes which attempt to apply a treatment to tackle these flaws, enumerated below:

1. Summation of Random Numbers
2. Nested Polynomial Functions
3. Bucketing

Summation of Random Numbers

A proposed order preserving encryption scheme defined in [20] involves the summation of random numbers generated by a pseudo-random number generator. The resulting ciphertext will be the sum of $R_1 \dots R_j \dots R_P$ where P is the value of the plaintext, and R_j is the jth randomly chosen number from the pseudo-random number generator R.

: Random Number Summation

$$\sum_{j=0}^{P} R_j \tag{2.1}$$

This will indeed provide an improvement in key space, as one will only be required to generate a key that is the size of the plaintext domain. Improvement is also realized in the graceful updating capabilities of the database. Though there are improvements with this scheme it is shown in [1] that the distribution of the encrypted values is directly proportional with the input distribution. This scheme is susceptible to frequency analysis as the distribution of the plaintext matches the ciphertext directly.

Nested Polynomial Functions

Further defined in [20] there is a scheme based in nested polynomial functions which provides similar improvements. When nesting a set of polynomial functions that are strictly increasing and providing the randomness through coefficients of said polynomial functions as the encryption key, this construction will allow for a transformation from plaintext to ciphertext. Using this method the ciphertext distribution appears different than the input distribution. Unfortunately though for this scheme, Agrawal found that the shape of the ciphertext distribution does still retain characteristics of the input data distribution [1], and therefore is not suitable.

: Monotonically Increasing Nested Polynomial Functions

$$f_i(x) = C_{i-1}x^i + C_i; i \in 1 \leq i < k \tag{2.2}$$

: Computation of Nested Polynomial Functions

$$f(x) = f_{i+1}(f_i(x)); \forall i : k > i \geq 1 \tag{2.3}$$

Bucketing

Another interesting solution to this problem is found in ciphertext partitioning based on the ordering of underlying plaintext. Within this construction conventional encryption is used to add data confidentiality, but the ciphertexts are grouped with buckets. For example imagine we have three buckets the boundaries of which are $B_1 \rightarrow [0, \ldots, 5]; B_2 \rightarrow [6, \ldots, 10]; B_3 \rightarrow [11, \ldots, 15]$. Considering we encrypt the plaintext number 7, we would first locate which bucket would be suitable to contain the number 7 which would be B_2, then we would encrypt the number 7 and insert it into the partition B_2. For a query of the range of plaintext numbers $4 \leq i \leq 8$, one would take both buckets which potentially contain those values, then decrypt every number within those buckets, returning the results of corresponding plaintext values within that range.

This scheme is more efficient than a non-bucketing scheme of conventionally encrypted values, due to the fact that you will not incur a $O(n)$ penalty, though there are potentially significant false-positives within the result set. Depending on the bucket size, this failure rate could be substantial. Moreover, Agrawal has shown that the bucket size is directly correlative to estimation exposure. "It is shown [...] that the post-processing overhead can become excessive if a course partitioning is used for bucketization. On the other hand, a fine partitioning makes the scheme vulnerable to estimation exposure, particularly if an equi-width partitioning is used." [1]

Order Preserving Encryption Scheme Construction

Based on the research from [1] we have two new requirements for an order preserving scheme. Firstly our resulting ciphertext should not retain statistical attributes of the underlying plaintext data. As seen in the Summation of Random Numbers scheme and the Nested Polynomial Function constructions, if our ciphertext retains the exact same, or slightly shifted, distribution of the input plaintext domain the scheme will be susceptible to estimation exposure and frequency analysis. It is paramount that the ciphertext values maintain a near perfectly uniform distribution such as that of a random sampling. Secondly the scheme should provide for an exact one to one mapping from plaintext to ciphertext in order to avoid post-processing overhead as well as estimation exposure. Thirdly the scheme should create ciphertext values that can act as a direct drop-in replacement for the plaintext within a database scheme, as to not over complicate queries.

The novel OPES scheme presented in [1] accomplishes these tasks through the use of a three phase scheme, enumerated below:

1. **Model:** The input and target distributions are modeled as piece-wise linear splines
2. **Flatten:** The plaintext database P is transformed into a "flat" database F such that the values in F are uniformly distributed.
3. **Transform:** The flat database F is transformed into the cipher database C such that the values in C are distributed according to the target distribution

Order Preserving Encryption Scheme Model Phase

Within the first phase of the OPES scheme stipulated in [1] the density function of the plaintext domain space is computed, and a linear spline is taken across the endpoints of the density of the domain. This linear spline is our reference point from which our first bucket splitting operation will be performed. The point at which the density of the underlying plaintext data differs most dramatically from this spline is the where the bucket boundary will be located. This process is repeated recursively until an arbitrary threshold is achieved where the density of each bucket is no longer differing with the computed linear spline across the bucket boundaries. This may seem like an odd first step, but this step is actually creating buckets that exhibit a linear density function, which is an important step for the flattening stage. At the end of the modeling phase we will have a multitude of buckets of plaintext values, all with linear densities.

Since within [1] the authors are allowing a custom ciphertext range to which the plaintext values will be mapped, it is important to note that we must also perform this modeling phase for our ciphertext range as well as the plaintext range. Just as stated above, we begin by calculating the linear spline over the density of the ciphertext range, and recursively bucket the ciphertext range so that in the end each ciphertext range will maintain a linear distribution.

Order Preserving Encryption Scheme Flatten Phase

The goal of the flattening phase within the OPES scheme is to convert the bucketed plaintext from a linear density to a constant density (aka uniform density). As stated in the analysis on the prior work in this field, all of the existing mechanisms (summation, nested polynomials) retained the underlying density of the plaintext values within the ciphertext. The result of this flattening operation is to convert our linear distribution to a uniform distribution.

Agrawal realized that if you want to convert a linear function into a constant function you need to apply a quadratic transformation to the linear function. In order to ensure that all flattened buckets are uniform as well, one can apply a "scaling factor" to each of the plaintext buckets within this flattening scheme which will result in uniform flattened bucket lengths. With uniform flattened bucket lengths, as well as uniform flattened values within the buckets, the plaintext values will no longer exhibit any particularities of the original plaintext values.

In order for this scheme to be reversed for decryption, or repeated deterministically, state needs to be kept. Firstly from our modeling phase we need to know the bucket boundaries that have been recursively constructed by start and endpoints. Secondly we need to keep track of each particular bucket's quadratic coefficient used to turn the plaintext bucket values into the flattened uniformly distributed values. Thirdly we need to keep track of the particular scaling factor for each plaintext bucket in order to scale them to uniformly sized flattened database buckets.

Since the authors dictate that the ciphertext range to which the plaintext will be mapped is given, the modeled ciphertext buckets must also be flatten as well. This follows the same procedure as the plaintext bucket to flattened database. We start by calculating the quadratic coefficient to which each of our bucketed ciphertext values will be mapped. Then

we create a scaling factor for each bucket such that the resulting flattened ciphertext buckets will be uniform in size as well. Much like the plaintext flattening stage, we need to maintain each ciphertext bucket's start and endpoint, as well as the quadratic coefficient and scaling factor in order to convert to and from the flattened database and the ciphertext range.

Order Preserving Encryption Scheme Transform Phase

Using our plaintext flattened intermediate database and our ciphertext flattened intermediate database, in order to map these two flattened databases together we need to create a matching factor for each bucket pairing which will allow us to seamlessly transition values from one flattened database to the other. This is required as the ciphertext flattened buckets are not guaranteed to exactly line up with the plaintext flattened buckets. This matching factor value per bucket pairing needs to be stored within the state so that this process can be deterministically reproduced.

It is important to note that within the construction [1] this shared state that has been growing is the actual "encryption key" for the algorithm.

Future Work for Order Preserving Encryption Scheme

As seen, the construction of [1] is limited. In order to have a truly uniform ciphertext distribution a requirement is to know, or have a tight sampling of, the entirety of the plaintext domain space prior to encryption. Though the author claims database updates are graceful, the resulting ciphertext values will be bound to the same bucket definitions defined in the initial modeling phase, thereby skewing the bucket densities potentially creating a non-uniform distribution in the resulting ciphertext. Moreover a significant improvement to this scheme would be to create a stateless scheme. Within this scheme the "encryption key" consists of the state of the buckets from the modeling phase and needed scaling factors for the flattening phase. Another shortcoming from this work is the cryptographic treatment of the scheme itself. The author provides no formal security definition for this order preserving encryption construction, nor what is leaked from this particular OPES scheme.

With these problems presented future order preserving encryption schemes must provide a stateless scheme, as well as a security definition and provable security with leakages.

Order Preserving Encryption from Cryptographic Community

The first security definition and cryptographic treatment of order preserving encryption was performed by Boldyreva et al. [6] Within this work an investigation of what OPE is from a security perspective was developed and new notions of security definitions were created. It might be apparent to the reader at this point that it is not possible for any order preserving scheme to preserve standard notions of security such as indistinguishability against chosen-plaintext attack (IND-CPA). This is primarily due to the fact that an order preserving encryption scheme must be deterministic in nature, as well as the fact that a particular quality of the plaintext value is leaked through the ciphertext. Simply put if you know two ciphertext values you will at the very least and by definition know which one is larger or smaller. This leakage is not defined in depth within [6] paper, and it remained as future work.

Due to this self evident fact that we cannot have an order preserving encryption scheme that will maintain IND-CPA, it is prudent to outline exactly what one could expect a perfect order preserving scheme to obtain. In the same fashion as [4] introduced the concept of indistinguishability against distinct chosen-plaintext attack (IND-DCPA) for the security model for deterministic encryption schemes based on a pseudo-random function (PRF), Boldyreva invented a new security definition which is to define a perfect order preserving function: indistinguishability against ordered chosen-plaintext attack (IND-OCPA). This security definition intentionally weakens IND-CPA in order to provide a starting point for analysis of order preserving schemes. Boldyreva goes on to show that this definition is, as of that time, unachievable [6] using their construction without the ciphertext range being exponentially larger than the plaintext domain. Never the less this scheme is something by which order preserving schemes should strive to achieve, and in future work be compared to.

Since it is infeasible [6] to create a deterministic order preserving scheme that can satisfy IND-OCPA, the authors decided to settle on a further lessened scheme: pseudo-random order preserving function against chosen-ciphertext attack (POPF-CCA). Instead of further restricting the definition of IND-OCPA, the author has chosen to focus instead on the security of the PRF which will be used to generate the mappings from plaintext domain to ciphertext range. POPF-CCA “requries that no adversary can distinguish between oracle

access to the encryption algorithm of the scheme or a corresponding "ideal" object." [6] This means that an adversary given the algorithm itself should not be able to determine the result of a ciphertext as oracle access should be indistinguishable from a perfectly random result. The author uses this definition as a building block of the security of the construction proving that the result of the algorithm itself (the order preserving pseudo-random function) cannot be distinguished from the "ideal" result of a truly random function the scheme is sound even if it cannot feasibly achieve IND-OCPA.

Order Preserving Encryption Construction

The construction outlined within [6] aims to produce an algorithm that samples a pseudo-random order preserving function, with provable POPF-CCA security, from a plaintext domain into a reasonably sized (2x the domain size) ciphertext range. The intuition behind the scheme is that any order-preserving function from $D \rightarrow [1, \ldots, M]$ plaintext domain to $R \rightarrow [1, \ldots, N]$ ciphertext range can be represented by an ordered combination of M out of N items. Below you can see a few examples of such combinations:

Table 2.2: Example k Permutations of Mapping $M \rightarrow N$ Permutations

Domain (M)	Range (N)	$RPF_1(M) \rightarrow N$	$RPF_2(M) \rightarrow N$	$RPF_k(M) \rightarrow N$
1	25	25	25	83
2	43	43	78	92
3	78	78	83	101
-	83	-	-	-
-	92	-	-	-
-	101	-	-	-

Boldyreva shows [6] that an M out of N bijective order preserving function is no different than the distribution of an experiment of selecting random items from a finite range without replacement as visualized in Figure 2.1.

Boldyreva discovered that a random order preserving function is identical to the negative hyper-geometric distribution.[6] In order to visualize, consider an experiment shown in Figure 2.1 followed by Table 2.3 where we have N total balls in a bin where M balls are black and $N - M$ balls are white. Drawing i balls from the bin at random without replacement each time a black ball is drawn we can map the least unmapped value in our domain to the ith number which corresponds to the total number of balls drawn to this point

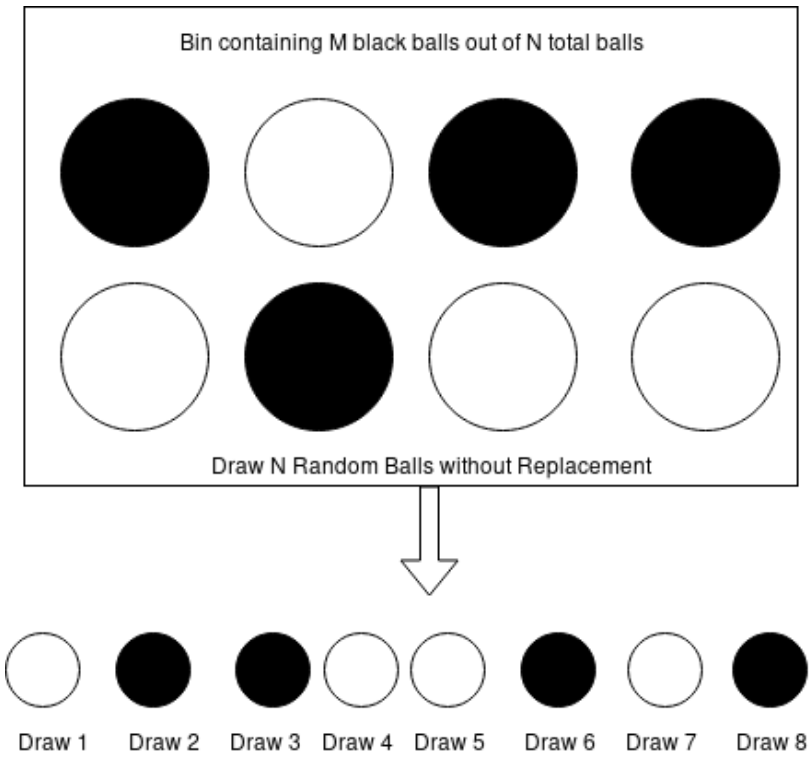


Figure 2.1: Drawing Randomly from a Bin of Balls

in the experiment. Below is a tabular visualization based on our figure above. Consider we are mapping $M \in [1, \ldots, 4] \rightarrow N \in [1, \ldots, 8]$. Table 2.3 shows the mapping based on drawing, and Table 2.4 shows the resultant mappings from this concrete example.

Table 2.3: Tabular Representation of Drawing from Bin of Balls Without Replacement

of draw	Color of Ball Drawn	$PRF(M) \rightarrow N[i-1]$
1	White	-
2	Black	$PRF(1) \rightarrow N[1] = 2$
3	Black	$PRF(2) \rightarrow N[2] = 3$
4	White	-
5	White	-
6	Black	$PRF(3) \rightarrow M[5] = 6$
7	White	-
8	Black	$PRF(4) \rightarrow M[7] = 8$

Using the link to the negative hyper-geometric function as a pseudo-random order preserving function one can inject randomness into the algorithm, and maintain POPF-CCA, as the algorithm itself does not contain the randomness. Oracle access to the PRF is there-

Table 2.4: Result of Hypergeometric Distribution Mapping

M	$\rightarrow N$
1	2
2	3
3	6
4	8

fore indistinguishable for chosen ciphertext attack. Unfortunately there is one hurdle which is that there is no efficient mechanism in place to calculate the negative hyper-geometric distribution. To remedy this [6] employs the algorithm from [14] which solves the problem of isolating hyper-geometric random variates efficiently and with exact precision. With this algorithm Boldyreva's construction is able to figure out the random variate, or the number of a particular type of balls drawn, from a population. The construction is able to isolate the number of white balls (failures) drawn given a domain and range for a particular element in a domain element.

: Negative Hypergeometric Distribution as PRF [6]

$$Pr[f(x) \leq y < f(x+1) : f \leftarrow OPF_{[M],[N]}] = \frac{\binom{y}{x}\binom{N-y}{M-x}}{\binom{N}{M}} \tag{2.4}$$

This construction is secure than the scheme presented from Agrawal because the random order preserving function oracle access for a given ciphertext is indistinguishable from a sampling of the ideal hypergeometric distribution. Agrawal's algorithm does not use a random order preserving function, but rather relies on the algorithm itself and the distribution density of the underlying plaintext data to provide the bijection. This scheme is better also due to the statelessness, as all one needs to inject is randomness in the form of coin tosses, as opposed to storing the state of buckets, coefficients and scaling factors for the computations.

Order Preserving Encryption Lazy Sampling Algorithm

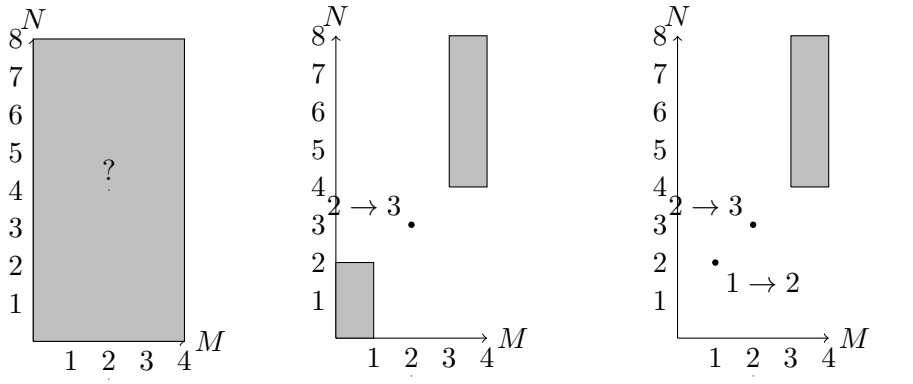
Boldyreva's OPE construction [6] consists of a recursive lazy sampling function LazySample (and inverse LazySampleInv), as well as a mechanism for keeping state of the random

coins used TapeGen.

Since it is untenable to pre-compute every possible mapping possibility ahead of time for very large plaintext domains, as in a one-part code-book implementation, Boldyreva's construction relies on a binary search algorithm $O(logn)$ for isolating the particular mappings. At a high level the algorithm calculates the mid value of the consecutive integer ciphertext range and determines what the domain gap should be for each side of the range based on the hypergeometric random variates, given a sequence of random coin flips from the key's randomness. When the domain and range gaps are mapped together, the range is then recursively subdivided, and the new subdivided sides of the range are mapped to the domain gap. This process continues through the binary search until there is only one domain element left for a particular subdivided ciphertext range.

After convergence on the plaintext value we wish to encrypt a uniformly random value from the resulting ciphertext values is chosen from the range gap that is left. The randomness is taken from the TapeGen algorithm just as the hypergeometric variates were taken from during the binary search. This allows for repeatability and deterministic behavior. You can see in Figure 2.2 the recursive algorithm used to LazySample. Starting from the mid point in the domain, the algorithm recursively maps domain values to range gaps.

Figure 2.2: Graph of Recursive Search



Order Preserving Encryption Drawbacks

Though this construction is intriguing and very novel, there are a few drawbacks that are noted within [6][7]. One drawback is that this scheme is not capable of providing

IND-OCPA without using an exponentially larger ciphertext image than the underlying plaintext. This is not feasible as the algorithm will significantly slow down with very large ciphertext ranges. Due to the recursive nature of the algorithm using a binary search, very large plaintext and ciphertext spaces will dramatically reduce efficiency. Another problem noted is that the algorithm has worse security than deterministic encryption, and you can gain much information from the ciphertext that is generated from this scheme. Outstanding from [6] but resolved in [7] is the open question of what exactly is leaked about the plaintext from the ciphertext. Within [7] the author follows up with the fact that approximations of plaintext values as well as the distance between plaintext values are significantly leaked within the ciphertext.

Another area for improvement is to replace the hypergeometric variate computation with an efficient negative hypergeometric calculation algorithm, but as of this no current solutions exist to address this open problem.

Of greater concern, and addressed in [7] is the open question regarding the ciphertext range size to the plaintext domain size. The authors of [6] merely state that the ciphertext range must be greater than or equal to the plaintext domain, and throw out that the ciphertext range can, quite arbitrarily, be twice the size of the plaintext domain.

Revisiting Order Preserving Encryption

Immediately after [6] was introduced it was seized by the applied community, according to Boldyreva, and implemented by many as the definitive order preserving encryption scheme. [7] This was unfortunate as the authors indicated in [7] that there were many unknowns within the construction regarding leakage, and these concerns warranted further examination prior to implementation. This Pandora's Box of sorts which was opened in [6] and the authors attempt to quell premature implementation in the followup work [7] by furthering the study of order preserving encryption security notations.

In the original work [6] we were introduced to the concept of a Random Order Preserving Function (ROPF) as well as a Pseudo-Random Order Preserving Function (POPF) as the basis for the security of the construction. The key difference between the two is that the ROPF is what a POPF attempts to strive to produce, indistinguishability of oracle access from a ROPF. The security notation for this indistinguishability is defined as ROPF-CCA. Within [7] the security definitions of order preserving encryption is further broken down

into the following composite constructs:

1. Window One-Wayness
2. Window Distance One-Wayness

Window One-Wayness

The definition of Window One-Wayness in order preserving cryptography is "For $1 <= r <= M$ and $z >= 1$, the adversary is given a set of z ciphertexts of (uniformly) random messages and is asked to come up with an interval of size r within which one of the underlying plaintexts lies." [7] This definition is abbreviated $(r, z) - WOW$ for brevity. Restated, given a number of random ciphertext values, from an order preserving encrypted database, can an adversary isolate a window of the plaintext domain, in which said ciphertext's corresponding plaintext values falls.

Window Distance One-Wayness

The definition of Window Distance One-Wayness in order preserving cryptography is defined by Boldyreva as "the adversary attempts to guess the interval of size r in which the distance between any two out of z random plaintexts lies, for $1 <= r <= M$ and $z >= 2$." [7] This notation is abbreviated $(r, z) - WDOW$. This can be restated as given two or more ciphertexts, can an adversary isolate the relative difference in the underlying plaintext within a particular range, or window size.

With these two One-Wayness definitions, it was proven by Boldyreva [7] that there exists a certain amount of leakage within OPE construction. In an attempt to remedy these leakages two new constructions were presented in [7]. One of the constructions is a modification to the OPE scheme which improves $(r, z) - WOW$ posture of the construction. The other construction is a brand new construction that relies on order preserving tagging based on monotone minimal hashing techniques.

Modular Order Preserving Encryption

Modular Order Preserving Encryption (MOPE) [7] uses the exact same OPE scheme defined in [6] but requires an additional step of performing a modular addition operation over

a secret base to the message space prior to encryption. This additional step improves (r, z)-WOW for the OPE construction due to the fact that the resulting ciphertext will end up "wrapping around" the plaintext space. Equation 2.5 shows the extent of the modification to the original OPE scheme. By drawing a random j from the plaintext domain M during the setup, and storing j as a part of the secret key, we are able to perform a modular subtraction in order to blind the location of the ciphertext result.

: MOPE - Modular OPE Encryption[7]

$$j \in M; Enc_{MOPE} = Enc_{OPE}(K, m–j(mod|M|)) \tag{2.5}$$

: MOPE - Modular OPE Decryption[7]

$$j \in M; Dec_{MOPE} = Dec_{OPE}(K, c) + j(mod|M|) \tag{2.6}$$

The decryption of the MOPE scheme is shown in Equation 2.6. Where $|M|$ is the length of the plaintext domain, K is the encryption key, j is the MOPE blinding secret and c is the ciphertext value. Since the modular addition is performed on the plaintext [7] shows that this will maintain the bijective nature of the original OPE scheme within the MOPE scheme. If the modular blinding operation were to be performed on the ciphertext, bijection is not guaranteed, meaning that there could be collisions in the mapping from plaintext to ciphertext. Also interesting to note is that this "wrap around" addition to the scheme only wraps around once as the modulo base is the size of the plaintext space.

This improved scheme completely blinds the approximate location of the plaintext, but still allows for effective range queries, so long as one takes into account the modular behavior. This scheme however looses the trait of order preserving encryption in that it is possible for a bigger plaintext to be a smaller ciphertext. This modified, modular order preserving encryption will handle range queries, but will not retain order as was an original requirement of any OPE scheme.

Order Preserving Tagging

An alternate scheme presented within [7] is that of order preserving tagging. It is proven in this work that the order preserving tagging scheme is capable of achieving IND-OCPA, the gold standard of order preserving encryption security notations. This construction is implemented through the use of monotone minimal perfect hashing defined by Belazzougui et al. [3]. Monotone minimal perfect hashing functions map bijectively n keys into the set of $[0, n-1]$ using a Trie data structure based on binary prefixing. After a compacted prefix trie is constructed from a plaintext domain, the resulting hash function will retain lexicographic ordering of the plaintext values.

The construction for Order Preserving Tagging as defined by [7] uses monotone minimal perfect hashing to create an order preserving tagging scheme. The process is informally defined as given the database of plaintext values, compute the monotone minimal perfect hash of each value within the plaintext domain. Then use strong encryption (IND-CPA) to encrypt the plaintext values, and merely append the corresponding monotone hash value to the ciphertext value. It is proven that the only attribute of the plaintext that is leaked is the relative ordering of the plaintext, which complies with the IND-OCPA security notation.

Unfortunately the drawbacks of this scheme completely defeat one of the main selling points of the original symmetric OPE scheme from the original Boldyreva construction [6], being that one would need to have the entire plaintext domain known ahead of time to use this scheme, as monotone minimal perfect hashing requires, by the nature of a prefix trie, all of the values ahead of time to determine the lexicographic ordering of the values in the trie. This particular construction is less resilient to changes in the underlying data than the Agrawal [1] construction in that it does not handle graceful updates at all. In order to add new values to the database one would need to decrypt the entire database, compute the monotone minimal perfect hashing of each ciphertext, then tag every single entry in the entire database. It is apparent that this scheme will not work in implementation, though it does initiate the discussion about order preserving encoding in the field.

Order Preserving Tagging (Encoding)

After Boldyreva showed in [7] that it is not possible to create a symmetric order preserving encryption with the ideal security notion of IND-OCPA where the only leakage of

the algorithm would be the ordering of the plaintext values, Popa et al. [22] took pursued a different direction. Very similar to Boldyreva's order preserving tagging technique, Popa's construction uses semantically secure encryption to protect the confidentiality of the plaintext values, and uses an additional "tagging" attribute to specify the relative ordering of the ciphertexts. Unlike Boldyreva's tagging solution, Popa implements an interactive protocol to remove the requirement of a priori full database knowledge.

Popa uses the fact that a binary search tree data-structure, by definition, is ordered which can be visualized in Figure 2.3. By creating a binary search tree of ciphertext values, where the decision on which branch to take is performed by the client who has the encryption key, an interactive protocol will allow for ideal security.

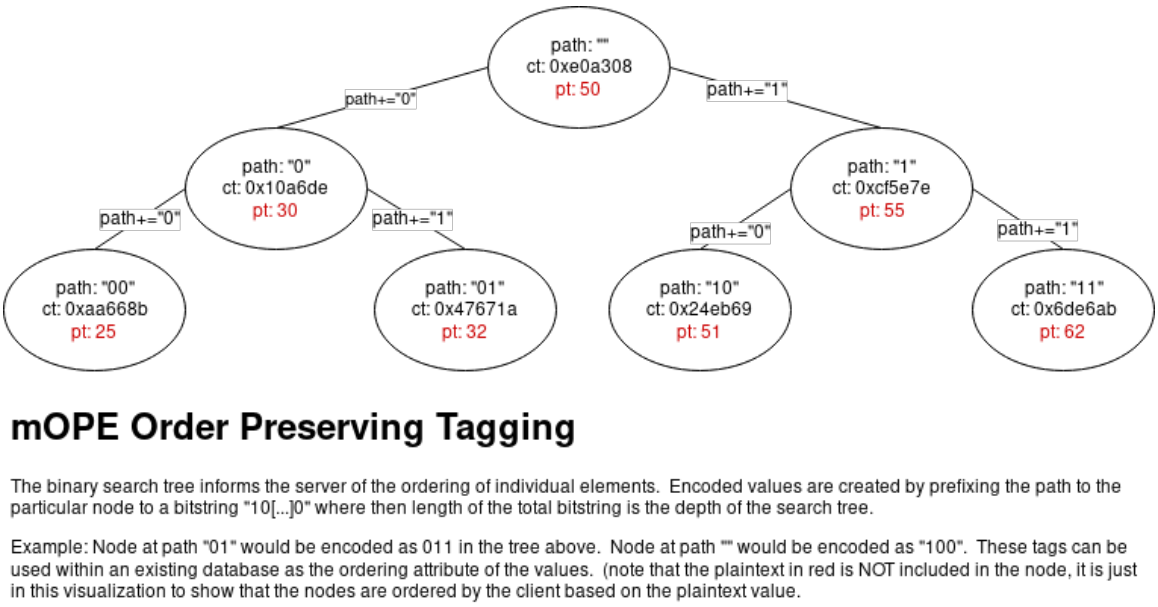


Figure 2.3: mOPE Binary Search Tree

Within Popa's construction [22] a client, or data owner, would request from the remote mOPE server the root node of the tree. When the server returns the ciphertext for the root node, the client is able to decrypt the value of the root node, and decide if the right or left path should be taken based on if the value being encrypted (or looked up) is greater than or less than the value of the root node. Well known recursive and non-recursive solutions exist for binary tree traversals, but the interactive nature of this tree allows the server no intuition as to the particular values of the ciphertexts, only the relative ordering. Figure 3.4

provides a flow chart of this interactive process.

Up until the advent of interactive protocols for order preserving tagging, no order preserving scheme was capable of providing ideal security. Improvements to the Popa construction have been shown which improve insertion performance [23] for big data applications where insertion optimization is crutial. Query optimization has also been constructed [15] which allow a data owner or client to perform faster lookups, but comes at the cost of maintaining a local cache of ciphtertext to plaintext mappings.

Order Revealing Encryption

Research from [7] shows that deterministic symmetric order preserving encryption has substantial leakage. This leakage primarily pertains to the approximate plaintext estimation, as well as approximate distance between plaintexts given a set of ciphertext values. The leakages have been found to be upwards of half of the bits of the plaintext given a ciphertext. It was also shown that the Boldyreva's MOPE construction outlined is vulnerable to inference attacks based on inference attacks by watching the queries performed, which leads to a trivial deduction of the "secret" modular shift value used thereby removing the blinding benefits the scheme introduced.

In 2015 Boneh et al. [9] developed a new theoretical construction that radically changed the trajectory of the order preserving encryption field. Within Boneh's construction, which uses multi-linear maps, the authors were able to create a semantically secure construction, albeit impractical, which does not comply with traditional symmetric order preserving definitions. Upon examination of what is required by practitioners in the field regarding sorting, searching and range queries, it is evident that a symmetric encryption scheme is not required. As eluded to in [7] Boldyreva's order preserving tagging construction and followed up by Popa's [22], all that is required is isolation of the ciphertext who's plaintext values fall within a particular range. There is no need to actually perform a decryption, or reverse transformation back to the plaintext value, as the plaintext can be encrypted with semantically secure cryptographic mechanisms. Order revealing encryption is therefore a subset of order preserving encryption.

Practical Order Revealing Encryption

Chenette et al. [11] set to the task of creating a practical construction of an order revealing encryption which was based on more practical primatives than Boneh's construction. The authors settled on a scheme which relied on the security of pseudo-random functions for their construction. This scheme resembles hashing and is implemented using provable strong security primitives already used in the cryptographic community.

As mentioned order revealing encryption is subtly different than order preserving encryption in that there is no decryption attribute. The ciphertext still retain order and allow for sorting and searching. When pairing this order revealing encryption with a strong traditional encryption mechanism on the value itself, one can create tags which allow for order revealing traits. When analyzing order revealing versus order preserving, Chenette discovered [11] that order preserving encryption is a subset of order revealing encryption, in that order preserving encryption incorporates a decryption mechanism.

The order revealing construction created by Chenette [11] provides an encryption algorithm which relies on a secure pseudo-random function F which derives it's randomness from a key K. Let $b_1||...||b_n$ be the binary representation of a plaintext value where n is the length of the plaintext bit string, and $M \in \mathbb{Z}_3$. For each i in $[n]$ compute the Equation 2.7 concatenating each u_i value into a ciphertext value $(u_1, \ldots, u_n)$.

: Practical Order Revealing Encryption[11]

$$\forall i \in [n]; ct = ||_{i=1}^{n} F(K, (i, b_1 \ldots b_{i-1}||0_{n-i})) + b_i (mod M) \tag{2.7}$$

As seen within the construction we are taking our secure pseudo-random function and passing a key and a bit-string the result of which is modular added base M to the bit value in the i-th location in the bit string. As seen in this construction, if we choose $M = 3$ we will have a 2 times expansion. Table 2.5 provides a simple toy example for visualization of the encryption of 15 (bit string of 1111).

Given Table 2.5 the resulting ciphertext value is shown in Equation 2.8.

As seen this construction requires n secure pseudo-random function computations where n is the length of the bit string being encrypted. Also seen is the amount of ciphertext ex-

Table 2.5: Example ORE Practical

Iteration (i)	PRF Parameter ($i \| b_{i-1}$)	u_i
1	1,0000	F(K, (1,0000)) + 1 (mod 3)
2	2,1000	F(K, (2,1000)) + 1 (mod 3)
3	3,1100	F(K, (3,1100)) + 1 (mod 3)
4	4,1110	F(K, (4,1110)) + 1 (mod 3)

: Practical Order Revealing Encryption Example Result

$$(F_k(1,0000)+1 mod 3, F_k(2,1000)+1 mod 3, F_k(3,1100)+1 mod 3, F_k(4,1110)+1 mod 3) \quad (2.8)$$

pansion required is a factor of $M-1$ where M is an arbitrary security parameter greater than 3. What is nice about this construction is the fact that it resembles a prefix trie, much like the monotone minimal perfect hashing scheme noted by Boldyreva [6]. The comparison mechanism simply traverses to the first u_i value between the two ciphertexts being compared, and merely un-blinds the modular addition. Equation 2.9 shows the comparison operation given two ciphtertext values ct_1 and ct_2 respectively. If this condition is true, then ct_2 is in fact larger than ct_1.

: Practical Order Revealing Comparison

$$u_i^{ct_2} = u_i^{ct_1} + 1 (mod M) \quad (2.9)$$

With this extremely elegant construction in place we can see clearly that the order of the plaintext values are revealed within the resulting ciphertext, and we are able to use a strong pseudo-random function to generate the ciphertext. Though this is a definite improvement over the order preserving schemes from [6] [7] there is still room for improvement. Notice in the algorithm the prefix of the bit strings leaks information about the two ciphertext values, specifically equality of high order bits. This does not fall within IND-OCPA security notation clearly as more than just the ordering of the plaintexts are realized. It is shown within [11] that this prefix equality between ciphertext values reveals equality of the first i

bits of each ciphertext value.

Improved Order Revealing Encryption Small Domain

Directly following the practical order revealing encryption notion was discovered, Lewi et al. started investigating a new construction that would solve a few notable problems within Chenette's practical order revealing scheme. The most obvious problem with the Chenette construction is the fact that a database of ciphertexts using the same secret key are all directly comparable by the server housing the ciphertexts. Another obvious problem is that the leakage incurred, up to the first differing bit of the ciphertext was untenable.

In order to address the first problem Lewi [16] implemented a "right/left" framework in which the encryption process was broken into a right encryption algorithm, and a left encryption algorithm. The right ciphertexts generated were not directly comparable with other right ciphertexts. Only the left ciphertext was directly comparable with the right ciphertext values. In implementation this means that a practitioner would be able to store right ciphertexts within the remote database system, and the remote database system would no longer be able to infer anything about the corresponding plaintext value from the right ciphertext alone. Within Equation 2.10 is the left side encryption algorithm, and Equation 2.11 demonstrates the right side encryption algorithm.

The left encryption algorithm as seen in Equation 2.10 requires additional key matter. Besides the encryption key, the additional key matter required is a random permutation function $\Pi(m) \rightarrow m$ over the plaintext domain M such that Π performs a unique bijection of each plaintext value within the plaintext domain to another plaintext value within said small domain. A practical example of this is realized in Table 2.6. Consider the values of a small domain in $[1, \ldots, 8]$ as shown in Table 2.6. The need for this permutation, and more importantly for the inverse permutation is to obfuscate the actual values being encrypted. The result of this "left" ciphertext is shown to actually be the modular blinding "key" to the right ciphertext.

: Improved "Left" Small Domain Order Revealing Encryption

$$ct_{left} = (F(K, \Pi(m)), \Pi(m)) \tag{2.10}$$

Table 2.6: Small Domain Order Revealing Encryption Π Function

m	$\Pi(m)$
1	5
2	6
3	7
4	1
5	3
6	8
7	4
8	2

The right encryption algorithm of this small domain ORE takes the value which is to be encrypted, and performs order comparisons between that value and all other values within the small domain. This algorithm can be seen in Equation 2.11. The resulting "right" ciphertext is therefore a tuple, or concatenation, of the comparison of each and every value within the small domain with the value being encrypted blinded by the "left" ciphertext value. This produces an incredible amount of ciphertext expansion in the resulting right side ciphertext as seen in Table 2.7. Given H is a one way hashing function, F is a secure pseudo-random function, and r is a random nonce generated at encryption time, which is stored along with the ciphertext value for comparison purposes.

: Improved "Right" Small Domain Order Revealing Encryption

$$ct_{right} = r||_{i=0}^{|M|} CMP(\Pi^{-1}(i), m) + H(F_k(i), r)(mod 3) \tag{2.11}$$

Table 2.7: Example Right Encryption of Value 5; Small Domain Order Revealing Encryption

i	$\Pi^{-1}(i)$	$ct_{right}[i]$
1	4	$CMP(4,5) + H(F_k(4), r) mod 3$
2	8	$CMP(8,5) + H(F_k(8), r) mod 3$
3	5	$CMP(5,5) + H(F_k(5), r) mod 3$
4	7	$CMP(7,5) + H(F_k(7), r) mod 3$
5	1	$CMP(1,5) + H(F_k(1), r) mod 3$
6	2	$CMP(2,5) + H(F_k(2), r) mod 3$
7	3	$CMP(3,5) + H(F_k(3), r) mod 3$
8	6	$CMP(6,5) + H(F_k(6), r) mod 3$

As seen in our simple example the ciphertext expansion is very large for the "right" side encryption algorithm. With our small domain example of 8 values, we can see that the right side encryption of the value '5' proves to be $|M| * 2 + r$ in size which is 16 bits plus the length of our random nonce r, compared to the 3 bits of plaintext. Within the construction [16] it is shown that the required space to encrypt one 32 bit integer is 224 bytes.

By encoding the comparisons of the small domain directly inside the ciphertext, and using the "left" component to "decrypt" the right ciphertext Lewi was able to get around the problem of directly comparable ciphertexts stored within the database. By using a small domain of size 256 for example and embedding this solution into Chenette's ORE scheme, it is proven [16] that the leakage from comparisons by left ciphertext only propogate to the first byte block that differs, instead of the first bit that differs.

Chapter 3

Intrusion Tolerent Order Preserving Encryption Implementation

Overview

Based on the prior art in the field from Chapter 2 it has been shown that there exist implementations of order preserving tagging that maintain ideal security (IND-OCPA) which provide ordering of ciphertexts based on the ordering of the underlying plaintext. It is shown that common to all of these implementations is the use of a pseudo-random function as a basis for the security of the ciphertext stored within a database. Furthermore, it is shown that one is able to create a secure distributed pseudo-random function using general access structures. Within this chapter it will be shown *how* an order preserving encryption scheme is constructed to provide intrusion tolerance as well as fault tolerance, improving existing order preserving encryption schemes.

Given in Table 3.1 is a list of existing OPE, OPT, ORE schemes along with their known leakages and security notations. Each entry is also expressed with the level of difficulty required to distribute this particular algorithm. As you can see, since many of the existing OPE, OPT and ORE constructions are completely stateless, and rely on a pseudo-random function, it is plain to see that those particular implementations can be distributed utilizing a distributed pseudo-random function.

Table 3.1: Ease of Intrusion Tolerance on Existing OPE/OPT/ORE

Encryption Scheme	Security	Distributable?	Leakage
Agrawal '04 OPES	None	Hard	Unknown
Boldyreva '09 OPE	POPF-CCA	Easy	Half bits/relative distance
Boldyreva '11 MOPE	POPF-CCA	Easy	Relative distance
Boldyreva '11 OPT	IND-OCPA	Hard	Order
Popa '13 mOPE/stOPE	IND-OCPA	Easy	Order
Kerschbaum '14 OPE	IND-OCPA	Hard	Order
Roche '2016	IND-OCPA	Easy	Order
Chenette '16 ORE	PRF	Easy	First differing bit
Lewi '16 sdORE	PRF	Easy	First differing block

More interestingly are the constructions that are *hard to* distribute. Agrawal's OPES

[1] scheme completely relies on the state of the algorithm setup as the encryption key. By maintaining the state of the quadratic coefficients, bucket boundaries and scaling factors a distributed OPES construction would need to maintain a shared distributed state across all participants, which is infeasible in practice. Boldyreva's OPT scheme [7] similarly requires state preservation from the setup process, which does not lend itself to intrusion tolerance. In essence it is very hard to carve up a construction that relies on initial state in a manner that will allow for discrete access structures to share said state.

Though intrusion tolerance can be applied to any encryption primitive that utilizes a pseudo-random function through the use of a distributed pseudo-random function, the decision on order preserving implementation outlined within this chapter was consciously made to provide the best possible security definition (IND-OCPA), as well as most clearly demonstrable. To that end it was decided that Popa's mOPE [22] was the most suitable due to the fact that it is capable of maintaining IND-OCPA security, and additionally it is fairly simple to understand as it uses common data structures and allows for ease in asymptotic calculations.

The choice in distributed pseudo-random function for this intrusion tolerant order preserving tagging implementation is derived from Agrawal's DISE [2] implementation primarily due to the fact that it builds off of a long line of successful distributed pseudo-random function implementations [19] [18] [17] with the added improvement of ciphertext integrity validation.

Design

The construction of an intrusion tolerant order preserving encryption scheme (itOPE) described can be broken into three independent classifications of processes, a key-share dealer process, a mOPE process, as well as multiple participant processes. The organization of this section follows:

1. Definition of the key sharing implementation
2. Definition of the mOPE interactive protocol implementation
3. Definition of the distributed pseudo-random function and client interface

itOPE Dealer Key Sharing Process

Symmetric key based pseudo-random functions do not allow for algebraic homomorphisms by design. In order to create a key sharing mechanism across a general access structure for which will accommodate a threshold of participants we need to follow the OR-of-ANDs boolean access structures which enables threshold key sequence sharing [**?**] [2]. This construction uses the algorithm mentioned in [2] Section 8.2 for key distribution. Informally, given n participants P, collaborating in $d = \binom{n}{n-t+1}$ access structures D, d key shares k need to be created, and distributed such that $P_i \leftarrow k_j$ where $P_i \in D_j$. This algorithm can be seen in Algorithm 1 and visualized within Figure 3.1. An implementation of this process in Go can be found in Appendix A within *GenerateKeys* and *KeyAssignment* functions.

Algorithm 1 Dealer Key Sharing Algorithm

procedure KEYSHARES

keys ←[1, ..., n] ▷ Initialize keys as array length n

for *i in range* $\binom{n}{n-t+1}$ **do** ▷ for all Access Structures based on (t,n)

keys$_i$ ← *RNG*() ▷ *Add randomly generated key to "keys"*

for ∀*i* ∈[n] **do** ▷ for all Access Structures based on (t,n)

for ∀*j* ∈*keys* ***do*** ▷ *for all Access Structures based on (t,n)*

if *participant*$_i$ ∈ *AccessStructure*$_j$ ***then*** *participant*$_i$ ← *key*$_j$

return *0*

=0

By distributing the keys using the aforementioned algorithm, it is shown in Figure 3.1 that it is not possible to have all n keyshares needed for the distributed pseudo-random function operations without at least t participants collaborating. This key sharing mechanism unfortunately expands key shares exponentially, but in all practicality it is rare that the number of participants involved in a distributed computation are more than twenty in size [2]. Although this general access structure key sharing scheme is less efficient than an algebraic constructs such as discovered by Shamir [24] it is never the less practical for limited collections of participants.

It should also be noted, that by using a symmetric pseudo-random function this con-

struction is fairly limited with regards to key rotation capabilities. It is evident that an encryption process that uses key shadows which are fixed, meaning that one would need all of the particular key shares that was used for an encryption process in order to perform decryption, does not allow for ease of key rotations. It would be interesting future work to work through key rotation capabilities within this scheme. This is effectively the same as having one key, as in this case the key shares together fully compose the one encryption key. Within Shamir [24] it is shown that due to the algebraic nature of key derivation, and use within schemes that abide by the Diffie-Hellman Assumption, the group encryption mechanism actually results in the encryption of the secret the shares were derived from. This is helpful in key rotation as so long as the modular addition of the key parts equal the original key, the encryption and decryption mechanism will still operate on the blinded original key. This is not the case because symmetric pseudo-random functions have no algebraic traits and do not abide by the Diffie-Hellman Assumption.

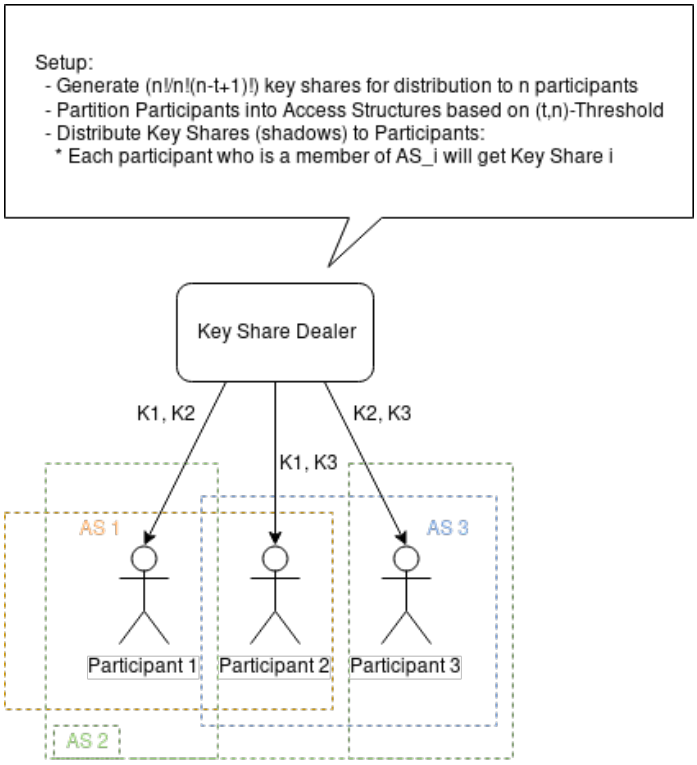


Figure 3.1: Visualization of dOPE Dealer Process

From a practical perspective it is of note that merely because this construction contains the dealer as the key generation service within this embodiment, does not mean that one

is required to have a secure dealer. In order to remove the requirement of having a secure dealer within another embodiment, one can merely implement the access structure code within Appendix A directly within the participant services themselves, removing the need for a dealer process completely.

In another embodiment where participants figure out for themselves what the key shadows should be two or more nodes who are within a particular access structure can create random numbers themselves and come up with a composite key by performing an exclusive-or of those randomly created numbers to use as the key for that particular key shadow. This implementation includes an authority key dealer for simplicity of design, and ease of understanding.

Also of note, within Appendix B it is shown that after all calculated key shares are distributed, the dealer process shuts down. This is to prevent the full key from staying available in a fully reconstructed state. It is important to never have the fully recomposed encryption key available.

As can be seen in Appendix B this particular construction uses the GRPC protocol over TLS with client certificate authentication over HTTP2. This client certificate authentication allows for the dealer to validate authentication of the connecting participants in order to validate the participants are actually part of the distributed pseudo-random function cluster.

itOPE Participant Encryption/Decryption Process

Due to the distributed design, it is important to have every participant capable of initiating the encryption, decryption, storage and range query functionality independently. One of the primary drawbacks with current OPE, OPT and ORE constructions is the inherent sequential operations required to encode or encrypt regarding the pseudo-random function access. To this end itOPE is designed such that any participant, no matter in which access structure that participant resides, must be able to perform all actions related to search, and data storage.

As seen in Figure 3.2 the encryption process follows directly the art from shared block ciphers over the past 20 years. As shown by Agrawal et al. [2], a distributed pseudo-random function is as secure as the composite pseudo-random functions it of which it is comprised. Initial attempts at creating "sequence sharing" schemes [10] prove that by nesting pseudo-random function accesses with a particular sequence of key shares allows

for decryption through the inverted nesting of said function accesses. This concept of sequence sharing allows for robustness, and distributed computation, although it requires a sequential operation across participants, which is not optimal for multi-party computation. Equation 3.1 outlines the particularities of Brickell's sequence sharing design for encryption and Equation 3.2 shows the corresponding decryption process.

: Brickell Sequence Sharing for Shared Block Cipher Encryption Computation

$$Enc(K, m) \rightarrow Enc(K_{\binom{n}{n-t+1}}, Enc(K_{\binom{n}{n-t+1}-1}, ..., Enc(K_1, m))) \tag{3.1}$$

: Brickell Sequence Sharing for Shared Block Cipher Decryption Computation

$$Dec(K, ct) \rightarrow Dec(K_1, ..., Dec(K_{\binom{n}{n-t+1}-1}, Dec(K_{\binom{n}{n-t+1}}, ct))) \tag{3.2}$$

Clearly this solution will work for shared computation of a block cipher, although it is not efficient, as each participant with the appropriate key share needs to be summoned sequentially, in order, to perform their portion of the block cipher computation.

Directly following Brickell's work, Martin et al. authored two follow up papers [17] and [18] wherein Martin showed the $\oplus$ operator could be used in order to more fully distribute the computation of block ciphers. Within Martin's work on Threshold MAC the authors proposed a scheme by which each individual participant was given the plaintext value, and asked to return the partial MAC result by performing a MAC operation with their own key shares. The result of all of the participants were then $\oplus$-ed together to produce a shared mac, see Equation 3.3.

: Martin Threshold MAC for Shared MAC Computation

$$MAC(K, m) \rightarrow \oplus_{i=1}^{\binom{n}{n-t+1}} MAC(K_i, m) \tag{3.3}$$

This works very well in a distributed model, but now each of the individual participants

were given the value of m, the plaintext. For MAC calculation this doesn't really matter, but when data confidentiality is paramount, this scheme does not limit the number of participants that are in control of the plaintext, which could lead to a curious participant recording the plaintext values. To that end Martin's followup work [18] revisited this concept and created a scheme which would allow for the best of both worlds, shown in Equation 3.4.

: Martin XOR based Shared Block Cipher Encryption Computation

$$r \leftarrow RNG; Enc_d(K, m) \rightarrow (r, m \oplus \oplus_{i=1}^{\binom{n}{n-t+1}} Dec(K_i, r)) \tag{3.4}$$

: Martin XOR based Shared Block Cipher Decryption Computation

$$(r, c) \leftarrow ct; Dec_d(K, r, c) \rightarrow c \oplus \oplus_{i=1}^{\binom{n}{n-t+1}} Dec(K_i, r) \tag{3.5}$$

This provides a blinding effect similar to a one-time pad $\oplus$ operation to the plaintext m. The inverse operation shown in Equation 3.5 uses the random nonce created from the encryption and asks all participants to compute the Encryption of r with their key share thereby coming up with the value needed to un-blind the plaintext. Within Martin's scheme it is very important that the random nonce is indeed random. Within Martin's scheme, clearly, it is possible to create a block cipher from symmetric primitives allowing for a fully distributed solution.

Agrawal improved these schemes by adding the concept of identity authentication to the resulting ciphertexts by having computation of the distrubuted pseudo-random function performed on a commitment instead of just a random nonce value. This commitment is comprised of $\alpha = (m \oplus r)$ where m is the plaintext value and r is a random nonce. By storing this commitment with the ciphertext a decryption can be validated by performing $\alpha \oplus r\prime = ?m'$. The construction expressed in this work implements the distributed pseudo-random function definition from Agrawal's DISE work [2]. Within Appendix C in method *dprf* you can how we are performing the distributed pseudo-random function. Figure 3.2 expresses how the inter-participant communication is performed for encryptions and Figure 3.3 shows

the corresponding decryption. This process allows the primary participant the only view into the value of the plaintext. By not propagating the plaintext to other participants the plaintext need not be sent to other participants in the distributed computation.

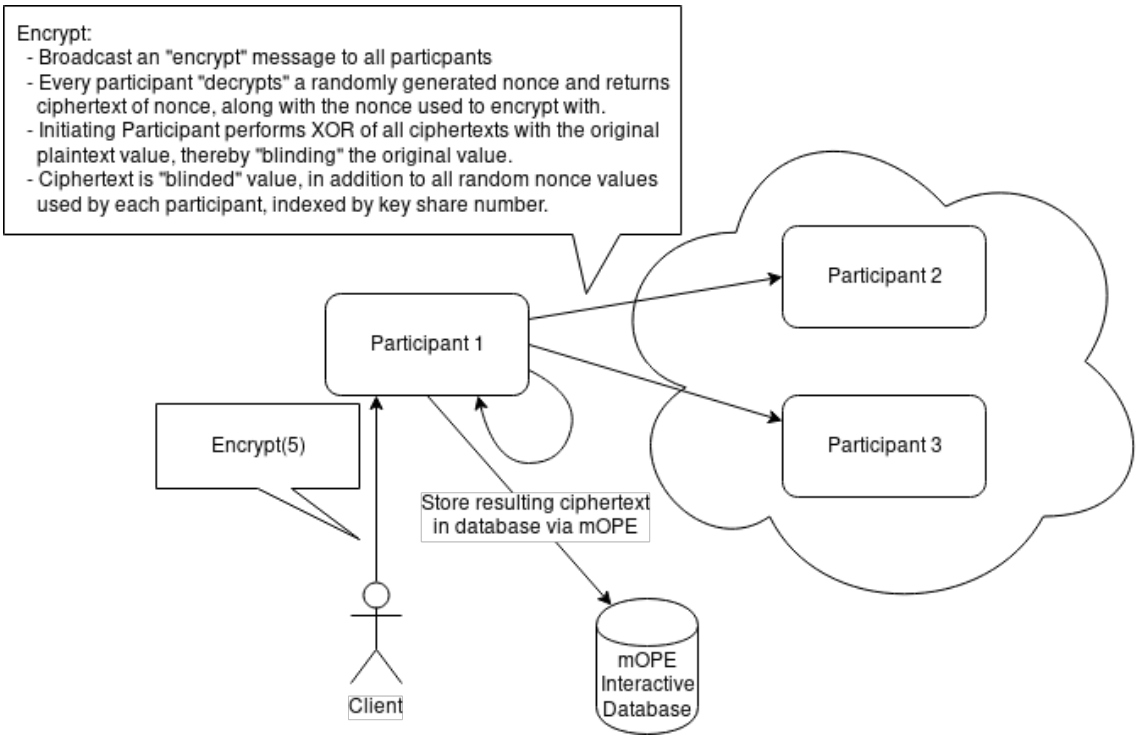


Figure 3.2: Visualization of itOPE Participant Encrypt and Store Process

Much as the dealer utilizes GRPC and TLS authenticated communication channels, so to do the participants. Each participant is issued a self signed certificate for authenticated secure communications. Upon registering with the dealer, the dealer will inform participants of their peers as seen in Appendix B. At this registration time, the participants create a long lived secure socket to their peers and inform their peers which keys they have.

itOPE Participant Integration with mOPE

Showcased within Chapter 2 the mOPE interactive protocol defined by Popa et al. [22] is a "remote" binary search tree, where the ordering comparison function is defined and performed within the client that has the encryption key, as opposed to on the server. The mOPE server implementation used within this research was written in Go in order to provide a side-by-side performance analysis and can be found in Appendix D. The binary

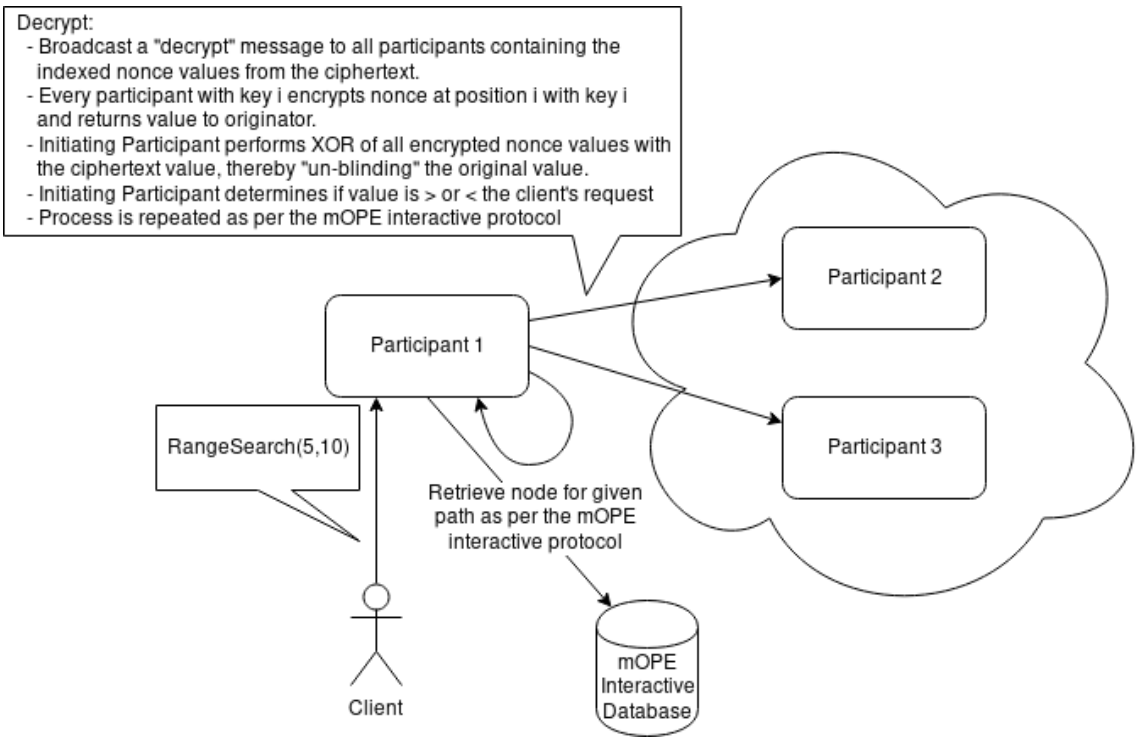


Figure 3.3: Visualization of itOPE Participant Range Search Process

tree data-structure which holds the encrypted data can be seen in Appendix E. Within the interactive protocol there are three primary functions, enumerated below.

1. *insert(path, ct)*
2. *delete(path)*
3. *retrieve(path)* $\rightarrow$ *ct*

The mOPE client as envisioned by Popa et al. [22] is visualized in Figure 3.4. Defined informally the mOPE interactive protocol is a remote binary search tree which allows the client to retain all authority in ordering of the ciphertexts within the tree. This binary search tree allows for easy in order traversal, but more pointedly, allows for a natural encoding scheme based on branching decisions. By requiring the client to manage all ordering operations within the interactive protocol the server learns absolutely about the plaintext when using semantically secure encryption, other than the ordering from the client through the protocol, and placement of ciphertexts within the search tree. Popa proves that this interactive protocol leaks nothing other than the ordering, and complies with IND-OCPA

[22].

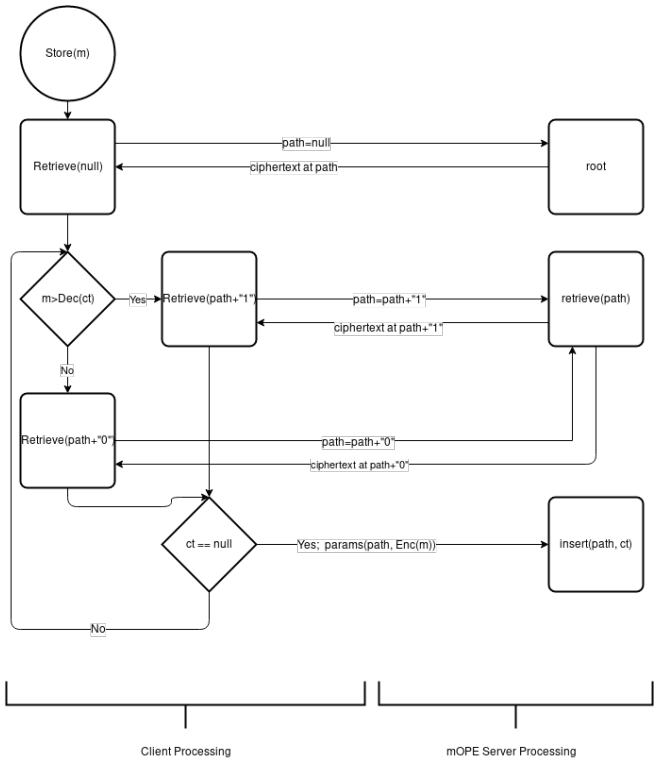


Figure 3.4: Visualization of mOPE Insertion Interactive Protocol

The order preserving encoding within Popa's scheme is a direct artifact of the binary search tree traversal, of which can be ingested directly into an existing database system with minor modification. By employing the mOPE interactive protocol within dOPE participant process it is realized that what is created is a distributed IND-OCPA order preserving solution. In terms of mOPE, the participant process is the "client" and performs all encryption and decryption using a shared block cipher.

As with the other components for this intrusion tolerant order preserving encryption scheme the interactive mOPE protocol is protected by secure sockets using TLS with client authentication through client certificates. Within this scheme each participant is responsible for maintaining an authenticated secure socket with the mOPE server and performing GRPC calls for insertions and traversals of the encrypted tree.

itOPE Infrastructure and Automation

The final construction provided consists of one dealer service, one mOPE service and several cooperative participant nodes. The construction requires t of n participants to successfully perform encryption and decryption in order to manipulate the mOPE data structure. As seen in Appendix F and Appendix G, for this construction we have automated the instantiation of these services using docker, and docker compose for orchestration.

Chapter 4

Intrusion Tolerant Order Preserving Encryption Performance Analysis

In order to test the performance of this intrusion tolerant order preserving encryption construction the code has been instrumented using a time series database solution prometheus. This testing was performed on a Dell XPS laptop with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor with 8MB cache and 16GB of RAM. Testing was performed on a vanilla mOPE implementation written in Go, of which the itOPE implementation was based. Testing was then performed on a two out of three threshold itOPE construction as well as a three out of five threshold itOPE construction. Key performance indicators included the rate of insertions per second into the mOPE server, rate of traversal requests per second to the mOPE server. On the itOPE construction metrics included number of distributed pseudo-random function calls on each participant per second as well as number of client requests per second.

Insertion Performance

In order to achieve a base line, best possible performance of itOPE, the mOPE implementation created was tested with inputs of factors of ten. One thousand random entries were inserted, then ten thousand, and finally one hundred thousand. The results are visualized within Figure 4.1. Between minute three and four in the graph we can see that the vanilla mOPE server was able to handle around 450 insertions per second. Between minute four and five it is shown that ten thousand were dispatched within 35 seconds. Finally one hundred thousand inputs were inserted into the mOPE database over the course of 8.5 minutes.

As can be expected by the performance of a binary search tree, we are able to realize a $log(n)$ performance degradation as the tree grows in size from more items being inserted into the tree.

With this performance metric realized, the testing then attempted the same test of insertions against the itOPE construction. As witnessed in Figure 4.2, the insertion rate realized that the mOPE server is half of what was seen with the vanilla mOPE client/server

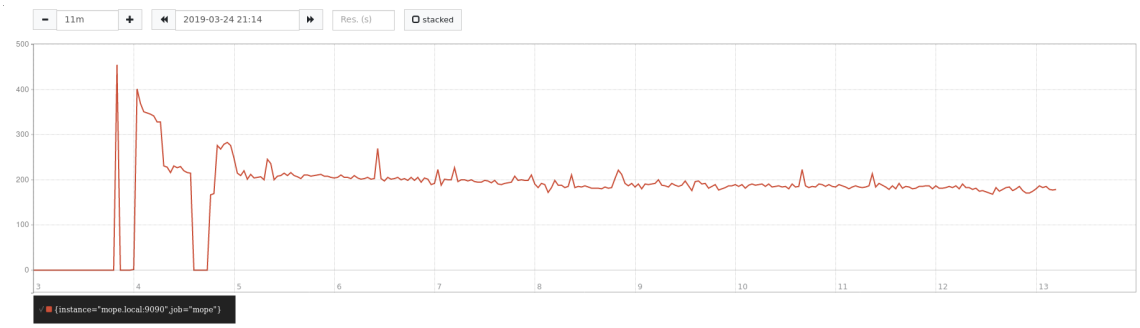


Figure 4.1: Insertion Performance (insertions/s) of Vanilla mOPE 1K, 10K, 100K

interaction. We can see clearly that the average insertion performance for itOPE is 160 insertions per second averaged for one thousand entries, then the next ten thousand averaged 70 insertions per second, and finally over the course of one hundred thousand insertions we realized 60 insertions per second. Comparing this to vanilla mOPE where we saw 477 insertions per second for one thousand, 300 insertions per second for ten thousand insertions and 175 insertions per second for one hundred thousand insertions.

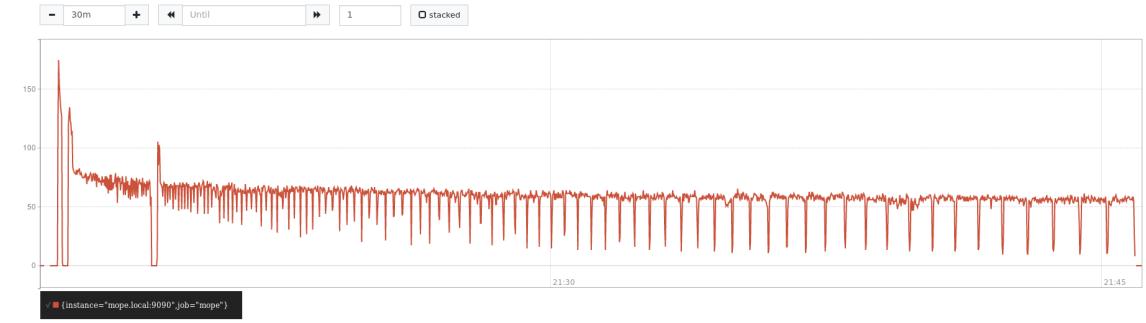


Figure 4.2: Insertion Performance (insertions/s) of itOPE 1K, 10K, 100K

Encryption/Decryption Performance

Figure 4.3 shows the number of distributed pseudo-random function initiations that were caused by the insertion requests during this test run. Each time a value needs to be encrypted or decrypted, as shown in Chapter 3, the participant handling the request is required to isolate which keys they are missing, and request from the other participants who have access to those missing keys to perform an encryption on the commitment. These results are then collated by the caller and exclusively-or-ed with the plaintext value for

encryption. Within this figure we can see the rate of requests to perform this distributed pseudo-random function access. It shows clearly that there are multiple requests to the distributed pseudo-random function access proportional to the number of requests that are being handled by the participant.

With the one thousand insertions range the participant averaged 1250 distributed pseudo-random function accesses per second. That equates to around 7 distributed pseudo-random function calls per one insertion request. This is directly due to the interactive mope protocol, as in order to perform an insertion, the client needs to traverse to the right location within the search tree.

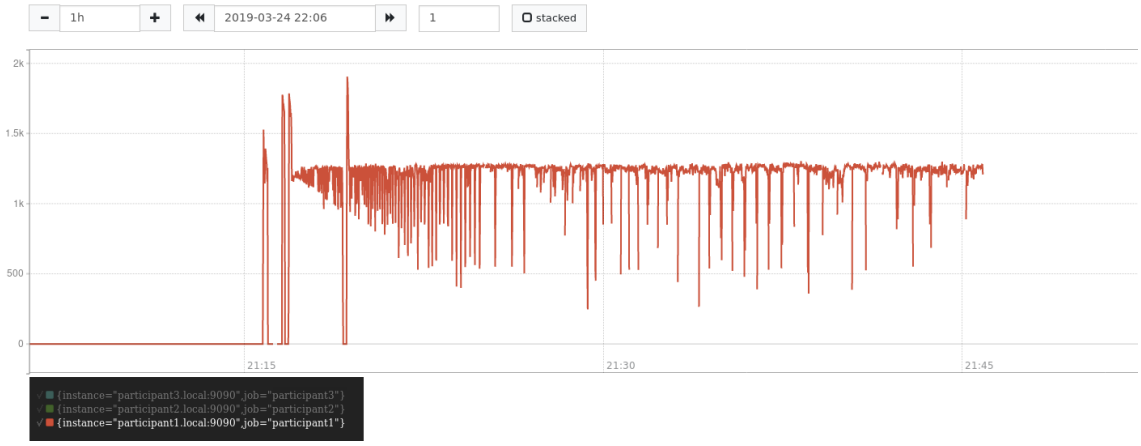


Figure 4.3: dPRF Initiations (dprf/s) of itOPE 1K, 10K, 100K

Since a collective of participants must work together to encrypt or decrypt a value, within Figure 4.4, it is shown the relative participant workload required to perform the distributed pseudo-random function. It is shown within this graph that one participant out of the three total (threshold 2 out of 3) does not need to participant in order to still allow encryptions and decryptions to succeed. Within the code implementation the author chose to always attempt to encrypt locally with a key that the current participant owns. Looking at this graph it is fairly clear that it would be more performant to allow the peers to compute the pseudo-random function with keys they have access to, instead of favoring the local key computation.

Within Figure 4.5 and Figure 4.6 we see that the graph of number of encryptions and decryptions performed is linear, though as stated before, the number of decryptions is much

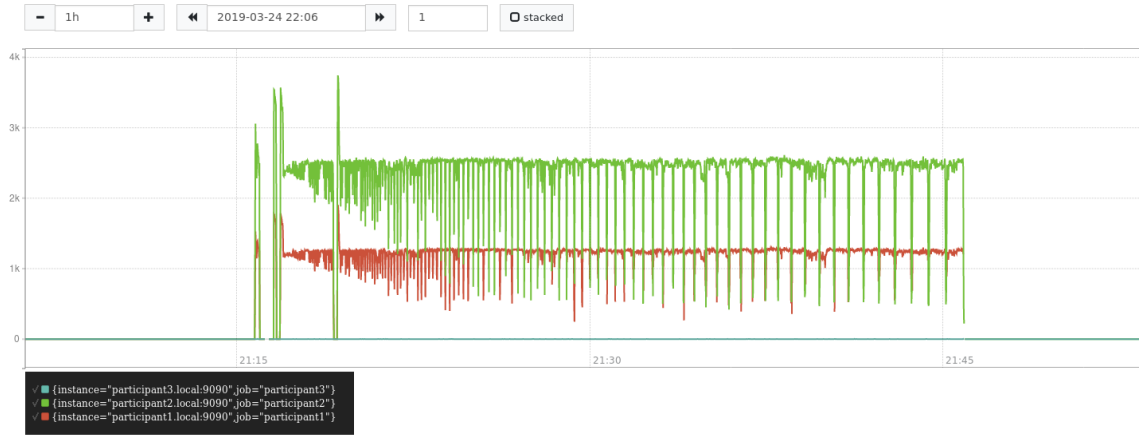


Figure 4.4: dPRF Computations (computations/s) of itOPE 1K, 10K, 100K

larger due to the interactive protocol. But these numbers follow closely to the asymptotic calculations based on the traversal/insertion cycle of a binary search tree which is to be expected.

Figure 4.5: Encryption Requests of itOPE 1K, 10K, 100K

Clearly there are some performance improvements that can affect the overall efficiency, such as not biasing toward local key usage. Overall, it is shown that an itOPE is linear in performance to a vanilla mOPE.

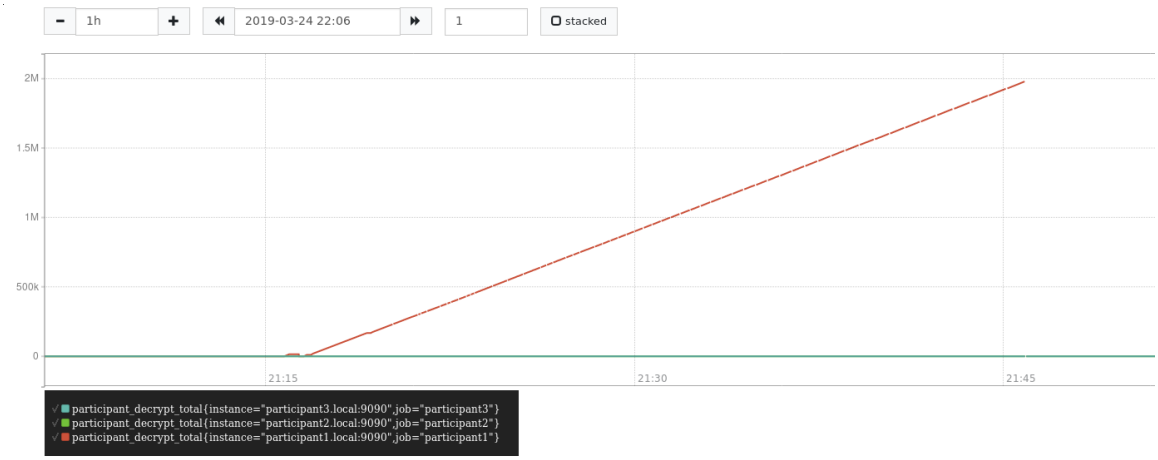


Figure 4.6: Decryptions Requests of itOPE 1K, 10K, 100K

Chapter 5

Conclusion

This work explains how to construct an intrusion and fault tolerant order preserving encryption scheme by modifying an existing scheme to use a distributed pseudo-random function. By employing an intrusion tolerant order preserving encryption scheme, practitioners are able to better safe guard encryption keys as well as provide more configurable access structures for participants to maintain ordered ciphertext values within a database. The implementation of itOPE, within Chapter 3, uses a secure distributed pseudo-random function which results in authenticated ciphertext values which can be validated at decryption time. Said implementation does not reconstruct the full encryption key and therefore is designed such that any particular node can be compromised without compromising the entirety of the database.

Based on the analysis performed in Chapter 4 of the proposed scheme, it is shown that the overhead incurred by itOPE is asymptotically linear to the number of participants regarding additional network overhead. The additional overhead is the point to point communications between the participant nodes during the distributed pseudo-random function calculation. It is also shown in analysis that the storage overhead in the ciphertext is linear, as the resulting ciphertext is two times the size of the ciphertext value in the unmodified scheme. Encryption processing time is also linear where the additional required encryptions are $k * \binom{n}{n-t}$ where k is the number of plaintexts to be encrypted, n is the number of participants and t is the number of participants required.

Appendix A

Key Sharing Dealer Source Code

src/dealer/dealer.go

```
1  package dealer
2
3  import (
4   "crypto/aes"
5   "crypto/rand"
6   "fmt"
7
8   "github.com/pkg/errors"
9  )
10
11 // KeyPart - a key shadow which is distributed
12 type KeyPart struct {
13  ID     int
14  Value  []byte
15 }
16
17 // GenerateKeys - based on (t, n) threshold compute a list of shadows
18 func GenerateKeys(n, t int) ([]KeyPart, error) {
19  var (
20   // number of keys required based on n and t
21   d = choose(n, n-t)
22   // the list of keyshares to be distributed
23   shadows = make([]KeyPart, d)
24  )
25  // create d key parts
26  for i := 0; i < d; i++ {
27   shadows[i].ID = i + 1
28   shadows[i].Value = make([]byte, aes.BlockSize)
29   _, err := rand.Read(shadows[i].Value)
30   if err != nil {
31    return nil, errors.Wrap(err, "failed to make keys")
32   }
33  }
34  return shadows, nil
35 }
36
37 // KeyAssignment - based on the id of the participant produce a list
38 // of keys that participant should get
39 func KeyAssignment(pid, n, t int, as [][]int, shadows []KeyPart) ([]KeyPart,
       error) {
40
41  if pid > n {
42   return nil, errors.New("invalid participant id")
43  }
44
45  // if pid is in a particular access structure, then give it that key
46  var (
```

```
47    d      = choose(n, n-t)
48    keys   = []KeyPart{}
49    count = 0
50   )
51
52   // for each access group
53   for j := 0; j < d; j++ {
54    // if this participant is a member of access group j
55    // give that participant key j
56    for k := 0; k < t; k++ {
57     if pid == as[j][k] {
58      keys = append(keys, shadows[j])
59      count++
60     }
61    }
62   }
63
64   return keys, nil
65  }
66
67  // choose - helper to perform combination calcuations
68  func choose(n, m int) int {
69   var (
70    factN  int = 1
71    factM  int = 1
72    factNM int = 1
73   )
74   for i := 1; i <= n; i++ {
75    factN = factN * i
76   }
77   for i := 1; i <= m; i++ {
78    factM = factM * i
79   }
80   for i := 1; i <= (n - m); i++ {
81    factM = factM * i
82   }
83   return factN / (factM * factNM)
84  }
85
86  // ComputeAS - takes in (t, n) and generates a generic access structure
87  func ComputeAS(t, n int) [][]int {
88   var (
89    participants = make([]int, n)
90    tmp          = make([]int, t)
91    result       = make([][]int, choose(n, n-t))
92   )
93
94   fmt.Println("d = ", choose(n, n-t))
95   // initialize result
96   for i := 0; i < choose(n, n-t); i++ {
97    result[i] = make([]int, t)
98   }
99   // initialize participants
100  for i := 0; i < n; i++ {
101   participants[i] = i + 1
102  }
103
104  var count int
105  // recursively compute the access structures
```

```
106   computeAS(t, 0, n-1, 0, participants, tmp, result, &count)
107
108   return result
109  }
110
111  // computeAS - recursive algorithm for figuring out access structures
112  func computeAS(t, start, end, index int, participants, tmp []int, result [][]
         int, count *int) {
113   if index == t {
114    for i, v := range tmp {
115     result[*count][i] = v
116    }
117    *count = *count + 1
118    return
119   }
120   for i := start; i <= end && end-i+1 >= t-index; i++ {
121    tmp[index] = participants[i]
122    computeAS(t, i+1, end, index+1, participants, tmp, result, count)
123   }
124  }
```

Appendix B

Key Sharing Dealer Server Source Code

src/dealer/server/main.go

```
1  package main
2
3  import (
4   "context"
5   "crypto/tls"
6   "crypto/x509"
7   "flag"
8   "io/ioutil"
9   "log"
10  "net"
11  "net/http"
12  "sync"
13  "time"
14
15  "dealer"
16  pb "dealer/protobuf"
17
18  "google.golang.org/grpc"
19  "google.golang.org/grpc/credentials"
20
21  "github.com/prometheus/client_golang/prometheus/promhttp"
22 )
23
24 type server struct {
25  ParticipantKeyAssignments [][]dealer.KeyPart
26  pkaMu                     sync.RWMutex
27  pkaNum                    int
28  Participants              []*pb.Peer
29 }
30
31 // Register implements protobuf.DealerService, this is the registration logic
32 func (s *server) Register(ctx context.Context, in *pb.RegisterRequest) (*pb.
       RegisterResponse, error) {
33  log.Printf("Received Registration Request: %v", in.Self)
34  s.pkaMu.Lock()
35  defer s.pkaMu.Unlock()
36  log.Printf("Assigning Caller ID: %d", s.pkaNum)
37
38  var keys = []*pb.Key{}
39  // add this peer to our list of peers we give participants
40  s.Participants = append(s.Participants, in.Self)
41
42  // convert our key to the response structure
43  for _, k := range s.ParticipantKeyAssignments[s.pkaNum] {
44   keys = append(keys, &pb.Key{
45    Id:    int32(k.ID),
46    Value: k.Value,
```

```
47      })
48    }
49
50    resp := &pb.RegisterResponse{
51      Message: "Success", Success: true,
52      KeyParts: keys, KnownPeers: s.Participants,
53    }
54
55    s.pkaNum++
56
57    return resp, nil
58  }
59
60  var (
61    n int
62    t int
63
64    // for tls authentication and confidentiality
65    serverCert string
66    serverKey  string
67
68    clientCert string
69    clientKey  string
70
71    caCert string
72
73    addr string
74  )
75
76  func init() {
77    flag.IntVar(&n, "n", 5, "Number of total Participants, default=5")
78    flag.IntVar(&t, "t", 3, "Number of Participants required to decrypt, default
        =3")
79
80    flag.StringVar(&serverCert, "sCrt", "", "Server TLS Cert")
81    flag.StringVar(&serverKey, "sKey", "", "Server TLS Key")
82    flag.StringVar(&clientCert, "cCrt", "", "Client TLS Cert")
83    flag.StringVar(&clientKey, "cKey", "", "Client TLS Key")
84    flag.StringVar(&caCert, "caCrt", "", "CA TLS Cert")
85
86    flag.StringVar(&addr, "addr", ":1234", "Server Address")
87
88    flag.Parse()
89  }
90
91  // main - main entrypoint for the dealer server
92  func main() {
93
94    go func() {
95      http.Handle("/metrics", promhttp.Handler())
96      log.Fatal(http.ListenAndServe(":9090", nil))
97    }()
98
99    // load in local CA
100   certificate, err := tls.LoadX509KeyPair(serverCert, serverKey)
101   if err != nil {
102     log.Fatalf("failed to load certs: %s\n", err.Error())
103   }
104
```

```
105   certPool := x509.NewCertPool()
106   ca, err := ioutil.ReadFile(caCert)
107   if err != nil {
108    log.Fatalf("failed to load certs: %s\n", err.Error())
109   }
110
111   if ok := certPool.AppendCertsFromPEM(ca); !ok {
112    log.Fatalf("failed to load certs: %s\n", err.Error())
113   }
114
115   // Dealer setup
116   keys, err := dealer.GenerateKeys(n, t)
117   if err != nil {
118    log.Fatalf("err: %s\n", err.Error())
119   }
120
121   as := dealer.ComputeAS(t, n)
122   log.Printf("%+v", as)
123
124   participantKeys := make([][]dealer.KeyPart, n)
125   // figure out who gets what key
126   for i := 1; i <= n; i++ {
127    keys, err := dealer.KeyAssignment(i, n, t, as, keys)
128    if err != nil {
129     log.Fatalf("err: %s\n", err.Error())
130    }
131    participantKeys[i-1] = append([]dealer.KeyPart{}, keys...)
132   }
133
134   log.Printf("%+v", participantKeys)
135
136   lis, err := net.Listen("tcp", addr)
137   if err != nil {
138    log.Fatalf("failed to listen: %v", err)
139   }
140
141   // require client credentials
142   creds := credentials.NewTLS(&tls.Config{
143    ClientAuth:    tls.RequireAndVerifyClientCert,
144    Certificates: []tls.Certificate{certificate},
145    ClientCAs:     certPool,
146   })
147
148   grpcServer := grpc.NewServer(grpc.Creds(creds))
149
150   var s = &server{
151    ParticipantKeyAssignments: participantKeys,
152    pkaMu:                     sync.RWMutex{},
153    pkaNum:                    0,
154   }
155
156   pb.RegisterDealerServiceServer(grpcServer, s)
157
158   // shutdown the server if we find we have given all the keys away
159   go func(s *server) {
160    for {
161     time.Sleep(5 * time.Second)
162     s.pkaMu.Lock()
163     if s.pkaNum >= len(s.ParticipantKeyAssignments) {
```

```
164       log.Fatalf("Dealer has dealt all of the keys to participants, exiting.")
165      }
166      s.pkaMu.Unlock()
167     }
168    }(s)
169
170    if err := grpcServer.Serve(lis); err != nil {
171     log.Fatalf("failed to serve: %v", err)
172    }
173   }
```

Appendix C

Participant Server Source Code

src/participant/server/main.go

```
1  package main
2
3  import (
4   "bytes"
5   "context"
6   "crypto/aes"
7   "crypto/rand"
8   "crypto/tls"
9   "crypto/x509"
10  "encoding/binary"
11  "encoding/gob"
12  "flag"
13  "fmt"
14  "io/ioutil"
15  "log"
16  "net"
17  "net/http"
18  "strings"
19  "sync"
20  "time"
21
22  dealerpb "dealer/protobuf"
23  mopepb "mope/protobuf"
24  pb "participant/protobuf"
25
26  "github.com/pkg/errors"
27  "github.com/prometheus/client_golang/prometheus"
28  "github.com/prometheus/client_golang/prometheus/promauto"
29  "github.com/prometheus/client_golang/prometheus/promhttp"
30  "google.golang.org/grpc"
31  "google.golang.org/grpc/credentials"
32 )
33
34 var (
35  // for tls authentication and confidentiality
36  serverCert string
37  serverKey  string
38
39  clientCert string
40  clientKey  string
41
42  caCert string
43
44  addr       string
45  dealerAddr string
46  mopeAddr   string
47 )
```

```
48
49 type server struct {
50  id           int
51  Keys         []*dealerpb.Key
52  Peers        []*pb.Peer
53  MopeClient   mopepb.MopeServiceClient
54  PeerClients  []pb.ParticipantServiceClient
55  pcMu         sync.Mutex
56  clientCert   tls.Certificate
57  certPool     *x509.CertPool
58 }
59
60 func (s *server) GetMissingKeys() []int32 {
61  localKeys := map[int32]struct{}{}
62  tmp := map[int32]struct{}{}
63  ret := []int32{}
64
65  for _, v := range s.GetSelfKeys() {
66   localKeys[v.Id] = struct{}{}
67  }
68
69  for _, v := range s.Peers {
70   // should we add any of this peer's keys
71   for _, kid := range v.Keys {
72    // should we add this key?
73    _, seenPeer := tmp[kid]
74    _, seenLocal := localKeys[kid]
75    if !seenLocal && !seenPeer {
76     tmp[kid] = struct{}{}
77    }
78   }
79  }
80  for k, _ := range tmp {
81   ret = append(ret, k)
82  }
83
84  return ret
85 }
86
87 func (s *server) GetSelfKeys() []*dealerpb.Key {
88  return s.Keys
89 }
90
91 func (s *server) dprf(w []byte) []byte {
92  prfCtr.Inc()
93  timer := prometheus.NewTimer(prfDuration)
94  defer timer.ObserveDuration()
95
96  var results = make(chan []*pb.DPRFValue)
97  for _, c := range s.PeerClients {
98   go func(w []byte) {
99    resp, err := c.ComputeDPRF(context.Background(), &pb.ComputeDPRFRequest{
100    W: w,
101    })
102    if err != nil {
103     log.Printf("failed to compute dprf: %s\n", err.Error())
104    }
105    //log.Println("here we go, got back: ", resp.Values)
106    results <- resp.Values
```

```
107     }(w)
108    }
109    go func(w []byte) {
110     resp, err := s.ComputeDPRF(context.Background(), &pb.ComputeDPRFRequest{
111      W: w,
112     })
113     if err != nil {
114      log.Printf("failed to compute dprf: %s\n", err.Error())
115     }
116     //log.Println("here we go, computed locally: ", resp.Values)
117     results <- resp.Values
118    }(w)
119
120    totalKeys := len(s.GetMissingKeys()) + len(s.GetSelfKeys())
121    keysFound := map[int32]struct{}{}
122    values := [][]byte{}
123
124    for i := 1; len(keysFound) < totalKeys; i++ {
125     //log.Println("len(keysFound): ", len(keysFound))
126     //log.Println("totalKeys: ", totalKeys)
127     // wait for results
128     //log.Println("waiting for all key results to roll in: ", i)
129     result := <-results
130     for _, r := range result {
131      // if we havent seen this key, append the value
132      if _, ok := keysFound[r.KeyUsed]; !ok {
133       keysFound[r.KeyUsed] = struct{}{}
134       values = append(values, r.Value)
135      }
136     }
137    }
138    ret := []byte{0x0}
139    for _, v := range values {
140     ret = xor(ret, v)
141    }
142    return ret
143   }
144
145   func encode(p interface{}) []byte {
146    var buf bytes.Buffer
147    enc := gob.NewEncoder(&buf)
148    enc.Encode(p)
149    b := buf.Bytes()
150    return b
151   }
152
153   func decode(p []byte, v interface{}) {
154    var buf = bytes.NewBuffer(p)
155    dec := gob.NewDecoder(buf)
156    dec.Decode(v)
157   }
158
159   func xor(p ...[]byte) []byte {
160    // figure out the length of v(s)
161    var length = 0
162    for _, v := range p {
163     if len(v) > length {
164      length = len(v)
165     }
```

```
166   }
167
168   for i := 0; i < len(p); i++ {
169    for j := length - len(p[i]); j > 0; j-- {
170     p[i] = append(p[i], 0x0)
171    }
172   }
173   var ret = make([]byte, length)
174
175   for _, v := range p {
176    for i, _ := range ret {
177     ret[i] = ret[i] ^ v[i]
178    }
179   }
180   return ret
181  }
182
183  func (s *server) Print() {
184   for {
185    time.Sleep(5 * time.Second)
186    fmt.Printf("Peer connections: ")
187    for _, v := range s.PeerClients {
188     fmt.Printf("%v, ", v)
189    }
190    fmt.Printf("\nPeers: ")
191    for _, v := range s.Peers {
192     fmt.Printf("%v, ", v)
193    }
194    fmt.Printf("\n")
195   }
196  }
197
198  // Hello implements protobuf.ParticipantService, this is the hello logic
199  func (s *server) Hello(ctx context.Context, in *pb.HelloRequest) (*pb.
        HelloResponse, error) {
200   log.Printf("Received: %v", in)
201   // we need to make a peerClient for this caller
202   creds := credentials.NewTLS(&tls.Config{
203    ServerName:   strings.Split(in.Self.Address, ":")[0],
204    Certificates: []tls.Certificate{s.clientCert},
205    RootCAs:      s.certPool,
206   })
207   conn, err := grpc.Dial(in.Self.Address, grpc.WithTransportCredentials(creds))
208   if err != nil {
209    return nil, errors.Wrap(err, "failed to connect to peer")
210   }
211   s.pcMu.Lock()
212   s.Peers = append(s.Peers, &pb.Peer{Address: in.Self.Address, Keys: in.Self.
        Keys})
213   s.PeerClients = append(s.PeerClients, pb.NewParticipantServiceClient(conn))
214   s.pcMu.Unlock()
215
216   var keys = []int32{}
217   for _, v := range s.Keys {
218    keys = append(keys, v.Id)
219   }
220
221   var peers = append(s.Peers, &pb.Peer{
222    Address: addr, Keys: keys,
```

```
223   })
224
225   return &pb.HelloResponse{Success: true, Message: "Success", Peers: peers},
          nil
226 }
227
228 func nopEnc(k, m []byte) (ct []byte, err error) {
229   ct = m
230   return
231 }
232
233 var (
234   encCtr = promauto.NewCounter(prometheus.CounterOpts{
235     Name: "participant_encrypt_total",
236   })
237   decCtr = promauto.NewCounter(prometheus.CounterOpts{
238     Name: "participant_decrypt_total",
239   })
240   prfCtr = promauto.NewCounter(prometheus.CounterOpts{
241     Name: "participant_prf_total",
242   })
243   computePRFCtr = promauto.NewCounter(prometheus.CounterOpts{
244     Name: "participant_compute_prf_total",
245   })
246   getRangeCtr = promauto.NewCounter(prometheus.CounterOpts{
247     Name: "participant_get_range_total",
248   })
249   storeCtr = promauto.NewCounter(prometheus.CounterOpts{
250     Name: "participant_store_total",
251   })
252
253   encDuration = prometheus.NewHistogram(prometheus.HistogramOpts{
254     Name:     "participant_encrypt_duration",
255     Buckets: prometheus.ExponentialBuckets(0.1, 1.5, 5),
256   })
257   decDuration = prometheus.NewHistogram(prometheus.HistogramOpts{
258     Name:     "participant_decrypt_duration",
259     Buckets: prometheus.ExponentialBuckets(0.1, 1.5, 5),
260   })
261   prfDuration = prometheus.NewHistogram(prometheus.HistogramOpts{
262     Name:     "participant_prf_duration",
263     Buckets: prometheus.ExponentialBuckets(0.1, 1.5, 5),
264   })
265 )
266
267 func (s *server) encrypt(pt []byte) ([]byte, error) {
268   encCtr.Inc()
269   timer := prometheus.NewTimer(encDuration)
270   defer timer.ObserveDuration()
271
272   // create a random number
273   var rho = make([]byte, 2*aes.BlockSize)
274   _, err := rand.Read(rho)
275   if err != nil {
276     return nil, errors.Wrap(err, "failed to make keys")
277   }
278
279   // convert id to binary
280   var id = make([]byte, 8)
```

```
281   n := binary.PutUvarint(id, uint64(s.id))
282
283   var (
284    alpha = xor(rho, pt)
285    w     = encode(&wparam{id[:n], alpha})
286    m     = encode(&plaintext{pt, rho})
287    prf   = xor(s.dprf(w), m)
288   )
289
290   ct := &ciphertext{prf, id[:n], alpha}
291   return encode(ct), nil
292  }
293
294  type wparam struct {
295   ID    []byte
296   Alpha []byte
297  }
298
299  type ciphertext struct {
300   PRF   []byte
301   ID    []byte
302   Alpha []byte
303  }
304
305  type plaintext struct {
306   M   []byte
307   Rho []byte
308  }
309
310  func (s *server) decrypt(ct []byte) ([]byte, error) {
311   decCtr.Inc()
312   timer := prometheus.NewTimer(decDuration)
313   defer timer.ObserveDuration()
314
315   var ctdec = new(ciphertext)
316   decode(ct, ctdec)
317
318   var (
319    prf   = ctdec.PRF
320    id    = ctdec.ID
321    alpha = ctdec.Alpha
322    m     = xor(s.dprf(encode(&wparam{id, alpha})), prf)
323    pt    = new(plaintext)
324   )
325   decode(m, pt)
326
327   // check integrity
328   // ptParts[0] is pt, ptParts[1] is rho
329   if bytes.Compare(xor(pt.Rho, alpha)[:len(pt.M)], pt.M) != 0 {
330    return nil, errors.New("unauthenticated ciphertext")
331   }
332
333   return pt.M, nil
334  }
335
336  // Store implements protobuf.ParticipantService, this is the store logic
337  func (s *server) Store(ctx context.Context, in *pb.StoreRequest) (*pb.
       StoreResponse, error) {
338   log.Printf("Received: %v", in.Value)
```

```
339   storeCtr.Inc()
340   s.MopeInsert(in.Value)
341   return &pb.StoreResponse{Success: true, Message: "Success!"}, nil
342  }
343
344  // GetRange implements protobuf.ParticipantService, this is the query logic
345  func (s *server) GetRange(ctx context.Context, in *pb.GetRangeRequest) (*pb.
         GetRangeResponse, error) {
346   getRangeCtr.Inc()
347   log.Printf("Received: %v, %v", in.A, in.B)
348   aTag, err := s.MopeFindTag(in.A)
349   if err != nil {
350    return &pb.GetRangeResponse{Success: false, Message: err.Error()}, nil
351   }
352   bTag, err := s.MopeFindTag(in.B)
353   if err != nil {
354    return &pb.GetRangeResponse{Success: false, Message: err.Error()}, nil
355   }
356   return &pb.GetRangeResponse{Success: true, Message: "Success!", StartTag:
         aTag, EndTag: bTag}, nil
357  }
358
359  // ComputeDPRF implements protobuf.ParticipantService, this is the compute d-
         prf logic
360  func (s *server) ComputeDPRF(ctx context.Context, in *pb.ComputeDPRFRequest)
         (*pb.ComputeDPRFResponse, error) {
361   log.Printf("Received: %v", in)
362   computePRFCtr.Inc()
363   // perform encryption on a randomly generated nonce
364   var dprfValues = []*pb.DPRFValue{}
365   for _, v := range s.Keys {
366    //log.Println("doing the computation here, key: ", v.Id)
367    dprfValues = append(dprfValues, &pb.DPRFValue{
368     KeyUsed: v.Id,
369     Value:    in.W,
370    })
371   }
372
373   return &pb.ComputeDPRFResponse{Success: true, Message: "Success!", Values:
         dprfValues}, nil
374  }
375
376  func init() {
377   flag.StringVar(&serverCert, "sCrt", "", "Server TLS Cert")
378   flag.StringVar(&serverKey, "sKey", "", "Server TLS Key")
379   flag.StringVar(&clientCert, "cCrt", "", "Client TLS Cert")
380   flag.StringVar(&clientKey, "cKey", "", "Client TLS Key")
381   flag.StringVar(&caCert, "caCrt", "", "CA TLS Cert")
382
383   flag.StringVar(&addr, "addr", ":1234", "Server Address")
384   flag.StringVar(&dealerAddr, "dealer", "dealer.local:8080", "Dealer Server
         Address")
385   flag.StringVar(&mopeAddr, "mope", "mope.local:8080", "mOPE Server Address")
386
387   flag.Parse()
388
389   gob.Register(new(plaintext))
390   gob.Register(new(ciphertext))
391   gob.Register(new(wparam))
```

```
392 }
393
394 // main - main entrypoint for the participant server
395 func main() {
396  go func() {
397   http.Handle("/metrics", promhttp.Handler())
398   log.Fatal(http.ListenAndServe(":9090", nil))
399  }()
400  // load in local CA
401  certificate, err := tls.LoadX509KeyPair(serverCert, serverKey)
402  if err != nil {
403   log.Fatalf("failed to load certs: %s\n", err.Error())
404  }
405
406  certPool := x509.NewCertPool()
407  ca, err := ioutil.ReadFile(caCert)
408  if err != nil {
409   log.Fatalf("failed to load certs: %s\n", err.Error())
410  }
411
412  if ok := certPool.AppendCertsFromPEM(ca); !ok {
413   log.Fatalf("failed to load certs: %s\n", err.Error())
414  }
415
416  lis, err := net.Listen("tcp", addr)
417  if err != nil {
418   log.Fatalf("failed to listen: %v", err)
419  }
420
421  // require client credentials
422  creds := credentials.NewTLS(&tls.Config{
423   ClientAuth:   tls.RequireAndVerifyClientCert,
424   Certificates: []tls.Certificate{certificate},
425   ClientCAs:    certPool,
426  })
427
428  // require client credentials
429  dealerClientCreds := credentials.NewTLS(&tls.Config{
430   ServerName:   strings.Split(dealerAddr, ":")[0],
431   Certificates: []tls.Certificate{certificate},
432   RootCAs:      certPool,
433  })
434
435  // fetch the keys/peers from dealer through registration
436
437  var (
438   id, peers, keys = dealerRegister(dealerClientCreds, addr, dealerAddr)
439  )
440
441  //log.Printf("Peers: %v", peers)
442  var keystr string
443  for _, v := range keys {
444   keystr += fmt.Sprintf("%d, ", v.Id)
445  }
446
447  //log.Printf("Keys: %s", keystr)
448
449  // setup initial peer clients we know about
450  var peerClients = []pb.ParticipantServiceClient{}
```

```
451
452   for _, v := range peers {
453    // we need to make a peerClient for this caller
454    creds := credentials.NewTLS(&tls.Config{
455     ServerName:   strings.Split(v.Address, ":")[0],
456     Certificates: []tls.Certificate{certificate},
457     RootCAs:      certPool,
458    })
459    conn, err := grpc.Dial(v.Address, grpc.WithTransportCredentials(creds))
460    if err != nil {
461     log.Fatalf("failed to connect to peer")
462    }
463    var c = pb.NewParticipantServiceClient(conn)
464    peerClients = append(peerClients, c)
465
466    var k = []int32{}
467    for _, v := range keys {
468     k = append(k, v.Id)
469    }
470
471    resp, err := c.Hello(context.Background(), &pb.HelloRequest{
472     Self: &pb.Peer{Address: addr, Keys: k},
473    })
474
475    // for all the returned peers, we should add them to our server
476    for _, v := range resp.Peers {
477     if !strings.HasPrefix(v.Address, addr) {
478
479      var found = false
480      for i, vv := range peers {
481       if vv.Address == v.Address {
482        peers[i] = v
483        found = true
484       }
485      }
486      if !found {
487       peers = append(peers, v)
488
489       creds := credentials.NewTLS(&tls.Config{
490        ServerName:   strings.Split(v.Address, ":")[0],
491        Certificates: []tls.Certificate{certificate},
492        RootCAs:      certPool,
493       })
494       conn, err := grpc.Dial(v.Address, grpc.WithTransportCredentials(creds))
495       if err != nil {
496        log.Fatalf("failed to connect to peer")
497       }
498       var c = pb.NewParticipantServiceClient(conn)
499       peerClients = append(peerClients, c)
500      }
501     }
502    }
503
504    //log.Printf("Received from peer: %v, %v", resp, err)
505   }
506
507   // require client credentials
508   clientCreds := credentials.NewTLS(&tls.Config{
509    ServerName:   strings.Split(mopeAddr, ":")[0],
```

```
510     Certificates: []tls.Certificate{certificate},
511     RootCAs:      certPool,
512    })
513
514    conn, err := grpc.Dial(mopeAddr, grpc.WithTransportCredentials(clientCreds))
515    if err != nil {
516     log.Fatalf("could not connect to mope: %s\n", err.Error())
517    }
518    mopeClient := mopepb.NewMopeServiceClient(conn)
519
520    grpcServer := grpc.NewServer(grpc.Creds(creds))
521    var s = &server{
522     id:          int(id),
523     PeerClients: peerClients,
524     pcMu:        sync.Mutex{},
525     clientCert:  certificate,
526     certPool:    certPool,
527     Peers:       peers, Keys: keys, MopeClient: mopeClient,
528    }
529    go s.Print()
530    pb.RegisterParticipantServiceServer(grpcServer, s)
531
532    if err := grpcServer.Serve(lis); err != nil {
533     log.Fatalf("failed to serve: %v", err)
534    }
535   }
536
537   func dealerRegister(clientCreds credentials.TransportCredentials, addr,
        dealerAddr string) (int32, []*pb.Peer, []*dealerpb.Key) {
538    conn, err := grpc.Dial(dealerAddr, grpc.WithTransportCredentials(clientCreds)
        )
539    if err != nil {
540     log.Fatalf("could not connect to dealer: %s\n", err.Error())
541    }
542    c := dealerpb.NewDealerServiceClient(conn)
543    resp, err := c.Register(context.Background(), &dealerpb.RegisterRequest{
544     Self: &dealerpb.Peer{
545      Address: addr,
546     },
547    })
548    if err != nil {
549     log.Fatalf("failed to get key parts from dealer: %s\n", err.Error())
550    }
551
552    peers := []*pb.Peer{}
553    for _, v := range resp.KnownPeers {
554     if !strings.HasPrefix(addr, v.Address) {
555      peers = append(peers, &pb.Peer{
556       Address: v.Address,
557      })
558     }
559    }
560
561    return resp.Id, peers, resp.KeyParts
562   }
563
564   // MopeInsert - takes a value, key and encryption function
565   func (s *server) MopeInsert(m []byte) error {
566    var (
```

```
567     path, err = s.MopeFindTag(m)
568     ct, _     = s.encrypt(m)
569    )
570
571    if err != nil {
572     return errors.Wrap(err, "failed insertion")
573    }
574
575    // insert into the path location
576    resp, err := s.MopeClient.Insert(context.Background(), &mopepb.InsertRequest{
577     Path: path,
578     Data: ct,
579    })
580    if err != nil {
581     return errors.Wrap(err, "failed insertion")
582    }
583    if !resp.Success {
584     return errors.New(resp.Message)
585    }
586    return nil
587   }
588
589   // MopeFindTag - takes a value, and figures out the tag of said value in the
        tree, for searching
590   func (s *server) MopeFindTag(m []byte) (string, error) {
591    // get root
592    // path will be the running path in the BST
593    path := ""
594    for {
595     //fmt.Println("path value: ", path)
596     resp, err := s.MopeClient.Traverse(context.Background(), &mopepb.
           TraverseRequest{
597      Path: path,
598     })
599     if err != nil {
600      log.Println("traverse error: ", err.Error())
601      return "", errors.Wrap(err, "failed to traverse for insert")
602     }
603
604     if resp.Data == nil || len(resp.Data) == 0 {
605      // done
606      log.Println("data is nil, reached the end")
607      break
608     }
609
610     mP, err := s.decrypt(resp.Data)
611     if err != nil {
612      log.Println("decrypt error")
613      return "", errors.Wrap(err, "failed to decrypt for insert")
614     }
615
616     var done bool
617     //  0 if a==b, -1 if a < b, and +1 if a > b.
618     switch bytes.Compare(m, mP) {
619     case 0:
620      // this is the path to use, break outer
621      //fmt.Println("value is equal.")
622      done = true
623      break
```

```
624     case -1:
625      // go left
626      //fmt.Println("value is less than.")
627      path += "0"
628     case 1:
629      // go right
630      //fmt.Println("value is greater than.")
631      path += "1"
632     }
633     if done {
634      break
635     }
636    }
637    return path, nil
638   }
```

Appendix D

mOPE Server Source Code

src/mope/server/main.go

```
1  package main
2
3  import (
4   "context"
5   "crypto/tls"
6   "crypto/x509"
7   "flag"
8   "io/ioutil"
9   "log"
10  "mope"
11  "net"
12  "net/http"
13  "sync"
14
15  pb "mope/protobuf"
16
17  "github.com/prometheus/client_golang/prometheus"
18  "github.com/prometheus/client_golang/prometheus/promauto"
19  "github.com/prometheus/client_golang/prometheus/promhttp"
20  "google.golang.org/grpc"
21  "google.golang.org/grpc/credentials"
22 )
23
24 var (
25  mopeInsCtr = promauto.NewCounter(prometheus.CounterOpts{
26   Name: "mope_insertion_total",
27   Help: "The total number of mope insertions",
28  })
29  mopeTravCtr = promauto.NewCounter(prometheus.CounterOpts{
30   Name: "mope_traverse_total",
31   Help: "The total number of mope traversals",
32  })
33 )
34
35 // server is used to implement helloworld.GreeterServer.
36 type server struct {
37  mMu      sync.RWMutex
38  mopeRoot *mope.Node
39 }
40
41 // Insert implements protobuf.MopeService, this is the insertion logic
42 func (s *server) Insert(ctx context.Context, in *pb.InsertRequest) (*pb.
       InsertResponse, error) {
43  log.Printf("Insert - Received: %v, %v", in.Path, in.Data)
44  mopeInsCtr.Inc()
45  s.mMu.Lock()
46  defer s.mMu.Unlock()
```

```
47
48   var err error
49   if s.mopeRoot, err = mope.InsertPath(s.mopeRoot, in.Path, in.Data); err !=
         nil {
50    return &pb.InsertResponse{Message: err.Error(), Success: false}, nil
51   }
52   return &pb.InsertResponse{Message: "Success", Success: true}, nil
53  }
54
55  // Traverse implements protobuf.MopeService, this is the insertion logic
56  func (s *server) Traverse(ctx context.Context, in *pb.TraverseRequest) (*pb.
        TraverseResponse, error) {
57   log.Printf("Traverse - Received: %v", in.Path)
58   mopeTravCtr.Inc()
59   s.mMu.RLock()
60   defer s.mMu.RUnlock()
61   node, err := mope.TraversePath(s.mopeRoot, in.Path)
62   if err != nil {
63    return &pb.TraverseResponse{Message: err.Error(), Success: false}, nil
64   }
65   if node == nil {
66    return &pb.TraverseResponse{Message: "Success", Success: true}, nil
67   }
68   return &pb.TraverseResponse{Message: "Success", Success: true, Data: node.
         Value}, nil
69  }
70
71  var (
72   // for tls authentication and confidentiality
73   serverCert string
74   serverKey  string
75
76   clientCert string
77   clientKey  string
78
79   caCert string
80
81   addr       string
82   sqlConnStr string
83  )
84
85  func init() {
86   flag.StringVar(&serverCert, "sCrt", "", "Server TLS Cert")
87   flag.StringVar(&serverKey, "sKey", "", "Server TLS Key")
88   flag.StringVar(&clientCert, "cCrt", "", "Client TLS Cert")
89   flag.StringVar(&clientKey, "cKey", "", "Client TLS Key")
90   flag.StringVar(&caCert, "caCrt", "", "CA TLS Cert")
91
92   flag.StringVar(&addr, "addr", ":1234", "Server Address")
93   flag.StringVar(&addr, "sql", "", "SQL connection string")
94
95   flag.Parse()
96  }
97
98  // main - main entrypoint for the dealer server
99  func main() {
100  go func() {
101   http.Handle("/metrics", promhttp.Handler())
102   log.Fatal(http.ListenAndServe(":9090", nil))
```

```
103   }()
104
105   var tree *mope.Node
106
107   go mope.PrintInorder(tree, "", mope.Depth(tree))
108   // load in local CA
109   certificate, err := tls.LoadX509KeyPair(serverCert, serverKey)
110   if err != nil {
111    log.Fatalf("failed to load certs: %s\n", err.Error())
112   }
113
114   certPool := x509.NewCertPool()
115   ca, err := ioutil.ReadFile(caCert)
116   if err != nil {
117    log.Fatalf("failed to load certs: %s\n", err.Error())
118   }
119
120   if ok := certPool.AppendCertsFromPEM(ca); !ok {
121    log.Fatalf("failed to load certs: %s\n", err.Error())
122   }
123
124   lis, err := net.Listen("tcp", addr)
125   if err != nil {
126    log.Fatalf("failed to listen: %v", err)
127   }
128
129   // require client credentials
130   creds := credentials.NewTLS(&tls.Config{
131    ClientAuth:   tls.RequireAndVerifyClientCert,
132    Certificates: []tls.Certificate{certificate},
133    ClientCAs:    certPool,
134   })
135
136   s := grpc.NewServer(grpc.Creds(creds))
137
138   pb.RegisterMopeServiceServer(s, &server{mMu: sync.RWMutex{}, mopeRoot: tree})
139
140   if err := s.Serve(lis); err != nil {
141    log.Fatalf("failed to serve: %v", err)
142   }
143  }
```

Appendix E

mOPE Data Structure Source Code

src/mope/tree.go

```
1  package mope
2
3  import (
4   "errors"
5   "fmt"
6
7   "github.com/prometheus/client_golang/prometheus"
8   "github.com/prometheus/client_golang/prometheus/promauto"
9  )
10
11 // Node - mope.Node is a node in the mOPE tree
12 type Node struct {
13  Left  *Node
14  Right *Node
15  Value []byte
16 }
17
18 // NewNode - create a new mope.Node
19 func NewNode(value []byte) *Node {
20  return &Node{Value: value}
21 }
22
23 var (
24  mopeTreePath = promauto.NewCounter(prometheus.CounterOpts{
25   Name: "mope_tree_traverse_path_total",
26   Help: "The total number of tree traversals",
27  })
28  mopeTreeInsert = promauto.NewCounter(prometheus.CounterOpts{
29   Name: "mope_tree_insert_path_total",
30   Help: "The total number of tree inserts",
31  })
32 )
33
34 // TraversePath - traverse the tree based on the path
35 func TraversePath(node *Node, path string) (*Node, error) {
36  mopeTreePath.Inc()
37  if node == nil || len(path) == 0 {
38   if len(path) > 0 {
39    return nil, errors.New("failed to traverse to path")
40   }
41   return node, nil
42  } else if path[0] == '0' {
43   return TraversePath(node.Left, path[1:])
44  } else {
45   return TraversePath(node.Right, path[1:])
46  }
47 }
```

```
48
49 // InsertPath - insert a node at a particular path in the tree
50 func InsertPath(node *Node, path string, data []byte) (*Node, error) {
51  mopeTreeInsert.Inc()
52  if node == nil {
53   if len(path) > 0 {
54    return nil, errors.New("failed to traverse to path")
55   }
56   return NewNode(data), nil
57  }
58  if len(path) == 0 {
59   // found the node where we need to insert
60   n := NewNode(data)
61   // new node belongs to left
62
63   n.Left = node
64
65   return n, nil
66  }
67
68  var err error
69  if path[0] == '0' {
70   node.Left, err = InsertPath(node.Left, path[1:], data)
71   return node, err
72  }
73  node.Right, err = InsertPath(node.Right, path[1:], data)
74  return node, err
75 }
76
77 func Depth(node *Node) int {
78  if node == nil {
79   return 0
80  }
81  return 1 + Depth(node.Left) + Depth(node.Right)
82 }
83
84 // PrintInorder - traverse the BST depth first
85 func PrintInorder(node *Node, tag string, depth int) {
86  if node == nil {
87   return
88  }
89  PrintInorder(node.Left, tag+"0", depth-1)
90
91  fmt.Printf("%s", tag)
92  fmt.Printf("%d", 1)
93  for i := 0; i < depth; i++ {
94   fmt.Printf("%d", 0)
95  }
96  fmt.Printf(" - %v\n", node.Value)
97
98  PrintInorder(node.Right, tag+"1", depth-1)
99 }
```

Appendix F

itOPE 2 of 3 Compose File

src/compose-2-out-of-3.yaml

```
1  version: '3'
2  services:
3      prometheus:
4          image: prom/prometheus
5          volumes:
6              - "./prom.yaml:/etc/prometheus/prometheus.yml"
7          expose:
8              - "9088"
9          ports:
10             - "9088:9090"
11     dealer:
12         image: dealer:latest
13         container_name: dealer.local
14         expose:
15             - "8080"
16             - "9090"
17         ports:
18             - "8080:8080"
19             - "9090:9090"
20         entrypoint:
21             - "/dealer"
22             - "-n"
23             - "3"
24             - "-t"
25             - "2"
26             - "-addr"
27             - "dealer.local:8080"
28             - "-sCrt"
29             - "/certs/dealer/dealer.crt"
30             - "-sKey"
31             - "/certs/dealer/dealer.key"
32             - "-caCrt"
33             - "/certs/ca/rootCA.pem"
34     mope:
35         image: mope:latest
36         container_name: mope.local
37         expose:
38             - "8079"
39             - "9089"
40         ports:
41             - "8079:8079"
42             - "9089:9090"
43         entrypoint:
44             - "/mope"
45             - "-addr"
46             - "mope.local:8079"
47             - "-sCrt"
```

```
48              - "/certs/mope/mope.crt"
49              - "-sKey"
50              - "/certs/mope/mope.key"
51              - "-caCrt"
52              - "/certs/ca/rootCA.pem"
53      participant1:
54          image: participant:latest
55          container_name: participant1.local
56          depends_on:
57              - mope
58              - dealer
59          expose:
60              - "8081"
61              - "9091"
62          ports:
63              - "8081:8081"
64              - "9091:9090"
65          entrypoint:
66              - "/participant"
67              - "-addr"
68              - "participant1.local:8081"
69              - "-sCrt"
70              - "/certs/participants/participant1/participant1.crt"
71              - "-sKey"
72              - "/certs/participants/participant1/participant1.key"
73              - "-caCrt"
74              - "/certs/ca/rootCA.pem"
75              - "-dealer"
76              - "dealer.local:8080"
77              - "-mope"
78              - "mope.local:8079"
79      participant2:
80          image: participant:latest
81          container_name: participant2.local
82          depends_on:
83              - mope
84              - dealer
85          expose:
86              - "8082"
87              - "9092"
88          ports:
89              - "8082:8082"
90              - "9092:9090"
91          entrypoint:
92              - "/participant"
93              - "-addr"
94              - "participant2.local:8082"
95              - "-sCrt"
96              - "/certs/participants/participant2/participant2.crt"
97              - "-sKey"
98              - "/certs/participants/participant2/participant2.key"
99              - "-caCrt"
100             - "/certs/ca/rootCA.pem"
101             - "-dealer"
102             - "dealer.local:8080"
103             - "-mope"
104             - "mope.local:8079"
105     participant3:
106         image: participant:latest
```

```
107         container_name: participant3.local
108         depends_on:
109             - mope
110             - dealer
111         expose:
112             - "8083"
113             - "9093"
114         ports:
115             - "8083:8083"
116             - "9093:9090"
117         entrypoint:
118             - "/participant"
119             - "-addr"
120             - "participant3.local:8083"
121             - "-sCrt"
122             - "/certs/participants/participant3/participant3.crt"
123             - "-sKey"
124             - "/certs/participants/participant3/participant3.key"
125             - "-caCrt"
126             - "/certs/ca/rootCA.pem"
127             - "-dealer"
128             - "dealer.local:8080"
129             - "-mope"
130             - "mope.local:8079"
```

Appendix G

itOPE 3 of 5 Compose File

src/compose-3-out-of-5.yaml

```
1  version: '3'
2  services:
3      prometheus:
4          image: prom/prometheus
5          volumes:
6              - "./prom.yaml:/etc/prometheus/prometheus.yml"
7          expose:
8              - "9088"
9          ports:
10             - "9088:9090"
11     dealer:
12         image: dealer:latest
13         container_name: dealer.local
14         expose:
15             - "8080"
16             - "9090"
17         ports:
18             - "8080:8080"
19             - "9090:9090"
20         entrypoint:
21             - "/dealer"
22             - "-n"
23             - "5"
24             - "-t"
25             - "3"
26             - "-addr"
27             - "dealer.local:8080"
28             - "-sCrt"
29             - "/certs/dealer/dealer.crt"
30             - "-sKey"
31             - "/certs/dealer/dealer.key"
32             - "-caCrt"
33             - "/certs/ca/rootCA.pem"
34     mope:
35         image: mope:latest
36         container_name: mope.local
37         expose:
38             - "8079"
39             - "9089"
40         ports:
41             - "8079:8079"
42             - "9089:9090"
43         entrypoint:
44             - "/mope"
45             - "-addr"
46             - "mope.local:8079"
47             - "-sCrt"
```

```
48              - "/certs/mope/mope.crt"
49              - "-sKey"
50              - "/certs/mope/mope.key"
51              - "-caCrt"
52              - "/certs/ca/rootCA.pem"
53      participant1:
54          image: participant:latest
55          container_name: participant1.local
56          depends_on:
57              - mope
58              - dealer
59          expose:
60              - "8081"
61              - "9091"
62          ports:
63              - "8081:8081"
64              - "9091:9090"
65          entrypoint:
66              - "/participant"
67              - "-addr"
68              - "participant1.local:8081"
69              - "-sCrt"
70              - "/certs/participants/participant1/participant1.crt"
71              - "-sKey"
72              - "/certs/participants/participant1/participant1.key"
73              - "-caCrt"
74              - "/certs/ca/rootCA.pem"
75              - "-dealer"
76              - "dealer.local:8080"
77              - "-mope"
78              - "mope.local:8079"
79      participant2:
80          image: participant:latest
81          container_name: participant2.local
82          depends_on:
83              - mope
84              - dealer
85          expose:
86              - "8082"
87              - "9092"
88          ports:
89              - "8082:8082"
90              - "9092:9090"
91          entrypoint:
92              - "/participant"
93              - "-addr"
94              - "participant2.local:8082"
95              - "-sCrt"
96              - "/certs/participants/participant2/participant2.crt"
97              - "-sKey"
98              - "/certs/participants/participant2/participant2.key"
99              - "-caCrt"
100             - "/certs/ca/rootCA.pem"
101             - "-dealer"
102             - "dealer.local:8080"
103             - "-mope"
104             - "mope.local:8079"
105     participant3:
106         image: participant:latest
```

```
107         container_name: participant3.local
108         depends_on:
109             - mope
110             - dealer
111         expose:
112             - "8083"
113             - "9093"
114         ports:
115             - "8083:8083"
116             - "9093:9090"
117         entrypoint:
118             - "/participant"
119             - "-addr"
120             - "participant3.local:8083"
121             - "-sCrt"
122             - "/certs/participants/participant3/participant3.crt"
123             - "-sKey"
124             - "/certs/participants/participant3/participant3.key"
125             - "-caCrt"
126             - "/certs/ca/rootCA.pem"
127             - "-dealer"
128             - "dealer.local:8080"
129             - "-mope"
130             - "mope.local:8079"
131     participant4:
132         image: participant:latest
133         container_name: participant4.local
134         depends_on:
135             - mope
136             - dealer
137         expose:
138             - "8084"
139             - "9094"
140         ports:
141             - "8084:8084"
142             - "9094:9090"
143         entrypoint:
144             - "/participant"
145             - "-addr"
146             - "participant4.local:8084"
147             - "-sCrt"
148             - "/certs/participants/participant4/participant4.crt"
149             - "-sKey"
150             - "/certs/participants/participant4/participant4.key"
151             - "-caCrt"
152             - "/certs/ca/rootCA.pem"
153             - "-dealer"
154             - "dealer.local:8080"
155             - "-mope"
156             - "mope.local:8079"
157     participant5:
158         image: participant:latest
159         container_name: participant5.local
160         depends_on:
161             - mope
162             - dealer
163         expose:
164             - "8085"
165             - "9095"
```

```
166         ports:
167             - "8085:8085"
168             - "9095:9090"
169         entrypoint:
170             - "/participant"
171             - "-addr"
172             - "participant5.local:8085"
173             - "-sCrt"
174             - "/certs/participants/participant5/participant5.crt"
175             - "-sKey"
176             - "/certs/participants/participant5/participant5.key"
177             - "-caCrt"
178             - "/certs/ca/rootCA.pem"
179             - "-dealer"
180             - "dealer.local:8080"
181             - "-mope"
182             - "mope.local:8079"
```

Bibliography

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004.

[2] S. Agrawal, P. Mohassel, P. Mukherjee, and P. Rindal. Dise: Distributed symmetric-key encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1993–2010. ACM, 2018.

[3] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with o (1) accesses. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 785–794. SIAM, 2009.

[4] M. Bellare, M. Fischlin, A. O'Neill, and T. Ristenpart. Deterministic encryption: Definitional equivalences and constructions without random oracles. In *Annual International Cryptology Conference*, pages 360–378. Springer, 2008.

[5] G. R. Blakley et al. Safeguarding cryptographic keys. In *Proceedings of the national computer conference*, volume 48, 1979.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 224–241. Springer, 2009.

[7] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Annual Cryptology Conference*, pages 578–595. Springer, 2011.

[8] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *International conference on the theory and applications of cryptographic techniques*, pages 506–522. Springer, 2004.

[9] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without

obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015.

[10] E. Brickell, G. Di Crescenzo, and Y. Frankel. Sharing block ciphers. In *Australasian Conference on Information Security and Privacy*, pages 457–470. Springer, 2000.

[11] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical order-revealing encryption with limited leakage. In *International Conference on Fast Software Encryption*, pages 474–493. Springer, 2016.

[12] H. F. Gaines. *Elementary cryptanalysis: a study of ciphers and their solution*. American photographic publishing Company, 1943.

[13] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[14] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *Journal of Statistical Computation and Simulation*, 22(2):127–145, 1985.

[15] F. Kerschbaum and A. Schroepfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 275–286. ACM, 2014.

[16] K. Lewi and D. J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178. ACM, 2016.

[17] K. M. Martin, J. Pieprzyk, R. Safavi-Naini, H. Wang, and P. R. Wild. Threshold macs. In *International Conference on Information Security and Cryptology*, pages 237–252. Springer, 2002.

[18] K. M. Martin, R. Safavi-Naini, H. Wang, and P. R. Wild. Distributing the encryption and decryption of a block cipher. *Designs, Codes and Cryptography*, 36(3):263–287, 2005.

[19] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and kdcs.

In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 327–346. Springer, 1999.

[20] G. Ozsoyoglu, D. A. Singer, and S. S. Chung. Anti-tamper databases. In *Data and Applications Security XVII*, pages 133–146. Springer, 2004.

[21] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[22] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy*, pages 463–477. IEEE, 2013.

[23] D. S. Roche, D. Apon, S. G. Choi, and A. Yerukhimovich. Pope: Partial order preserving encoding. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2016.

[24] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[25] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55. IEEE, 2000.

[26] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *USENIX Security Symposium*, pages 79–91, 1999.

¡++¿