The ISC Int'l Journal of Information Security
ISeCure

January 2019, Volume 11, Number 1 (pp. 15–34)
http://www.isecure-journal.org

SESOS: A Verifiable Searchable Outsourcing Scheme for Ordered Structured Data in Cloud Computing

Javad Ghareh Chamani [1,2,*], Mohammad Sadeq Dousti [1], Rasool Jalili [1], and Dimitrios Papadopoulos [2]

[1] *Data & Network Security Lab, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran*
[2] *Hong Kong University of Science and Technology, Hong Kong, China*

ARTICLE INFO.

Article history:
Received: 13 September 2018
Revised: 18 December 2018
Accepted: 25 January 2019
Published Online: 30 January 2019

Keywords:
Cloud Computing, Data Outsourcing, Order-Preserving Encryption, Searchable Encryption.

ABSTRACT

While cloud computing is growing at a remarkable speed, privacy issues are far from being solved. One way to diminish privacy concerns is to store data on the cloud in encrypted form. However, encryption often hinders useful computation cloud services. A theoretical approach is to employ the so-called fully homomorphic encryption, yet the overhead is so high that it is not considered a viable solution for practical purposes. The next best thing is to craft special-purpose cryptosystems which support the set of operations required to be addressed by cloud services. In this paper, we put forward one such cryptosystem, which supports efficient search over structured data types, such as timestamps, which are comprised of several segments with well-known values. The new cryptosystem, called SESOS, provides the ability to execute LIKE queries, along with the search for exact matches, as well as comparison. In addition, the extended version, called XSESOS, allows for verifying the integrity of ciphertexts. The overhead of executing equality and comparison operations is negligible. The performance of LIKE queries is significantly improved by up to 1370X and the performance of result decryption improved by 520X compared to existing solutions on a database with merely 100K records.

1 Introduction

The rapid and constant generation of organizational data complicates storing and management of data for their owners. While increasing storage capacity is costly even for small companies, it imposes greater costs and problems for medium and larger enterprises. Cloud computing provides storage and computational resources required to handle outsourced data. In comparison to proprietary infrastructure, clouds are significantly cheaper for the same degree of availability and capacity of storage and processing. However, security and privacy concerns are the main obstacles of data outsourcing. Although the data owner and the service provider (SP) often agree on some kind of privacy statement, the issue still remains. If data owners store their data as plaintext on cloud servers, it is possible for the SP to view, analyze, and even sell the data to third parties. Therefore, the only reliable solution is to encrypt the outsourced data. However, fetching and decrypting all data per each query causes significant communication and performance penalties,

[*] Corresponding author.

Email addresses: gharehchamani@ce.sharif.edu (J. Ghareh Chamani), dousti@ce.sharif.edu (M.S. Dousti), jalili@sharif.ir (R. Jalili), dipapado@cse.ust.hk (D. Papadopoulos)

ISSN: 2008-2045 © 2019 ISC. All rights reserved.

thus rendering ordinary encryption impractical.

To cope with the issue, many data outsourcing architectures such as CryptDB [1], Monomi [2] and SDB [3] were proposed in last seven years, whose aim is to provide search capability on encrypted data. To this end, they used different attribute-based encryption schemes in a unified structure. For instance, CryptDB used multiple columns to store ciphertexts corresponding to different encryption schemes for each plaintext value. To execute a search query, only a subset of those columns were used, and fetching all ciphertexts was not necessary anymore.

Although using such architectures helps in the execution of some types of queries, they neither support all data types, nor adequately cover all operations in queries. For example, they support timestamps by converting it to other data types. Even though this approach solves some types of queries requirement such as comparison and equality, it cannot support other types of queries such as `LIKE` and addition.

There are two possible approaches to solve these problems: (1) a high-throughput fully-homomorphic encryption (FHE) scheme [4] which supports all desired mathematical operations on encrypted data, and (2) a type-dependent encryption scheme. Although some theoretical efforts were made in the first direction [5, 6], the proposed FHEs are still far from being practical. Therefore, many researchers are actively crafting encryption schemes for specific data types to address practical requirements.

A common data type which is frequently used in databases (specially, data warehouses and analytic systems) is the *structured* data types, which are composed of several segments with well-known values. For instance, a timestamp is comprised of the following segments: year, month, day, hour, minute, second, and perhaps millisecond or nanosecond. Another example is a network address which, in case of IPv4, has four segments (e.g., `192.168.1.1`). As explained above, existing solutions support such data types via type conversion, such as storing a timestamp as a `long int` value. While this approach allows for some types of queries (such as fetching all fields with a certain value), it falls short of efficiently executing queries regarding segment values. For instance, the existing solutions do not support queries such as “fetch all timestamps whose month is February.”

1.1 Obvious Ideas Which Fail to Work

There are several seemingly “obvious” solutions for encrypting structured data types, though careful investigation reveals inherent issues with such basic approaches. We will discuss some of these solutions, and explain their shortcomings, before proposing our solution.

The easiest encryption scheme for structured data types is the one already used in the literature: The structured data type is converted to some simple data type, and then an order-preserving encryption (OPE) is applied. For instance, a timestamp T is often converted to a (long) integer by counting the number of milli-/nano-seconds from some point in time (e.g., `1970 Jan 1 00:00:00`, known as the “Unix epoch”) until T. This conversion gives a number which preserves the order of dates because of its incremental nature. Equality and comparison queries can be executed on converted data after encryption with an OPE algorithm. However, as explained earlier, the structured data have several segments, and the users may desire to retrieve specific data according to their constituent segments. For example, a user may execute a query to retrieve all record with specific month value. In this situations, the converted value does not support the user’s query. Consequently, to execute such queries, it is required to decrypt all records and filter out unmatched ones in a trusted location (i.e., at the client side).

Another solution is to separately encrypt each segment using an OPE scheme, and then store each ciphertext in a separate column. As a result, each value is scattered in multiple columns. While this method supports most operations on structured data, a service provider (SP) can easily cluster data based on their individual segments, as OPE supports equality testing. For instance, the SP can partition timestamps into 12 clusters based on the OPE-encrypted value of the “month” column.

Another issue with this approach is that the SP can recover plaintexts by applying a simple frequency analysis on each column, as the domain (i.e., the set of possible values) for each segment is very limited. For instance, the month segment in a timestamp contains at most 12 different values. If SP knows *a priori* that September is the most frequent month in the database, he can guess that the most frequent ciphertext in the month column corresponds to September.

Another solution which may seem applicable is using a *sliding window* on structured data segments. In this method, data is randomly padded prior to order-preserving encryption (OPE). For Instance, an IP value such as `192.168.1.5` is converted to `192.168.1.5`.$R_1.R_2.R_3.R_4$, and the window slides from left to right. For instance:

$$\mathsf{Enc}(\mathsf{192.168.1.5}.R_1) \;\mid\; \mathsf{Enc}(\mathsf{168.1.5}.R_1.R_2) \;\mid\; \mathsf{Enc}(\mathsf{1.5}.R_1.R_2.R_3) \;\mid\; \mathsf{Enc}(\mathsf{5}.R_1.R_2.R_3.R_4)\,,$$

where Enc is any OPE scheme, and “|” is the concatenation operator. In this method, the comparison between two segments of plaintext can be easily tested. As such, LIKE queries is practical on the encrypted data. The random padding is applied to diversify the encryption and foil frequency analysis. Yet this kind of padding is futile: The service provider (SP) can simply cluster ciphertexts based on the value of their segments. For example, the SP can sort IP ciphertexts based on the second octet, and infer equivalent plaintexts based on a priori knowledge regarding their distribution.

To eliminate such issue, one may suggest to increase the randomness in the sliding window—e.g., use only one plaintext segment in each window and fill the rest with random padding. However, the problem still remains: Using OPE properties, the SP can mount the same attack described above.

1.2 Our Contribution

Evidently, providing a secure and practical solution for structured data types is not trivial. In this paper, we propose a new encryption scheme called SESOS (searchable outsourcing scheme for ordered structured data), which does not suffer from the mentioned issues. It supports the execution of different operations such as LIKE, equality, and comparison on encrypted structured data. Practical evaluation demonstrates that SESOS is much faster than previous schemes: It does not add any overhead to operations such as equality, comparison, and encryption. On the contrary, SESOS speeds up the decryption time by yp to 520X and improves the execution time of queries with LIKE condition by up to 1370X (depending on the percentage of records fetched).

SESOS can incorporate any order-preserving encryption (OPE) scheme, as well as a novel encryption scheme called Multi-map Perfectly Secure Cryptosystem (MuPS). The latter is defined in this paper, and proven to be secure in an information-theoretic framework: MuPS is perfectly secure, and hence a MuPS ciphertext leaks no information about the plaintext, even to an infinitely powerful adversary. As a result, SESOS is as secure as the underlying OPE scheme.

In short, SESOS is an encryption method with the following properties:

- **Preserving the order of plaintexts:** This property is achieved via the underlying OPE scheme. Therefore, an untrusted cloud service provider (SP) can execute queries which compare encrypted columns.
- **Ability to execute LIKE queries:** One of the most important features of SESOS is its ability to execute query with LIKE condition on encrypted data: SESOS supports queries where the client wants to fetch structured data whose segments satisfy a specific condition. For instance, a query such as SELECT * FROM tbl WHERE date LIKE ‘‘Feb’’ is supported on the encrypted date column in the table tbl.
- **Very fast execution of LIKE queries and decryption of the results:** SESOS significantly increases the performance of LIKE queries. The performance gain depends on the fraction of records fetched, but our analysis shows that a speed-up of at most 1370X can be expected. The performance of LIKE queries is important mostly in data warehouses and analytic systems which have a huge number of timestamp data. Furthermore, because of using simple operation such as lookup, it improves the decryption performance up to 520X.
- **Execution of equality and range queries with negligible overhead:** SESOS provides the ability to execute equality and range queries over encrypted data. This feature has negligible overhead compared to existing methods.
- **Verifiability of ciphertext:** SESOS makes it possible to verify the integrity of the returned ciphertexts. This is due to using two underlying encryption schemes: OPE and MuPS. The client can verify the integrity of ciphertexts by decrypting each scheme separately, and comparing the results.
- **Provable security:** As described earlier, many obvious solutions which solve one problem by creating a bigger one, such as compromising the security, cab be proposed. In this paper, we prove the prefect secrecy of MuPS. As a result, SESOS is shown to be as secure as the underlying OPE scheme, which can be selected based on the security and efficiency requirements.

1.3 Organization

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 describes some preliminaries and definitions. Section 4 presents SESOS and gives some examples. Section 5 evaluates the performance of SESOS. Finally, Section 7 concludes the paper.

2 Related Work

Research in data outsourcing has two directions: Architectures and encryption schemes. Below, we briefly review previous work in each category.

2.1 Architectures

CryptDB [1, 7] is one of the most popular architectures proposed in recent years. It can be considered as the first practical architecture that used encryption for outsourcing data in the cloud. CryptDB influenced subsequent database outsourcing architectures. It uses several encryption schemes in a unified "onion" structure to support different operations on encrypted data. Although CryptDB seems to be practical, it suffers from some important issues such as the lack of supporting complex queries or lazy encryption methods.

Monomi [2] was introduced by some authors of CryptDB in 2013. They improved CryptDB through a planner and an optimizer to support complex and nested queries. Although Monomi fixed some issues of CryptDB, some problems such as supporting timestamps still exist.

In 2015, SDB was proposed as an alternative to CryptDB's trend [3]. Its architecture is based on mathematical properties of operations, and supports some basic operations as well as a planner for executing nested queries. Similar to CryptDB and Monomi, some data types and queries are still unsupported in SDB.

Although some other frameworks and architectures such as [8, 9] exist in the literature, they did not have significant impact on the trend of this field. We ignore their details in this survey.

2.2 Encryption Schemes

In recent years, several encryption schemes were introduced for data outsourcing. Several of these schemes [10, 11] use indexing information along with ciphertexts. Indexing can be done in different ways such as encryption, segmentation, hashing, or order-preserving encryptions [12–14].

Another category of encryption schemes uses secret sharing [15, 16] to split ciphertexts among several service providers (SPs). This category requires several non-collaborative SPs to preclude the recovery of plaintexts. This requirement is only applicable for large enterprises; small businesses are unable to pay more than one SP.

In 2005, Aggarwal suggested data partitioning to provide privacy in data outsourcing [17]. The basic idea is that most of the existing data are not private in themselves. However, the privacy is required when the data is stored in conjunction with other data. As an example, "salary" is just a numeric value, but it may be considered private only when it is stored along with the name of an employee. Thus, by storing each part of data separately in different places, privacy can be achieved. This method has the same problem as secret sharing, as it requires more than one SP.

There has been lots of work on order-preserving encryption (OPE) schemes both in the research community [15, 18, 19] and in industry [20, 21]. Some tried to propose new methods while the others improved the existing ones. In 2013, Raluca proposed mOPE as an ideal secure OPE [22]. It was based on mutable ciphertexts needed for ideal security. This scheme was used in CryptDB to provide comparison capability for users. Two years later, Hwang *et al.* [23] suggested Fast OPE (FOPE) which was significantly faster than mOPE. Kerschbaum [24] conducted a research to prevent service providers (SPs) from frequency analysis on OPEs. The idea was to introduce some randomness in the encryption mechanism to change the deterministic property of OPE encryption. Although this method increased the security, it reduced the performance in comparison to the previous schemes.

2.3 Searchable Encryption

Searching on encrypted texts was first examined by Song et al [25] in 2000. They proposed a non-interactive method for searching on encrypted texts. Non-interactive search means that the client gives his queries to the system at the setup time and will not interact with the system anymore. Although this definition is not practical in a real system, it was the base for future research.
In 2006, Curtmola et al proposed an improved definition and efficient construction for searching on encrypted texts [26]. They proposed an adaptive SSE scheme in addition to a non-adaptive one which was more practical in real applications. However, it still had some issues which prevented users from using it. The main issue was dynamism which means the ability of users to add and remove keywords to the currently existing database or files. In 2012, Kamara et al proposed the first dynamic SSE which supported the insertion and deletion of keywords [27]. From that time, lots of work has been done to improve the performance of existing dynamic SSE schemes [28–34].

3 Preliminaries

Let S be a finite set and n be a natural number. Then $|S|$ denotes the number of elements in S, and S^n denotes the n-time Cartesian product of S by itself. Specifically, the n-time Cartesian product of $\{0,1\}$, denoted as $\{0,1\}^n$, has a special meaning: The set of all binary strings of length n. For any binary string $s \in \{0,1\}^n$, the length of s is denoted by $|s|$, and is equal to n. If $s_1, \ldots, s_t$ are t string values, $s_1 \mid \cdots \mid s_t$ shows their concatenation.

We use $\vec{v}$ to suggest that v is a vector. An *exclama-*

tion mark is used to signify factorial: $n! \overset{\text{def}}{=} 1\times2\times\cdots\times n$. The symbol $\perp$ is used to show the output of an algorithm upon failed. We use λ to denote the security parameter. It is customary to provide algorithms with λ in *unary notation* 1^λ. This is because, by convention, an algorithm's running time is measured in the *length* of its input, and $|1^\lambda| = \lambda$.

Let $Gen_{prf}(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $G : \{0,1\}^\lambda \times \{0,1\}^\ell \times \{0,1\}^{\ell'} \to \{0,1\}^{\ell''}$ be a pseudorandom function (PRF) family. $G_K(P, x)$ denotes $G(K, P, x)$. G is a secure PRF family if for all PPT adversaries Adv, $|\Pr[K \leftarrow Gen_{prf}(1^\lambda); \mathsf{Adv}^{G_K(\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{R(\cdot)}(1^\lambda) = 1]| \leq v(\lambda)$, where $R : \{0,1\}^\ell \to \{0,1\}^{\ell'}$ is a truly random function.

Definition 1 (Order-Preserving Encryption). A quintuple $\Pi_{\mathsf{ope}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathcal{P}, \mathcal{C})$ is called an *order-preserving encryption* (OPE) if it satisfies the following properties:

- **Key Generation:** On input the security parameter 1^λ, the efficient key generation algorithm Gen outputs the key k.
- **Encryption:** On input the key k output by Gen and a plaintext $p \in \mathcal{P}$, the efficient encryption algorithm Enc outputs a ciphertext $c \in \mathcal{C}$.
- **Decryption:** On input the key k output by Gen and a ciphertext $c \in \mathcal{C}$, the efficient decryption algorithm Dec outputs a plaintext $p \in \mathcal{P}$.
- **Order Preservation:** Let $\leq$ be a total order on $\mathcal{P}$, and $\preceq$ be a total order on $\mathcal{C}$. For any key k output by Gen, and all plaintexts $p_1, p_2 \in \mathcal{P}$, it is required that:

$$p_1 \leq p_2 \quad \Leftrightarrow \quad \mathsf{Enc}(k, p_1) \preceq \mathsf{Enc}(k, p_2)\,.$$

Correctness: For any key k output by Gen, and all plaintexts $p \in \mathcal{P}$, it is required that:

$$\mathsf{Dec}(k, \mathsf{Enc}(k, p)) = p\,.$$

Security: There are several notions of security for OPE. Examples include IND-O-CPA and POPF-CCA [35], $(r, q+1)$-WOW and $(r, q+1)$-WDOW [18], and $(\mathcal{X}, \theta, q)$-indistinguishability [36]. We do not investigate these definitions here. Our proposed scheme can use any OPE, and inherits its security under the same definition.

Remark 1. In this paper, we will assume that $\mathcal{C} = \{0,1\}^\kappa$ for some positive integer κ. This is just for notational simplicity, as the algorithms can easily split concatenated strings based on their lengths (looking ahead, Algorithm 3 uses this property).

For a suitable value of κ, any OPE ciphertext can be padded by enough zeros (to the left) so that the length of its binary representation is κ. Therefore, there is no loss of generality in assuming that $\mathcal{C} = \{0,1\}^\kappa$.

4 The Proposed Encryption Scheme

The proposed encryption scheme, called SESOS, combines two types of encryption schemes:

(1) Any order-preserving encryption (OPE) scheme as defined in Definition 1;
(2) A novel encryption scheme called MuPS.

Section 4.1 defines MuPS formally, demonstrates the proper way to select its parameters based on the dataset, and proves that MuPS achieves perfect secrecy. In Section 4.2, SESOS is formally defined and we explain how MuPS is combined with an arbitary OPE scheme to provide the final encryption scheme. We also show how SESOS achieves the desired properties such as executing comparison and `LIKE` queries. Finally, Section 4.3 describes a numerical example to clarify SESOS operation.

4.1 MuPS: A Multi-map Perfectly Secure Cryptosystem

Let $\mathcal{P}$ and $\mathcal{C}$ denote the set of plaintexts and ciphertexts, respectively. The *encryption function* Enc, proposed in this section, is a stateful one. A single plaintext might be mapped into one of m ciphertexts, depending on the state (as well as the key). Furthermore, we will prove in Section 4.1.3 that the cryptosystem is perfectly secure. For these reasons, it is called a *Multi-map Perfectly Secure* (MuPS) cryptosystem.

Let us explain, on an intuitive level, two design choices regarding MuPS:

(1) The decryption function Dec of MuPS is stateless. Otherwise, the encryption state must be saved together with the ciphertext in order to make decryption possible. For instance, consider the One-Time Pad (OTP), which is a perfectly secure cryptosystem, whose encryption function is defined as the exclusive-or (XOR) of the key and plaintext: $\mathsf{Enc}(k, p) = k \oplus p$. OTP can be considered as a stateful encryption, as exemplified next.

Let $k = 0001101101100\ldots01$. We first encrypt $p_1 = 11000$. To this end, a prefix of k with length $|p_1| = 5$ is chosen: $k_1 = 00011$, and $c_1 = k_1 \oplus p_1 = 11011$. Next, let us encrypt $p_2 = 101$. The rules of OTP disallows reusing the same key bits. Therefore, the five used bits are ignored. Let k' be k with the first five bits removed. To encrypt p_2, a prefix of k' with length $|p_2| = 3$ is chosen: $k_2 = 011$, and $c_2 = k_2 \oplus p_2 = 110$. The same approach is taken for other plaintexts.

In stateful OTP, the state is the index of the first fresh (unused) key bit. Initially, this index is

0, and it increases by the length of each plaintext.

However, the decryption function of OTP is stateful as well: To decrypt a ciphertext, one must know which part of the key to use.

(2) MuPS never maps a plaintext to the same previously generated ciphertext. Otherwise, the adversary can detect that some plaintext is repeated, which is in sharp contrast to *perfect secrecy* because according to the perfect secrecy definition, the ciphertexts should not reveal **any** information about the plaintext while such repeatation reveals that an already existing plaintext is encrypted again.

One might argue that why OTP can map a plaintext to the same ciphertext while maintaining perfect secrecy. For instance, in OTP, 0 can be encrypted as 0, as 1, and again as 0. This is because the decryption function of OTP is stateful, while the decryption function of MuPS is stateless.

The second property is seemingly a restriction: MuPS can encrypt a plaintext at most m times. Therefore, there is an upper bound n on the total number of "encryptable" plaintexts. However, as shown in Section 4.1.2, m can be picked reasonably to defeat this restriction for all practical purposes. Moreover, we discuss in Section 6 an approach to make the restriction less tight.

MuPS is formally defined in Definition 2. See Section 4.1.1 for a toy example.

Definition 2 (MuPS). *MuPS is a quintuple* $\Pi_{\mathsf{mups}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathcal{P}, \mathcal{C})$, where Gen, Enc, and Dec are efficient algorithms, each of which is described below, while $\mathcal{P}$ and $\mathcal{C}$ are finite sets. Without loss of generality and for notational simplicity, we will assume that $\mathcal{C} = \{0,1\}^{\kappa}$ for some positive integer κ. [1]

It is assumed that one can enumerate the members of $\mathcal{P}$. That is, its members can be indexed as $p_1, p_2, \ldots, p_\ell$ unambiguously. In order for MuPS to be non-degenerate, we assume that $\ell \overset{\text{def}}{=} |\mathcal{P}| > 1$.

MuPS is parameterized by a positive integer m, denoting the number of different mappings per plaintext.

It must hold that $\ell \cdot m \leq 2^{\kappa}$, otherwise, key generation/encryption will be impossible (see below). Furthermore, κ, ℓ, and m must be **polynomially bounded** in the security parameter λ.

- **Key Generation:** On input $(1^{\lambda}, \mathcal{P}, \kappa, m)$, the key generation algorithm Gen outputs $(\mathbf{st}, \mathbf{K}, \mathbf{K^{-1}})$.

  The encryption *state* is denoted by $\mathbf{st}$. It can be regarded as a lookup table with ℓ indices. $\mathbf{st}[p]$ denotes the *state value* corresponding to $p \in \mathcal{P}$. Initially, all state values are zero.

  $\mathbf{K}$ is the key, which is also a lookup table with ℓ indices. $\mathbf{K}[p]$ denotes the *key value* corresponding to $p \in \mathcal{P}$. Contrary to $\mathbf{st}[p]$, which is a numeric value, $\mathbf{K}[p]$ is a vector in $\mathcal{C}^m$.

  To construct $\mathbf{K}$, the key generation algorithm uniformly picks $\ell \cdot m$ elements from $\mathcal{C} = \{0,1\}^{\kappa}$ **without replacement**. The chosen values are denoted as $(c_1, \ldots, c_{\ell \cdot m})$, where c_i is the i^{th} picked element, and all elements are *distinct*. Gen splits $(c_1, \ldots, c_{\ell \cdot m})$ into vectors of length m each, and assigns the j^{th} vector to $\mathbf{K}[p_j]$. That is,

$$\begin{aligned} \mathbf{K}[p_1] &= (c_1, \ldots, c_m) \\ \mathbf{K}[p_2] &= (c_{m+1}, \ldots, c_{2m}) \\ &\cdots \\ \mathbf{K}[p_\ell] &= (c_{(\ell-1)\cdot m+1}, \ldots, c_{\ell \cdot m}) \end{aligned}$$

  Notice that here, we used the fact that the elements of $\mathcal{P}$ can be indexed unambiguously; see Definition 2. Denote by $\mathbf{K}[p][i]$ the i^{th} component of the vector $\mathbf{K}[p]$.

Algorithm 1 Encryption algorithm

```
Enc(st, K, p)
  if $\mathbf{st}[p] \geq m$
    return $(\perp, \mathbf{st})$            //Encryption failure
  $\mathbf{st}[p] \leftarrow \mathbf{st}[p] + 1$            //Update the state
  $c \leftarrow \mathbf{K}[p][\mathbf{st}[p]]$
  return $(c, \mathbf{st})$
```

  The lookup table $\mathbf{K^{-1}}$ is the inverse of $\mathbf{K}$: It takes a ciphertext $c \in \mathcal{C}$, and outputs $p \in \mathcal{P}$ such that c is a component of the vector $\mathbf{K}[p]$. That is, $\mathbf{K^{-1}}[c] = p$ if there exists $p \in \mathcal{P}$ and $i \in \{1, \ldots, m\}$ such that $\mathbf{K}[p][i] = c$. Otherwise, $\mathbf{K^{-1}}[c] = \perp$.

- **Encryption:** On input $\mathbf{st}$, $\mathbf{K}$, and a plaintext $p \in \mathcal{P}$, MuPS encryption function Enc executes Algorithm 1. The algorithm is exemplified in Section 4.1.1.

  Notice that the encryption may fail, but we show in Section 4.1.2 how to pick m such that the chance of failure is arbitrarily small.

- **Decryption:** On input $\mathbf{K^{-1}}$ and $c \in \mathcal{C}$, MuPS decryption function Dec simply outputs $\mathbf{K^{-1}}[c]$.

[1] There are two reasons for this assumption: One is already explained in Remark 1. The second reason is that the key generation algorithm Gen requires to sample uniformly from $\mathcal{C}$, and so $\mathcal{C}$ must be passed to it as an argument. However, if $|\mathcal{C}|$ is exponential in λ, it is infeasible to pass it as an argument to an efficient (i.e., polynomial-time) algorithm, such as Gen. As a result, one must pass a short (i.e., polynomial in λ) "description" of $\mathcal{C}$ to Gen. This is possible when assuming $\mathcal{C} = \{0,1\}^{\kappa}$, as in this case, κ fully specifies $\mathcal{C}$, and we will assume that κ is polynomially bounded in λ.

Notice that Dec is *stateless*, and hence decryption does not depend on the state. Moreover, it is not required to show how it decrypts a vector of ciphertexts.

Algorithm 2 Vector encryption

```
Enc($\mathbf{K}, \vec{p}$)                    // $\vec{p} = (p^{(1)}, p^{(2)}, \ldots, p^{(n)})$
  $\mathbf{st} \leftarrow \mathbf{0}$                        // Initial, all-zero state
  for $i = 1$ to $n$
    $(c^{(i)}, \mathbf{st}) \leftarrow \mathsf{Enc}(\mathbf{st}, \mathbf{K}, p^{(i)})$
    if $c^{(i)} = \perp$
      return $\perp$                   // Encryption failure
  $\vec{c} = (c^{(1)}, c^{(2)}, \ldots, c^{(n)})$
  return $\vec{c}$
```

4.1.0.1 Encrypting multiple plaintexts.

One may consider a sequence of n plaintexts as a *vector* with n plaintext components: $\vec{p} = \left(p^{(1)}, p^{(2)}, \ldots, p^{(n)}\right) \in \mathcal{P}^n$. Encrypting this vector is simply equivalent to encrypting each component from left to right, while updating the state:

$$\mathsf{Enc}(\vec{p}) \overset{\text{def}}{=} \left(\mathsf{Enc}(p^{(1)}), \ldots, \mathsf{Enc}(p^{(n)})\right) .$$

Notice that the state $\mathbf{st}$ and key $\mathbf{K}$ are omitted from the above equation for the sake of readability. A vector is encryptable if no component in its ciphertext is $\perp$. Algorithm 2 formalizes vector encryption.

4.1.1 A Toy Example

Let $\mathcal{P} = \{0, 1\}$, and $\mathcal{C} = \{0, 1\}^3$, $m = 4$, and $n = 6$. Notice that $\ell = 2$. Suppose that Gen outputs the following:

- $\mathbf{st}[0] = 0$ and $\mathbf{st}[1] = 0$.
- $\mathbf{K}[0] = (Q, E, R, Z)$ and $\mathbf{K}[1] = (G, X, D, B)$, where $Q = 001$, $E = 101$, $R = 000$, $Z = 110$, $G = 111$, $X = 011$, $D = 100$, and $B = 010$.
- $\mathbf{K^{-1}}[Q] = \mathbf{K^{-1}}[E] = \mathbf{K^{-1}}[R] = \mathbf{K^{-1}}[Z] = 0$ and $\mathbf{K^{-1}}[G] = \mathbf{K^{-1}}[X] = \mathbf{K^{-1}}[D] = \mathbf{K^{-1}}[B] = 1$.

We next describe how to encrypt a plaintext vector $\vec{p} \in \mathcal{P}^n = \{0, 1\}^6$. Let $\vec{p} = (0, 0, 0, 1, 1, 1)$:

(1) $p^{(1)} = 0$: The encryption algorithm verifies that $0 = \mathbf{st}[p^{(1)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(1)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(1)}] = 1$. Finally, $c^{(1)}$ is set to $\mathbf{K}[p^{(1)}][\mathbf{st}[p^{(1)}]] = \mathbf{K}[0][1] = Q$.

(2) $p^{(2)} = 0$: The encryption algorithm verifies that $1 = \mathbf{st}[p^{(2)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(2)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(2)}] = 2$. Finally, $c^{(2)}$ is set to $\mathbf{K}[p^{(2)}][\mathbf{st}[p^{(2)}]] = \mathbf{K}[0][2] = E$.

(3) $p^{(3)} = 0$: The encryption algorithm verifies that $2 = \mathbf{st}[p^{(3)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(3)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(3)}] = 3$. Finally, $c^{(3)}$ is set to $\mathbf{K}[p^{(3)}][\mathbf{st}[p^{(3)}]] = \mathbf{K}[0][3] = R$.

(4) $p^{(4)} = 1$: The encryption algorithm verifies that $0 = \mathbf{st}[p^{(4)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(4)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(4)}] = 1$. Finally, $c^{(4)}$ is set to $\mathbf{K}[p^{(4)}][\mathbf{st}[p^{(4)}]] = \mathbf{K}[1][1] = G$.

(5) $p^{(5)} = 1$: The encryption algorithm verifies that $1 = \mathbf{st}[p^{(5)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(5)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(5)}] = 2$. Finally, $c^{(5)}$ is set to $\mathbf{K}[p^{(5)}][\mathbf{st}[p^{(5)}]] = \mathbf{K}[1][2] = X$.

(6) $p^{(6)} = 1$: The encryption algorithm verifies that $2 = \mathbf{st}[p^{(6)}] < m = 4$, which is true. Next, $\mathbf{st}[p^{(6)}]$ is incremented by 1, resulting in $\mathbf{st}[p^{(6)}] = 3$. Finally, $c^{(6)}$ is set to $\mathbf{K}[p^{(6)}][\mathbf{st}[p^{(6)}]] = \mathbf{K}[1][3] = D$.

Ultimately, the encryption function outputs $\vec{c} = (c^{(1)}, \ldots, c^{(6)}) = (Q, E, R, G, X, D)$.

For the second example, consider encrypting $\vec{p'} = (0, 1, 0, 0, 1, 0)$. Following the same steps, one can see that $\vec{c'} = (Q, G, E, R, X, Z)$.

For the final example, let us consider an "unencryptable" vector: $\vec{p''} = (0, 0, 0, 0, 0, 1)$. As the number of 0s is more than $m = 4$, the encryption fails for the fifth component $p''^{(5)}$. This is because when encrypting this component, $\mathbf{st}[p''^{(5)}] = 4$, which does not pass the condition $\mathbf{st}[p''^{(5)}] < m$. As a result, Enc outputs $\perp$.

It is fairly easy to demonstrate decryption. For instance, $\vec{c} = (Q, E, R, G, X, D)$ is decrypted as $\vec{p} = \left(\mathbf{K^{-1}}[Q], \mathbf{K^{-1}}[E], \mathbf{K^{-1}}[R], \mathbf{K^{-1}}[G], \mathbf{K^{-1}}[X], \mathbf{K^{-1}}[D]\right) = (0, 0, 0, 1, 1, 1)$, which is correct according to what we computed above.

4.1.2 Setting the Parameter m for MuPS

For fixed values of ℓ and n, the parameter m determines the number of encryptable plaintext vectors. If $m < n/\ell$, no plaintext vector of length n can be encrypted. For instance, if $\mathcal{P} = \{0, 1\}$ and $m = 2$, we cannot encrypt plaintext vectors of length $n = 5$, as the number of mappings per plaintext is not enough.

At the other end of the spectrum, if $m \geq n$, *any* plaintext vector of length n (or less) is encryptable by MuPS. However, we want to keep m as small as possible, since the query complexity on the outsourced DB is proportional to m.

The goal of this section is to show how to pick m just above its lower bound (n/ℓ), such that a *uniformly* chosen vector $\vec{p} \in \mathcal{P}^n$ is encryptable with high probability. This is proven in Theorem 1. However, prior to that proof, we need several notations, as well as some lemmas.

For $i \in \{1, \dots, n\}$, let U_i denote a *uniform* random variable over $\mathcal{P}$. That is, for any $p \in \mathcal{P}$, it holds that $\Pr[U_i = p] = \frac{1}{\ell}$. For $j \in \{1, \dots, \ell\}$, define the indicator function $I_j(U_i)$ as follows:

$$I_j(U_i) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } U_i = p_j \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, define $X_j = \sum_{i=1}^{n} I_j(U_i)$. That is, X_j counts the number of U_is whose outcome is p_j. (Again, we used the assumption that members of $\mathcal{P}$ can be enumerated; see Definition 2.) Since X_js are identically distributed, we may use X to represent any of X_js when the index is of no importance.

Lemma 1. *X is binomially distributed.*

Proof. For $x \in \{0, \dots, n\}$, we have $f_X(x) \overset{\text{def}}{=} \Pr[X = x] = \binom{n}{x}\left(\frac{1}{\ell}\right)^n$. Therefore, the probability mass function (pmf) of X is that of a binomial distribution (for $\ell = 2$). □

Corollary 1. *X has expectation $\mu_X = E[X] = \frac{n}{\ell}$ and variance $\sigma_X^2 = \mathrm{Var}[X] = n\frac{1}{\ell}(1 - \frac{1}{\ell}) = \frac{n(\ell-1)}{\ell^2}$.*

The following useful inequality can be obtained for $\epsilon > 0$ by applying the Chernoff bound (e is the base of the natural logarithm):

$$\Pr\left[X - \mu_X > \epsilon\right] \leq e^{-\epsilon^2/(2\cdot\sigma_X^2)} . \tag{1}$$

Lemma 2. *The random vector $\mathbf{Y} = (X_1, \dots, X_\ell)$ has multinomial distribution.*

Proof. For $x_1, \dots, x_\ell \in \{0, \dots, n\}$ with $\sum_{i=1}^{\ell} x_i = n$, it holds that:

$$\begin{aligned} f_{\mathbf{Y}}(x_1, \dots, x_\ell) &\overset{\text{def}}{=} \Pr\left[(X_1, \dots, X_\ell) = (x_1, \dots, x_\ell)\right] \\ &= \binom{n}{x_1, \dots, x_\ell}\left(\frac{1}{\ell}\right)^n , \end{aligned}$$

where $\binom{n}{x_1,\dots,x_\ell}$ is the *multinomial coefficient* defined as $\frac{n!}{x_1!\cdots x_\ell!}$. Therefore, the pmf of $\mathbf{Y}$ is that of a multinomial distribution. □

Next, we prove a lower bound on the probability that in the random vector $\mathbf{Y}$, no component is greater than m.

Lemma 3. *Define the cumulative distribution functions (CDFs) $F_{X_j}(x) \overset{\text{def}}{=} \Pr[X_j \leq x]$ and $F_{\mathbf{Y}}(x_1, \dots, x_\ell) \overset{\text{def}}{=} \Pr[X_1 \leq x_1, \dots, X_\ell \leq x_\ell]$. Then*

$$F_{\mathbf{Y}}(x_1, \dots, x_\ell) \geq 1 - \sum_{j=1}^{\ell}\left(1 - F_{X_j}(x_j)\right) .$$

Proof. Let us recast the lemma statement in terms events: For $j \in \{1, \dots, \ell\}$, define A_j as the event $X_j \leq x_j$. By De Morgan's laws,

$$1 - \Pr\left[\bigcap_{j=1}^{\ell} A_i\right] = \Pr\left[\left(\bigcap_{j=1}^{\ell} A_i\right)^c\right] = \Pr\left[\bigcup_{j=1}^{\ell} A_i^c\right] .$$

On the other hand, using a *union bound*:

$$\Pr\left[\bigcup_{j=1}^{\ell} A_i^c\right] \leq \sum_{j=1}^{\ell} \Pr[A_i^c] = \sum_{j=1}^{\ell}(1 - \Pr[A_i]) .$$

Therefore,

$$\Pr\left[\bigcap_{j=1}^{\ell} A_i\right] \geq 1 - \sum_{j=1}^{\ell}(1 - \Pr[A_i]) .$$

Consequently ($\bigwedge$ denotes the "logical and" operation):

$$\Pr\left[\bigwedge_{j=1}^{\ell}(X_j \leq x_j)\right] \geq 1 - \sum_{i=1}^{\ell}(1 - \Pr[X_j \leq x_j]) ,$$

as needed. □

Corollary 2. *For any nonnegative integer m:*

$$F_{\mathbf{Y}}(m, m, \dots, m) \geq 1 - \ell \cdot (1 - F_X(m)) . \tag{2}$$

Theorem 1. *Let $(\mathbf{st}, \mathbf{K}, \mathbf{K^{-1}})$ be the output of $\mathsf{Gen}(1^\lambda, \mathcal{P}, \kappa, m)$, and assume n is such that $m > n/\ell$. The probability that a* uniformly *chosen vector $\vec{p} \in \mathcal{P}^n$ is encryptable by Enc is at least*

$$1 - \ell \cdot e^{-\frac{(m\ell-n)^2}{2n(\ell-1)}} . \tag{3}$$

Proof. As proven in Lemma 2, if $\vec{p} \in \mathcal{P}^n$ is picked uniformly, the vector $\mathbf{Y}$ whose components are the count of each plaintext in $\vec{p}$ follows the multinomial distribution.

$\vec{p}$ is encryptable if no component in $\vec{p}$ is repeated more than m times. The probability of this event is $F_{\mathbf{Y}}(m, m, \dots, m)$ for a uniform $\vec{p} \in \mathcal{P}^n$. Inequality (2) gives a lower bound for this probability in terms of the CDF of a binomial random variable. The latter can be approximated using inequality (1), using μ_X and σ_X^2 as defined in Corollary 1, and with $\epsilon = m - n/\ell$:

$$\begin{aligned} 1 - F_X(m) &= \Pr[X > m] \\ &= \Pr[X - n/\ell > \epsilon] \leq e^{-\epsilon^2/(2\cdot\sigma_X^2)} \\ &= e^{-\frac{(m-n/\ell)^2}{2n(\ell-1)/\ell^2}} \\ &= e^{-\frac{(m\ell-n)^2}{2n(\ell-1)}} . \end{aligned}$$

Finally, using inequality (2),

$$\begin{aligned} F_{\mathbf{Y}}(m, m, \dots, m) &\geq 1 - \ell \cdot (1 - F_X(m)) \\ &\geq 1 - \ell \cdot e^{-\frac{(m\ell-n)^2}{2n(\ell-1)}} . \end{aligned}$$

□

Corollary 3. *In order for a* uniformly *chosen vector $\vec{p} \in \mathcal{P}^n$ to be encryptable with probability $\delta < 1$, the value of m must be at a few standard deviations away from the mean (notice that μ_X and σ_X are the mean and standard deviation of the underlying binomial distribution X, rather than those of the multinomial distribution $\mathbf{Y}$):*

$$\begin{aligned} m &\geq \frac{n}{\ell} + \sqrt{\frac{n(\ell-1)}{\ell^2}\cdot 2\ln\left(\frac{\ell}{1-\delta}\right)} \\ &= \mu_X + \sigma_X \cdot \sqrt{2\ln\left(\frac{\ell}{1-\delta}\right)} . \end{aligned}$$

This can be proven by simple algebraic manipulation of inequality (3).

Let us exemplify the result of Corollary 3 with some examples.

Example 1. Let $\ell = 3$ and $n = 100$. We already know that $m \geq n/\ell \approx 33.3$, and with $m = n = 100$, the probability that $\vec{p}$ is encryptable is 1.

Let us now find m such that $\delta = 0.90$. An actual computation (i.e., without any approximation) shows that $m \geq 42$. Using Corollary 3, we get $m \geq 46$.

For $\delta = 0.99$, the actual computation requires $m \geq 46$, while Corollary 3 asks for $m \geq 50$.

In Example 1, we could perform an exact computation, since the values of n and ℓ were small. That said, such a computation took about 10 seconds on a commodity computer. For larger values, exact computation can become very time consuming. Interestingly, numerical computation becomes infeasible for the following example, and we have to resort to the approximation given in Corollary 3.

Example 2. Let $\ell = 256$ and $n = 10^6$. This is a real-world example, where the DB stores a million records, each of which has 256 possible values (such as a byte). We already know that $m \geq n/\ell \approx 3906.3$, and with $m = n = 10^6$, the probability that $\vec{p}$ is encryptable is 1.

Let us now find m such that $\delta = 0.90$. Using Corollary 3, we get $m \geq 4154$.

Moreover, for $\delta = 0.99999$, we get $m \geq 4271$. Notice the huge advantage here: While we need $m = 1,000,000$ to make all vectors encryptable, we merely need $m = 4271$ to make over 99.999% of them encryptable.

4.1.3 MuPS is Perfectly Secure

In this section, we prove that MuPS is perfectly secure. Define $\mathsf{Good}^n_m(\mathcal{P})$ as the set of vectors $\vec{p} \in \mathcal{P}^n$ such that no vector component is repeated more than m times. This set exactly contains all vectors encryptable by MuPS with parameters (m, n).

Example 3. For $\mathcal{P} = \{A, B, C\}$:

$$\mathsf{Good}^2_1(\mathcal{P}) = \Big\{(A,B),(A,C),(B,C),(B,A),(C,A),(C,B)\Big\}.$$

Notice that $(A, A) \notin \mathsf{Good}^2_1(\mathcal{P})$, as A is repeated more than once ($m = 1$).

The MuPS encryption $\vec{p} \in \mathsf{Good}^n_m(\mathcal{P})$ is **not** an arbitrary vector in $\mathcal{C}^n$, as the components in the ciphertext vector cannot be repeated. This is the result of the way MuPS works: No plaintext is ever mapped to the same ciphertext. Therefore, the encryption of $\vec{p} \in \mathsf{Good}^n_m(\mathcal{P})$ is a vector in $\mathsf{Good}^n_1(\mathcal{C})$: That is, the set of vectors in $\mathcal{C}^n$ whose components are repeated at most once.

Let $\mathbf{P}$ be an *arbitrary* random variable over $\mathsf{Good}^n_m(\mathcal{P})$, and define the random variable $\mathbf{C}$ over $\mathsf{Good}^n_1(\mathcal{C})$ as follows:

(1) Let $\vec{p} \leftarrow \mathbf{P}$. That is, we sample $\mathbf{P}$ to get the plaintext vector $\vec{p} \in \mathsf{Good}^n_m(\mathcal{P})$.
(2) Run MuPS key generation algorithm: $(\mathbf{st}, \mathbf{K}, \mathbf{K^{-1}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathcal{P}, \kappa, m)$.
(3) Encrypt $\vec{p}$ by running $\vec{c} \leftarrow \mathsf{Enc}(\mathbf{st}, \mathbf{K}, \vec{p})$.
(4) Output $\vec{c}$.

Using the definition of random variables $\mathbf{P}$ and $\mathbf{C}$, the perfect security of MuPS is formalized in Theorem 2. It basically states that if the adversary has *a priori* information on the distribution of plaintexts (i.e., $\mathbf{P}$), and he observes a valid ciphertext vector, the *a posteriori* information on the distribution of plaintexts (i.e., $\mathbf{P}$ conditioned on $\mathbf{C}$) does not change. In other words, observing a valid ciphertext vector does not entail any information, even to an infinitely powerful adversary.

Theorem 2 (Perfect Secrecy of MuPS)**.** *For all vectors $\vec{p} \in \mathsf{Good}^n_m(\mathcal{P})$ and all vectors $\vec{c} \in \mathsf{Good}^n_1(\mathcal{C})$:*

$$\Pr\left[\mathbf{P} = \vec{p} \mid \mathbf{C} = \vec{c}\right] = \Pr\left[\mathbf{P} = \vec{p}\right] . \tag{4}$$

Proof. We prove the following property, which is *equivalent* to perfect secrecy (see [37, Theorem 2.1] for a proof): For all plaintexts and ciphertexts, the number of keys mapping the former to the latter must be a constant (i.e., independent of the actual plaintext or ciphertext). In terms of MuPS notation, this means that for all vectors $\vec{p} \in \mathsf{Good}^n_m(\mathcal{P})$ and all vectors $\vec{c} \in \mathsf{Good}^n_1(\mathcal{C})$, the size of the following set is independent of both $\vec{p}$ and $\vec{c}$:

$$S(\vec{p}, \vec{c}) \overset{\text{def}}{=} \{\mathbf{K} \mid \mathsf{Enc}(\mathbf{K}, \vec{p}) = \vec{c}\} .$$

Recall from Definition 2 that $\mathbf{K}$ is a uniformly picks $\ell \cdot m$ elements from $\mathcal{C} = \{0, 1\}^\kappa$ without replacement. Therefore, $\mathbf{K} \in \mathsf{Good}^{\ell \cdot m}_1(\mathcal{C})$. For $i \in \{1, \ldots, n\}$, the

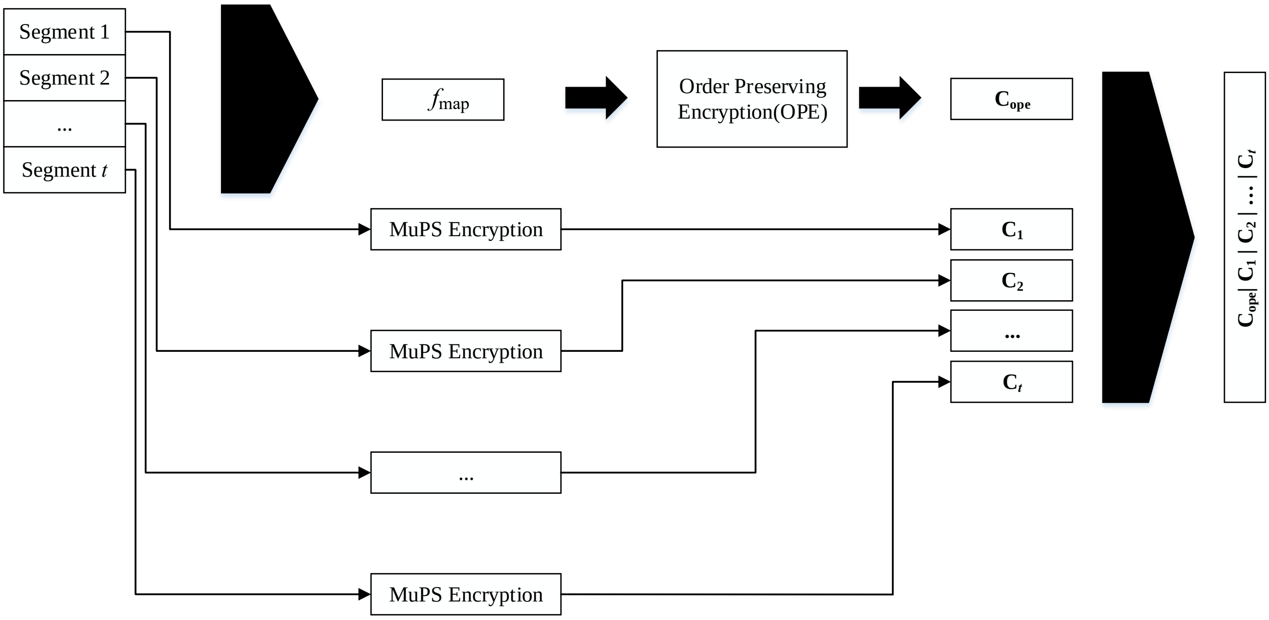

Figure 1. Graphical representation of SESOS encryption function. In the upper part of the diagram, the order-preserving encryption (OPE) of the converted value of plaintext (using f_{map}) is calculated. In the lower part, each segment is encrypted by MuPS. Finally, all ciphertexts are concatenated.

i^{th} component of $\vec{p}$ and $\vec{c}$ fixes an entry in $\mathbf{K}$. That is, $\mathbf{K}\left[\vec{p}^{(i)}\right]\left[\mathbf{st}\left[\vec{p}^{(i)}\right]\right] = \vec{c}^{(i)}$. Therefore, $S(\vec{p},\vec{c})$ is the set of keys with n fixed entries, and $\ell \cdot m - n$ free entries. Consequently,

$$|S(\vec{p},\vec{c})| = P\big(2^{\kappa} - n, \ell \cdot m - n\big)\,,$$

where $P(y,x) \overset{\text{def}}{=} \frac{y!}{(y-x)!}$ is the number of x-permutations of y. The theorem is proven by noting that $|S(\vec{p},\vec{c})|$ is independent of both $\vec{p}$ and $\vec{c}$. □

4.2 SESOS: A Searchable Outsourcing Scheme for Ordered Structured Data

Before formalizing SESOS, let us show how a "date" is encrypted. The date is composed of year (Y), month (M), and day (D) segments. Here, we assume that $Y \in \{1970, \ldots, 3000\}$, while $M \in \{1, \ldots, 12\}$ and $D \in \{1, \ldots, 31\}$. The date value can also be converted to a 32-bit integer S by counting the number of days from the Unix epoch. Here, $S \in \{0,1\}^{32}$.

SESOS encrypts a date value as follows: It uses an OPE scheme to encrypt the "converted" value of date, S. Furthermore, it encrypts each segment (Y, M, and D) using an independent instantiation of MuPS. Finally, SESOS concatenates the resulting ciphertexts. The process can be described *informally* as:

$$\text{OPE}(S) \mid \text{MuPS}(Y) \mid \text{MuPS}(M) \mid \text{MuPS}(D)$$

Decryption works the other way round: The ciphertext is decomposed into its constituent parts (it is assumed that this can be done in a unique way). The first part of the ciphertext is decrypted using OPE decryption function, while each remaining part is decrypted using MuPS decryption function for verification procedure. The concept is formalized in Definition 3.

Definition 3 (SESOS Encryption Scheme). Let $\Pi_{\mathsf{ope}} = (\mathsf{Gen}_{\mathsf{ope}}, \mathsf{Enc}_{\mathsf{ope}}, \mathsf{Dec}_{\mathsf{ope}}, \mathcal{P}_{\mathsf{ope}}, \mathcal{C}_{\mathsf{ope}})$ be any OPE encryption scheme, and $\Pi_i = (\mathsf{Gen}_i, \mathsf{Enc}_i, \mathsf{Dec}_i, \mathcal{P}_i, \mathcal{C}_i)$ be a MuPS encryption scheme for $i \in \{1, \ldots, t\}$.

A *SESOS encryption scheme using* Π_{ope} *with* t *segments over* $\mathcal{P}_1, \ldots, \mathcal{P}_t$ is a quintuple $\Pi_{\mathsf{sesos}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathcal{P}, \mathcal{C})$ defined as follows:

- **Plaintext Space:** $\mathcal{P} \overset{\text{def}}{=} \mathcal{P}_1 \times \cdots \times \mathcal{P}_t$.
- **Ciphertext Space:** $\mathcal{C} \overset{\text{def}}{=} \mathcal{C}_{\mathsf{ope}} \times \mathcal{C}_1 \times \cdots \times \mathcal{C}_t$. By definition of the underlying schemes, $\mathcal{C} = \{0,1\}^{\kappa}$ for $\kappa \overset{\text{def}}{=} \kappa_{\mathsf{ope}} + \kappa_1 + \cdots + \kappa_t$.
- **Key Generation:** On input (where κ and m are as per Definition 2):
$$\left(1^{\lambda}, \{\mathcal{P}_i, \kappa_i, m_i\}_{i=1,\ldots,t}\right),$$
the output of Gen is defined as $(\mathbf{st}_{\mathsf{sesos}}, k_{\mathsf{sesos}})$. Here, $\mathbf{st}_{\mathsf{sesos}}$ is defined as $(\mathbf{st}_1, \ldots, \mathbf{st}_t)$, and
$$k_{\mathsf{sesos}} \overset{\text{def}}{=} \left(k_{\mathsf{ope}}, \{\mathbf{K}_i, \mathbf{K}_i^{-1}\}_{i=1,\ldots,t}\right),$$
where $k_{\mathsf{ope}} \leftarrow \mathsf{Gen}_{\mathsf{ope}}(1^{\lambda})$, and for $i \in \{1, \ldots, t\}$,
$$(\mathbf{st}_i, \mathbf{K}_i, \mathbf{K}_i^{-1}) \leftarrow \mathsf{Gen}_i(1^{\lambda}, \mathcal{P}_i, \kappa_i, m_i)\,.$$
- **Encryption:** SESOS first maps the plaintext $p = (p_1, \ldots, p_t) \in \mathcal{P}$ onto Π_{ope} by applying a specific *bijection* $f_{\mathsf{map}} \colon \mathcal{P} \to \mathcal{P}_{\mathsf{ope}}$. Let $p_{\mathsf{ope}} \overset{\text{def}}{=} f_{\mathsf{map}}(p)$.

On input $\mathbf{st}_{\mathsf{sesos}}$, k_{sesos}, and a plaintext p, the SESOS encryption function Enc outputs the updated state $\mathbf{st}'_{\mathsf{sesos}} \overset{\text{def}}{=} (\mathbf{st}'_1, \ldots, \mathbf{st}'_t)$, as well as the ciphertext $c \overset{\text{def}}{=} c_{\mathsf{ope}} \mid c_1 \mid \cdots \mid c_t$ (which is the concatenation [2] of the corresponding ciphertexts). Here, $c_{\mathsf{ope}} \leftarrow \mathsf{Enc}_{\mathsf{ope}}(k_{\mathsf{ope}}, p_{\mathsf{ope}})$, and

$$(c_i, \mathbf{st}'_i) \leftarrow \mathsf{Enc}_i(\mathbf{st}_i, \mathbf{K}_i, p_i)\,,$$

for $i \in \{1, \ldots, t\}$.

The encryption procedure is graphically demonstrated in Figure 1.

- **Decryption:** On input k_{sesos}, and a ciphertext $c = c_{\mathsf{ope}} \mid c_1 \mid \ldots \mid c_t \in \mathcal{C}$, the SESOS decryption function Dec outputs $p = (p_1, \ldots, p_t) \in \mathcal{P}$. While the plaintext can be computed by either of the following strategies, the second strategy is much more efficient (see Section 5.1.2):
  (a) **Decrypting OPE:** Let $p_{\mathsf{ope}} \leftarrow \mathsf{Dec}(k_{\mathsf{ope}}, c_{\mathsf{ope}})$, and output the plaintext by inverting p_{ope} under the bijection f_{map}. That is, $p \leftarrow f^{-1}_{\mathsf{map}}(p_{\mathsf{ope}})$.
  (b) **Decrypting MuPS:** Decrypt individual MuPS by invoking Dec_i on each c_i for $i \in \{1, \ldots, t\}$. That is, $p_i \leftarrow \mathsf{Dec}_i(\mathbf{K}_i^{-1}, c_i)$.

  We define the **Extended SESOS** (**XSESOS**) as SESOS *with verification*: Let p_a and p_b be the plaintexts computed using strategies (a) and (b), respectively. XSESOS decrypts each ciphertext using both strategies, and outputs $\perp$ if $p_a \neq p_b$, and outputs $p = p_a = p_b$ otherwise.

4.2.1 Executing Comparison and Equality Queries

One of the main features of SESOS is its ability to preserve the order of plaintexts due to the underlying OPE scheme [3]. As a result, queries with comparison and equality conditions can be executed by the service provider (SP) over the encrypted data. To this end, it is required to separate the c_{ope} part of ciphertexts. This can be achieved using the `SUBSTRING (str, pos, len)` function of SQL, which returns `len` characters of `str` starting from the position `pos`. Consequently, c_{ope} can be obtained from a SESOS ciphertext c by:

$$c_{\mathsf{ope}} = \texttt{SUBSTRING}(c, 0, \kappa_{\mathsf{ope}})\,.$$

Using the following property, the SP can compare two existing ciphertexts with each other:

$$p_1 \leq p_2 \iff c^1_{\mathsf{ope}} \preceq c^2_{\mathsf{ope}}$$

As an example, suppose that the column `reg_date` contains the registration date for users. The following SQL query retrieves all users who have registered after `2000/01/01`, or have registered on `1998/05/26`, or their registration date is after their wedding date. The first two conditions demonstrate comparison with specific values, while the last condition shows the comparison between two encrypted columns.

```
SELECT * FROM tbl WHERE
(reg_date > '2000/01/01') or
(reg_date = '1998/05/26') or
(reg_date > wedding_date);
```

In order for the SP to be able to execute the query, the client software should replace the conditions as follows:

```
SELECT * FROM tbl WHERE
  (SUBSTRING(reg_date, 0, $\kappa_\ope$)
   > $\Enc_\ope$($k_\ope$, '2000/01/01')) or
  (SUBSTRING(reg_date, 0, $\kappa_\ope$) =
  $\Enc_\ope$($k_\ope$, '1998/05/26')) or
  (SUBSTRING(reg_date, 0, $\kappa_\ope$) >
  SUBSTRING(wedding_date, 0, $\kappa_\ope$));
```

Algorithm 3 Client-Side Compilation of the `WHERE` Clause in SQL Queries.

```
ConvertQuery($query$)
  $pairs \leftarrow$ ExtractSegmentNoAndTheirValues($q-uery$    // segment number/value pairs
  for each $(i, p) \in pairs$
    $(\texttt{pos}, \texttt{len}) \leftarrow$ FindTarget($i$)    // position and length of the target segment
    $newCond \leftarrow$ "SUBSTRING(col, pos, len) IN $\mathbf{K}_i[p]$"    // col is the target column in query
    $query \leftarrow$ ReplaceOldConditionWithNewOne($query, newCond$)
  return $query$
```

4.2.2 Executing `LIKE` queries

In previous OPE schemes, execution of queries with one or more `LIKE` conditions required the retrieval of all data, and decryption and filtering them at the client side. SESOS, however, can execute `LIKE` conditions on segments of structured data (the number of segments can be increased to provide fine grain search capability). Let us describe the steps required by the client software to prepare such queries for execution on the server side (see Algorithm 3). We will use the following simple SQL query to demonstrate the concepts:

```
SELECT * FROM tbl WHERE reg_date LIKE '%Feb%';
```

[2] Although storing each segment of the ciphertext in its own column is conceivable, it may be very hard to compile *nested* SQL queries into a query which is suitable for encrypted data.
[3] If the user uses a randomized OPE scheme in SESOS, then he has to use the comparison and equality conditions which are provided by the scheme

The first step is to find out the segment(s) of the structured data which can potentially be the target of the given query. This can be done by comparing the given pattern (e.g., `'%Feb%'`) with the domain of each data segment. In our example, the pattern only matches the domain for the "month" segment, which is the second segment in a six-segment timestamp format (containing years, months, days, hours, minutes, and seconds). Let ExtractSegmentNoAndTheirValues(·) be a function which, on input the query, returns a list of pairs (i, p), where i is the segment number, and p is its value. In the example above, this function returns a list with a single pair: $\{(2, \texttt{Feb})\}$.

Next, the algorithm should iterate over the list, and find the "target" for each segment number. By *target*, we mean the parameters to be passed to SQL `SUBSTRING` function (the starting position `pos`, and the length `len`). The following pseudocode demonstrates this concept:

$$(\texttt{pos}, \texttt{len}) \leftarrow \textsc{FindTarget}(i)\,.$$

In our example, $i = 2$. Therefore, $\texttt{pos} = \kappa_{\mathsf{ope}} + \kappa_1$ and $\texttt{len} = \kappa_2$.

Furthermore, it is required to transform each plaintext value p into the corresponding vector of possible ciphertexts. This vector is exactly $\mathbf{K}_i[p]$, where $\mathbf{K}_i$ is the MuPS key for segment i. For instance, assume that `Feb` can be mapped to vector $(423, 665)$. Consequently, the compiled query will be:

```
SELECT * FROM tbl WHERE SUBSTRING(reg_date ,
$\kappa_\ope + \kappa_1$ , $\kappa_2$) IN
(423 , 665);
```

The function ReplaceOldConditionWithNewOne is responsible with replacing the old condition in a `WHERE` clause with the new one. Algorithm 3 summarizes the pseudocode for compilation of `LIKE` queries.

4.3 A Numerical Example

Timestamp is one of the most popular structured data types where SESOS is applicable. As described earlier, this data type has six different segments (years, months, days, hours, minutes, and seconds). In this section, we exemplify SESOS for the timestamp `2002 Feb 23 18:50:07` to clarify the proposed encryption method.

SESOS generates random mappings for each segment. We will use the mappings shown in Figure 2 as an example. It should be noted that this example is for demonstration purposes only.

4.3.1 Encryption

The first step in encryption of a timestamp value is to map it to some value in the domain of the OPE scheme (using f_{map} in Definition 3). In this example, f_{map} computes the number of seconds passed since the Unix epoch. In our example, the output is $p_{\mathsf{ope}} = \texttt{1014490207}$.

The first part of ciphertext (c_{ope}) is computed by encrypting p_{ope} by an order-preserving encryption (OPE) scheme, which in our example is: $c_{\mathsf{ope}} = \texttt{345347336788722327395922}$.

Next, it is required to encrypt each segment of plaintext with MuPS. Assuming this is not the first plaintext being encrypted, each segment is encrypted based on some hypothetical state as follows (cf. Figure 2):

$$\begin{array}{llllll} 2002 & \rightarrow 658, & \texttt{Feb} & \rightarrow 143, & 23 & \rightarrow 645, \\ 18 & \rightarrow 240, & 59 & \rightarrow 321, & 7 & \rightarrow 023\,. \end{array}$$

The ciphertext is generated as the concatenation of the all partial ciphertexts: $c = \texttt{34534733678872232 7395922658143645240321023}$.

4.3.2 Decryption

Decrypting a given ciphertext requires splitting the ciphertext into its constituent segments, which can be done according to values of κ for each segment. In the example above, $\kappa_{\mathsf{ope}} = 24$, and $\kappa_1 = \cdots = \kappa_6 = 3$. For convenience, this example allows κs to denote the number of digits rather than the number of bits. Therefore,

$$\begin{aligned} c_{\mathsf{ope}} &= \texttt{345347336788722327395922}, \\ c_1 &= \texttt{658}, \\ c_2 &= \texttt{143}, \\ c_3 &= \texttt{645}, \\ c_4 &= \texttt{240}, \\ c_5 &= \texttt{321}, \\ c_6 &= \texttt{023}\,. \end{aligned}$$

As explained in Definition 3, two possible strategies can be applied for SESOS decryption: (a) Decrypting c_{ope} and applying f_{map}^{-1} to the result, or (b) Decrypting $c_1, \ldots, c_6$. Both strategies results in the original plaintext.

4.3.3 Executing `LIKE` Queries

Consider the following SQL query

```
SELECT * FROM tbl WHERE date LIKE '%:59%';
```

In a timestamp, the pattern `%:59%` can occur in two positions: minutes (fifth segment) or seconds (sixth segment). Therefore, in Algorithm 3, the func-

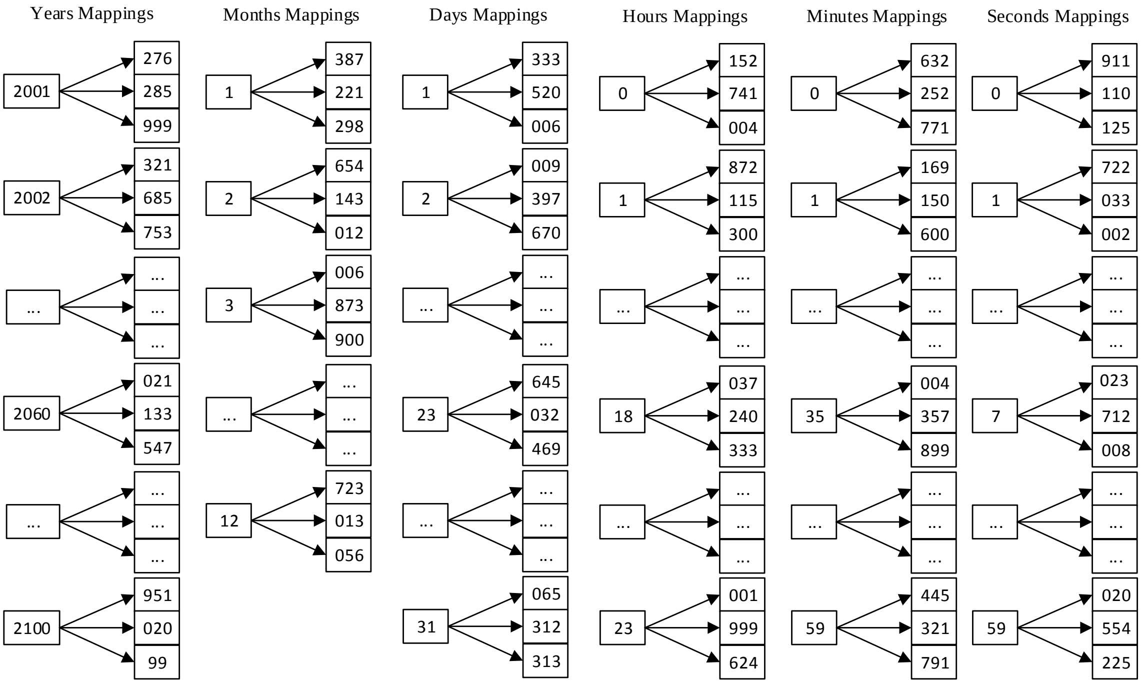

Figure 2. Sample Mapping of Each Segment of the Timestamp Structured Data.

tion ExtractSegmentNoAndTheirValues(·) outputs the list $\{(5,59),(6,59)\}$, and the query is compiled as:

```
SELECT * FROM tbl WHERE
    (SUBSTRING(date,36,3) IN (445,321,791)) or
    (SUBSTRING(date,39,3) IN (020,554,225));
```

Here, $36 = \kappa_{\mathsf{ope}} + \kappa_1 + \cdots + \kappa_4$ and $39 = \kappa_{\mathsf{ope}} + \kappa_1 + \cdots + \kappa_5$ are the starting positions for minutes and seconds, respectively. Furthermore, $(445, 321, 791)$ and $(020, 554, 225)$ are the corresponding ciphertext values for 59 as minutes and seconds, respectively (see Figure 2).

4.4 Reducing the Storage of the Client

With MuPS the size of client storage increases linearly with the number of each value's mappings (m). In this section, we propose an improved version of our scheme which eliminates such a coefficient from the storage asymptotic at the client by modifying the MuPS structure for clients with small storage. To do that, we use a pseudorandom function with a separate key for each value in eah segment and store these PRF keys at the client in a map. Most of the times, the number of records in a relation is significantly higher than each segment's domain size which means that such improvement reduces storage size significantly. E.g. consider the number of records in an activity log table containing timestamps in comparison to possible values for year, month, or minute in a timestamp field.

In the new version of SESOS, at the beginning of the system setup, the client executes $Gen_{prf}(1^\lambda)$ to generate a new random key for each segment which we call them $S^1_{key}, S^2_{key}, \ldots, S^p_{key}$ where p is the number of segments. Next, it sets a counter to zero for each value of each segment which we denote them by $Cnt^1_{S_1}$, $Cnt^2_{S_1}, \ldots, Cnt^{k_1}_{S_1}, Cnt^1_{S_2}, Cnt^2_{S_2}, \ldots, Cnt^{k_2}_{S_2}, \ldots, Cnt^1_{S_p}, Cnt^2_{S_p}, \ldots Cnt^{k_p}_{S_p}$ where k_i is the domain size of segment i. Based on these primitives, we explain how to execute the previous operations based on the PRF function G as follows. As explained earlier, there are several instances of MuPS in SESOS. For simplicity, we just explain the needed modifications for one of them while the others are similar.

Encryption The client retrieves the corresponding counter of the plaintext value for the target segment ($Cnt^{value}_{S_i}$) from the stored map. Increments the counter and computes the following PRF value to use as the output of MuPS for the target segment:$c_i = G_{S^i_{key}}(value, Cnt^{value}_{S_i})$

Executing `LIKE` Query To execute `LIKE` query, the client executes the same procedure as before except that instead of iterating over the MuPS lookup table, it evaluates $G_{S^i_{key}}(value, loopCounter)$ for $loopCounter = 1, \ldots Cnt^{value}_{S_i}$. $Cnt^{value}_{S_i}$ is the current counter of the target value. Then, it uses the achieved ciphertexts as replacement of lookup values and continues the previous procedure of SESOS.

Decryption The decrypt operation requires to evaluate all PRF values which could be appeared in each segment and find their corresponding keys. Although this computation seems to have enormous overhead, it can be done very fast by considering two points. First of all, users do not normally ask for decryption of random records. Instead, they ask for some queries which their results should be decrypted. It means that the client already knows the target keys which their PRFs should be evaluated because at the rewriting phase of the input query the client had created the corresponding ciphertext values. This number is usually very smaller than the whole relation size and can be computed very efficiently. Second, the PRF function can be computed using AES which is supported by the hardware (AES-NI) and has incredibly high performance. Indeed, even though the decryption performance does not reach the previous value (with a look-up table), it is still higher than the previously existing solutions.

Comparison We do not change the OPE part of SESOS. As the result, the comparison operation is the same as the previously explained algorithm.

As you can see, we reduced the security level of MuPS to the security level of PRF in order to decrease the needed permanent storage at the client while the performance does not change a lot.

5 Evaluation

The test scenarios were performed on an Intel® Core™ i7-5500 2.4 GHz with 4 GB of RAM on a 64-bit Ubuntu 14.04 operating system. PostgreSQL 9.3 was used as the DBMS.

The OPE scheme FOPE [23] is considered, as it is significantly faster and more practical than CryptDB's mOPE scheme. We implemented FOPE in C++ using the MPZ library [38]. We used FOPE as a deterministic OPE scheme in SESOS construction too. Note that since SESOS performs extra operations (MuPS encryptions) compared to the underlying OPE, it is expected to have some overhead. However, experiments show that the overhead is small, and in some scenarios (such as `LIKE` queries), SESOS can hugely outperform the OPE scheme.

In the remainder of this section, we first report the results of performance tests, and then pertain to the storage and network results.

5.1 Performance

In order to compare the performance of SESOS without storage improvement and FOPE, some experiments were conducted. In each experiment, the target operation was executed 100 times and its averaged value is provided as the result. The impact of record numbers on compared methods have been analyzed over different number of timestamps which were randomly generated between 1970 and 2017. As you can see, the highest value for ℓ among different segments of experimented timestamps is 60. For simplicity, we used the different $MuPS$ with the same parameters ($\ell = 60, n = 10^5$, and $\delta = 0.999$) which leads to $m = 1857$ for each segment.

Table 1. Encryption Time of FOPE and SESOS (in *Seconds*) Based On the Number of Records

# Records	FOPE	SESOS
20,000	62	65
40,000	125	126
60,000	187	188
80,000	268	272
100,000	336	342

5.1.1 Encryption

To evaluate the encryption performance, a column of table with timestamp values was encrypted by both FOPE and SESOS. Because of the small overhead of MuPS encryption (it just uses a lookup table), encryption speed of both methods are approximately equal, as expected. It means that, SESOS encryption adds negligible overhead to an existing OPE mechanisms. The results are shown in Figure 3 and Table 1.

5.1.2 Decryption

Figure 4 and Table 2 show the decryption time for FOPE, SESOS, and XSESOS. Notice that SESOS decryption is extremely faster than FOPE decryption. This is because SESOS only uses a lookup table for decryption (strategy 2), while FOPE requires an enormous amount of computation for the same purpose. The little overhead of SESOS is due to the fact that SESOS has to call the DBMS `SUBSTRING` function. As you can see, its decryption time is faster than FOPE scheme by up to 520X which is a huge improvement.

XSESOS has to deal with just a little more overhead than FOPE, since it requires to perform both decryption strategies, as well as a comparison. The extra overhead is paid off by noting that XSESOS provides verifiability as well, which is provided by neither FOPE nor SESOS.

5.1.3 Equality and Comparison Queries

Evaluation of equality and comparison queries was done by placing random timestamps in the `WHERE` clause of the respective queries. As is customary in

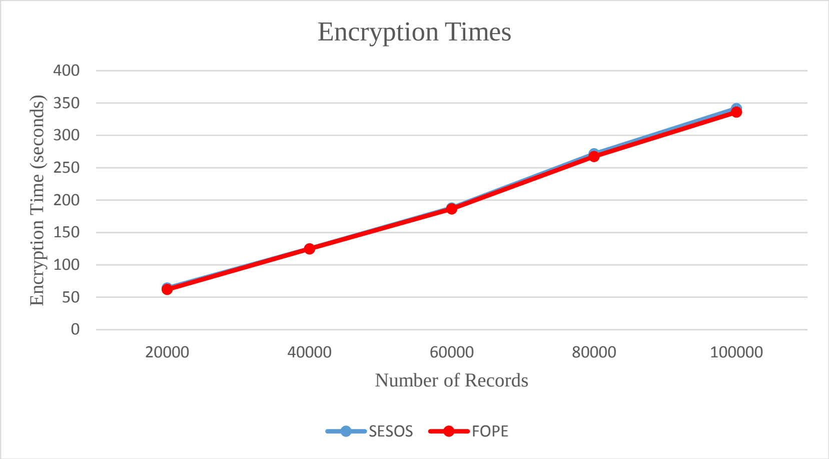

Figure 3. Graph of the Encryption Time of FOPE and SESOS (in *Seconds*) vs. the Number of Records

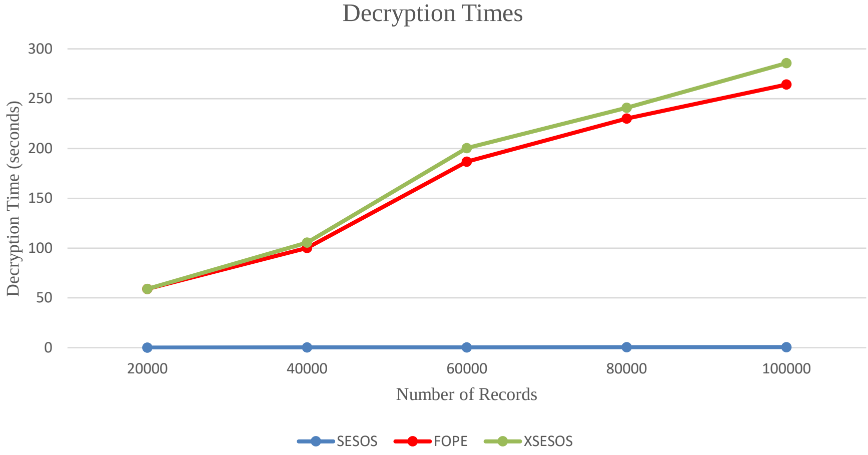

Figure 4. Graph of the Decryption Time of FOPE, SESOS, and XSESOS (in *Seconds*) vs. the Number of Records

Table 2. Decryption Time of FOPE, SESOS, and XSESOS (in *Seconds*) Based On the Number of Records

# Records	FOPE	SESOS	
20, 000	60	0.122	60
40, 000	101	0.293	106
60, 000	187	0.380	201
80, 000	231	0.443	241
100, 000	265	0.603	286

real-world scenarios, a B-Tree index was defined on encrypted values. The results of the experiments are presented in Figure 5 and Figure 6, respectively. As can be seen, the overhead of SESOS is negligible compared to FOPE. The reason for the small difference is the increase in the size of the ciphertexts, which makes string operations a little bit slower.

5.1.4 LIKE Queries

Similar to the other existing solutions, FOPE does not support LIKE queries directly. This means that the search should be performed after fetching and decrypting all target data in a trusted zone (such as a proxy). The overhead of this method depends on the total number of records, and is independent of the size of the result set.

In this experiment, LIKE queries were executed for both methods in order to retrieve the records with a specific random month of the year. Using this methodology, two different criteria were evaluated: (1) The search time required by the DBMS, and (2) the decryption time required by the proxy. As it is shown in Figure 7, SESOS database search time is negligible in comparison to the decryption time which is required by FOPE method. Furthermore, the decryption time of SESOS is proportional to number of retrieved results, while the FOPE's time is approximately constant. Figure 8 and Figure 9 show the details of Fig-

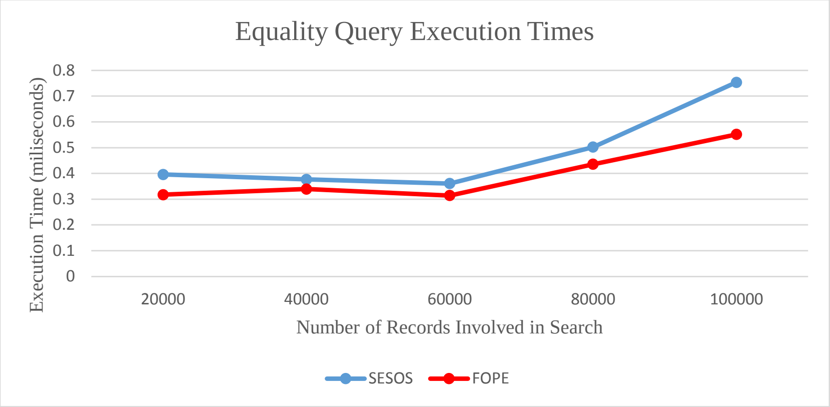

Figure 5. Execution Time of Queries with Equality Condition (in *Milliseconds*) vs. the Number of Records

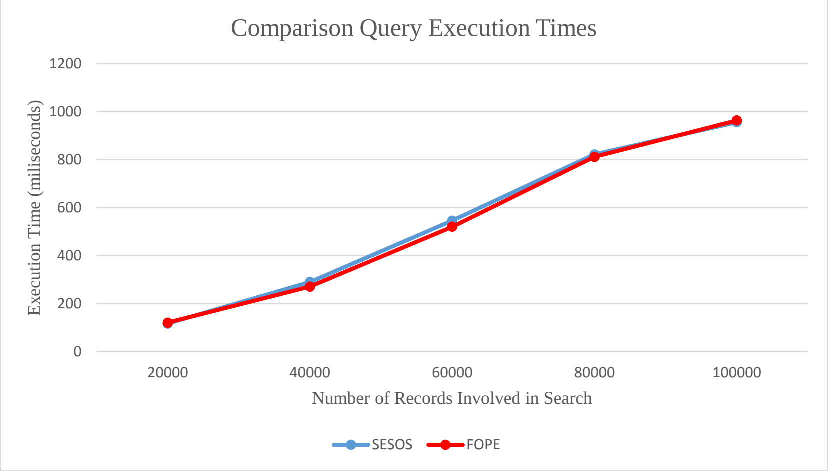

Figure 6. Execution Time of Queries with Comparison (in *Milliseconds*) vs. the Number of Records

ure 7 in separate diagrams. As it can be seen, SESOS has much better performance.

In real-world scenarios, results of LIKE queries contain only a small fraction of all records. By considering the evaluation results, it can be concluded that SESOS increases performance of LIKE queries by up to 1370X depending on the number of target results in comparison with FOPE. This result is achieved while other operations such as encryption, decryption, equality check, and comparison do not have any noticeable overhead.

5.2 Storage and Network

SESOS concatenates multiple partial ciphertexts to generate the final ciphertext. The number of segments depends on the structured data under consideration. For instance, timestamps need seven ciphertext segments (one for OPE, and six for each plaintext segment), while IPv4 addresses need five segments (one for OPE, and four for each plaintext segment). The size of SESOS ciphertext is almost twice the size of the underlying OPE, as it encompasses one OPE ciphertext plus the encryption of several small segments. Although this amount of storage may seem significant, the provided performance improvement makes such a price acceptable.

The network traffic required to transfer the result set of a SESOS query vary in different situations. All queries (except those containing LIKE) fetch the same result set as FOPE. In XSESOS, the traffic is doubled, as it requires both OPE ciphertexts *and* the MuPS ciphertexts in order to provide verifiability.

As discussed earlier, SESOS treats LIKE queries much better than FOPE, and can return a result set which is much smaller than the whole database records. Therefore, the generated network traffic of SESOS is several times lower than that of FOPE. This also holds for XSESOS. Table 3, presents a summary of network traffic comparison between FOPE, SESOS and XSESOS.

6 Future Work

In SESOS, the size of a compiled query increases when the parameter m increases (recall from Definition 2

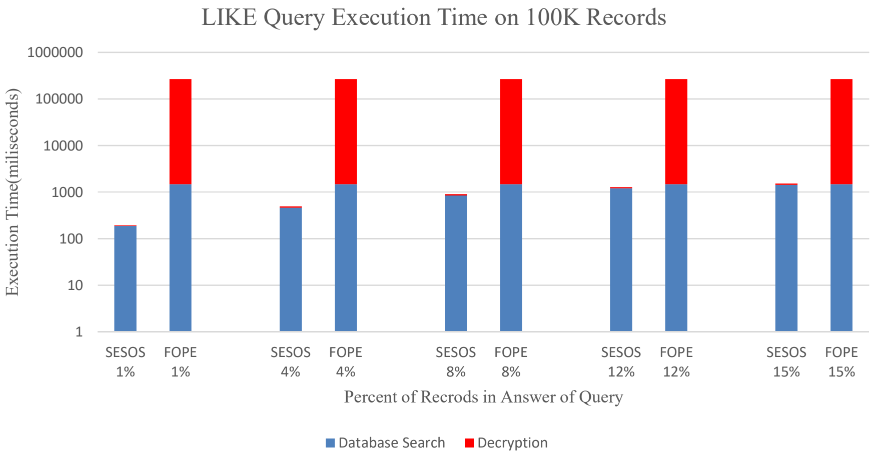

Figure 7. Execution Time of `LIKE` Queries (in *Seconds*) vs. the Percentage of Records in the Result

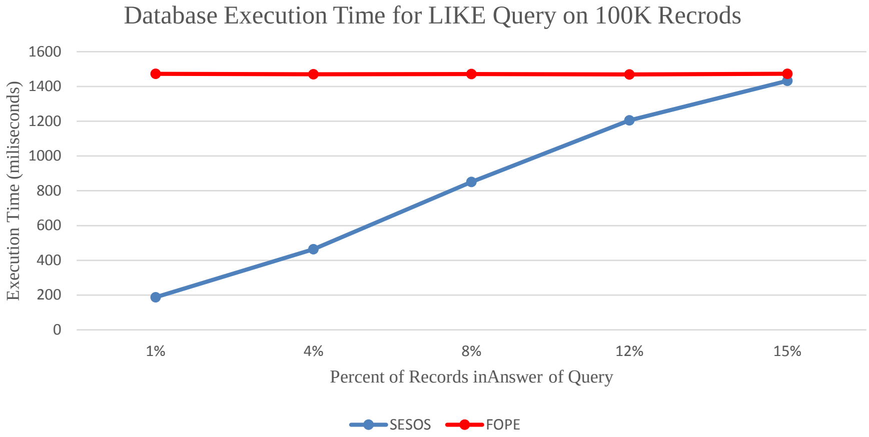

Figure 8. Decryption Time of `LIKE` Queries (in *Seconds*) vs. the Percentage of Records in the Result

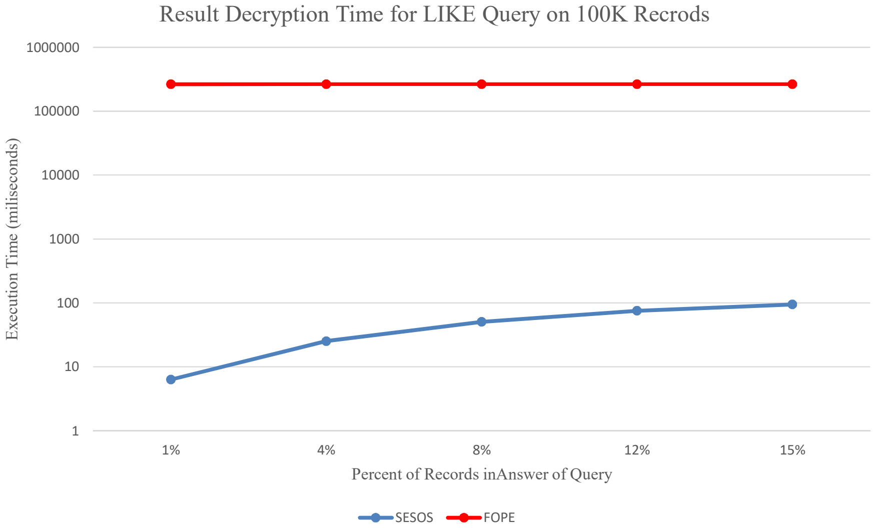

Figure 9. Database Search Time of `LIKE` Queries (in *Seconds*) vs. the Percentage of Records in the Result

that m is the number of mappings per plaintext). One idea is to retrieve the result set in a "page-by-page" manner: Instead of retrieving all m ciphertexts corresponding to a single plaintext, the client software

Table 3. FOPE Network Traffic Relative to SESSOS and XSESOS

Operation	FOPE/SESOS	FOPE/XSESOS
Equality	1	≥ 0.5
Range Query	1	≥ 0.5
`LIKE`	2 to 100	1 to 50

can fetch a subset of those results, and show them to the user. If the user asks for the next "page" of the results, the client software can retrieve another subset, and so on.

Another approach is to modify MuPS as follows: For each plaintext p, the key specifies m' subsets $\mathcal{C}_p^1, \ldots, \mathcal{C}_p^{m'}$ in the ciphertext space. The encryption function randomly maps p to a ciphertext in one of those subsets. Here, m' can be much smaller than m. Yet the modified version is no longer perfect secure. A future study can be conducted to examine the ramifications of such modification on the security of SESOS.

Another line of work is the study of query analysis. The current version of SESOS allows a service provider (SP) to cluster ciphertexts based on the mappings provided in each query. For instance, in a query such as:

```
SELECT * FROM tbl WHERE
    (SUBSTRING(date,36,3) IN (445,321,791)) or
    (SUBSTRING(date,39,3) IN (020,554,225));
```

The SP receiving the above query can be sure that ciphertexts in $(445, 321, 791)$ correspond to one plaintext p, and ciphertexts in $(020, 554, 225)$ correspond to a distinct plaintext p'. To prevent this type of analysis, random noise can be added to the `IN` clause, and the `IN` clause can be split into multiple queries (the "paging" method described above). A formal analysis of the impact of these techniques on the security of SESOS is another direction for future research.

This paper concentrated on `LIKE` operation on structured data. There exist other operations on various data types, most of which are not supported by existing encryption schemes. Furthermore, each DB encryption scheme supports only a limited set of operations, and so it is required to store a separate column per scheme to support multiple operations on encrypted data. This type of storage causes issues in executing nested queries which has to be eliminated.

7 Conclusions

In this paper, a searchable encryption scheme for ordered structured data (SESOS), and an extended variant (XESOS) were proposed. The schemes combined any order-preserving encryption (OPE) with a novel encryption scheme called MuPS. We proved the perfect secrecy of MuPS, demonstrating that it can be combined with any OPE without weakening its security properties. SESOS and XSESOS were evaluated under various criteria. It was shown that the overhead is negligible compared to the underlying OPE scheme, while it outperforms the OPE by up to 1370X and 520X in response to `LIKE` queries and decryption in a database with merely 100K records. The performance gain can be increased on larger databases. SESOS also provides extremely faster decryption compared to the underlying OPE. On the other hand, XSESOS allows for ciphertext verifiability, with negligible performance overhead with respect to the underlying OPE.

References

[1] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari 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.

[2] Stephen Tu, M Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment, Vol. 6, No. 5*, pages 289–300, 2013.

[3] Zhian He, Wai Kit Wong, Ben Kao, David Wai Lok Cheung, Rongbin Li, Siu Ming Yiu, and Eric Lo. SDB: a secure query processing system with data interoperability. *Proceedings of the VLDB Endowment*, 8(12):1876–1879, 2015.

[4] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482. Springer, 2012.

[5] Xiaoqiang Sun, Jianping Yu, Ting Wang, Zhiwei Sun, and Peng Zhang. Efficient identity-based leveled fully homomorphic encryption from RLWE. *Security and Communication Networks*, 9(18):5155–5165, 2016.

[6] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[7] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: processing queries on an encrypted database. *Communications of the ACM*, 55(9):103–111, 2012.

[8] Li Xiong, Slawomir Goryczka, and Vaidy Sunderam. Adaptive, secure, and scalable distributed data outsourcing: a vision paper. In *Proceedings of the 2011 workshop on Dynamic distributed data-intensive applications, programming abstractions, and systems*, pages 1–6. ACM, 2011.

[9] Jin Li, Zheli Liu, Xiaofeng Chen, Fatos Xhafa, Xiao Tan, and Duncan S Wong. L-EncDB:

A lightweight framework for privacy-preserving data queries in cloud computing. *Knowledge-Based Systems*, 79:18–26, 2015.

[10] Ernesto Damiani, SDCD Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 93–102. ACM, 2003.

[11] Dongmei Li, Xiaolei Dong, and Zhenfu Cao. Secure and privacy-preserving pattern matching in outsourced computing. *Security and Communication Networks*, 9(16):3444–3451, 2016.

[12] Wei Song, Zhiyong Peng, Qian Wang, Fangquan Cheng, Xiaoxin Wu, and Yihui Cui. Efficient privacy-preserved data query over ciphertext in cloud computing. *Security and Communication Networks*, 7(6):1049–1065, 2014.

[13] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong 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.

[14] Hui Wang and Laks VS Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 127–138. VLDB Endowment, 2006.

[15] Divyakant Agrawal, Amr El Abbadi, Fatih Emekci, and Ahmed Metwally. Database management as a service: Challenges and opportunities. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1709–1716. IEEE, 2009.

[16] Mohammad Ali Hadavi, Morteza Noferesti, Rasool Jalili, and Ernesto Damiani. Database as a service: towards a unified solution for security requirements. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 415–420. IEEE, 2012.

[17] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two can keep a secret: A distributed architecture for secure database services. *CIDR 2005*, 2005.

[18] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *CRYPTO*, volume 6841, pages 578–595. Springer, 2011.

[19] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa. A Secure and Efficient Order Preserving Encryption Scheme for Relational Databases. In *KMIS*, pages 25–35, 2010.

[20] George Weilun Ang, John Harold Woelfel, and Terrence Peter Woloszyn. System and method of sort-order preserving tokenization, May 2014. US Patent 8,739,265.

[21] Fahmida Y Rashid. Salesforce. com acquires SaaS encryption provider Navajo Systems. eWeek. com, 2011.

[22] Raluca Ada Popa, Frank H Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 463–477. IEEE, 2013.

[23] Yong Ho Hwang, Sungwook Kim, and Jae Woo Seo. Fast order-preserving encryption from uniform distribution sampling. In *Proceedings of the 2015 ACM Workshop on Cloud Computing Security Workshop*, pages 41–52. ACM, 2015.

[24] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 656–667. ACM, 2015.

[25] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[26] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88. ACM, 2006.

[27] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976. ACM, 2012.

[28] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 310–320. ACM, 2014.

[29] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in cryptology–CRYPTO 2013*, pages 353–373. Springer, 2013.

[30] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 351–368. Springer, 2014.

[31] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption

via blind storage. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 639–654. IEEE, 2014.

[32] Ian Miers and Payman Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. *IACR Cryptology ePrint Archive*, 2016:830, 2016.

[33] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482. ACM, 2017.

[34] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1038–1055. ACM, 2018.

[35] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, volume 5479, pages 224–241. Springer, 2009.

[36] Isamu Teranishi, Moti Yung, and Tal Malkin. Order-preserving encryption secure beyond onewayness. In *ASIACRYPT*, pages 42–61. Springer, 2014.

[37] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Draft of a book, version 0.3, December 2016. Available online from https://crypto.stanford.edu/~dabo/cryptobook/draft_0_3.pdf.

[38] The GNU Multiple Precision Arithmetic Library (GMP). Available from https://gmplib.org.

Javad Ghareh Chamani received his B.S. degree in software engineering from University of Tehran, Tehran, Iran, in 2012, and received his M.S. degree in software engineering from Sharif University of Technology,Tehran, Iran, in 2014. He is currently Ph.D. student of Sharif University of Technology,Tehran, Iran and The Hong Kong University of Science and Technology. His research interests include cloud security, database security, database outsourcing, and cloud computing.

Mohammad Sadeq Dousti got his Ph.D. from Sharif University of Technology in software engineering, and his M.S. and B.S. from Sharif University of Technology in IT engineering. He did extensive work on zero-knowledge proofs in particular, and provable security in general. He has taught several courses at the university, including applied cryptography, theory of cryptography, secure software development, and network security. His research interests include foundations of cryptography and computational complexity theory.

Rasool Jalili received his B.S. degree in computer science from Ferdowsi Universityof Mashhad in 1985, and M.S. degree in computer engineering from Sharif University of-Technology in 1989. He received his Ph.D.in computer science from University of Sydney, Australia, in 1995. He then joined the Department of Computer Engineering, Sharif University of Technology in 1995. He has published more than 140 papers in international journals and conference proceedings. He is now an associate professor, doing research in the areas of computer dependability and security, access control,distributed systems, and database systems in his Data and Network Security Laboratory (DNSL).

Dimitrios Papadopoulos is an assistant professor at the Computer Science and Engineering Department of the Hong Kong University of Science and Technology. He received a Diploma in applied mathematics from the National Technical University of Athens in 2010, a Ph.D. in computer science from Boston University in 2016 and was subsequently a Post-doctoral researcher at the University of Maryland Institute for Advanced Computer Studies. He has published multiple papers in international conference proceedings and journals. His research is focused on the development of cryptographic protocols for verifiable computation, zero-knowledge proofs, searchable/structured encryption and oblivious computation for data privacy, as well as Blockchain security.