Enabling secure and fast indexing for privacy-assured healthcare monitoring via compressive sensing

Yuan, Xingliang; WANG, Xinyu; WANG, Cong; Weng, Jian; Ren, Kui

Published in:
IEEE Transactions on Multimedia

Published: 01/10/2016

Document Version:
Post-print, also known as Accepted Author Manuscript, Peer-reviewed or Author Final version

License:
Unspecified

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1109/TMM.2016.2602758

Publication details:

Citing this paper
Please note that where the full-text provided on CityU Scholars is the Post-print version (also known as Accepted Author Manuscript, Peer-reviewed or Author Final version), it may differ from the Final Published version. When citing, ensure that you check and use the publisher's definitive version for pagination and other details.

General rights
Copyright for the publications made accessible via the CityU Scholars portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights. Users may not further distribute the material or use it for any profit-making activity or commercial gain.

Publisher permission
Permission for previously published items are in accordance with publisher's copyright policies sourced from the SHERPA RoMEO database. Links to full text versions (either Published or Post-print) are only available if corresponding publishers allow open access.

Take down policy
Contact lbscholars@cityu.edu.hk if you believe that this document breaches copyright and provide us with details. We will remove access to the work immediately and investigate your claim.
Enabling Secure and Fast Indexing for Privacy-assured Healthcare Monitoring via Compressive Sensing

Xingliang Yuan, Student Member, IEEE, Xinyu Wang, Cong Wang, Member, IEEE, Jian Weng, Member, IEEE, and Kui Ren, Fellow, IEEE

Abstract—As e-health technology continues to advance, health related multimedia data is being exponentially generated from healthcare monitoring devices and sensors. Coming with it are the challenges on how to efficiently acquire, index, and process such a huge amount of data for effective healthcare and related decision making, while respecting the user’s data privacy. In this paper, we propose a secure cloud-based framework for privacy-aware healthcare monitoring systems, which allows fast data acquisition and indexing with strong privacy assurance. For efficient data acquisition, we adopt compressive sensing for easy data sampling, compression, and recovery. We then focus on how to secure and fast index the resulting large amount of continuously generated compressed samples, with the goal to achieve secure selected retrieval over compressed storage. Among others, one particular challenge is the practical demand to cope with the incoming data samples in high acquisition rates. For that problem, we carefully exploit recent efforts on encrypted search, efficient content-based indexing techniques, and fine-grained locking algorithms, to design a novel encrypted index with high-performance customization. It achieves memory efficiency, provable security, as well as greatly improved building speed with non-trivial multi-thread support. Comprehensive evaluations on Amazon Cloud show that our encrypted design can securely index 1 billion compressed data samples within only 12 minutes, achieving a throughput of indexing almost 1.4 million encrypted samples per second. Accuracy and visual evaluation on a real healthcare dataset shows good quality of high-value retrieval and recovery over encrypted data samples.

Index Terms—privacy-aware healthcare, multimedia-based healthcare, compressive sensing, cloud computing, fast encrypted indexing.

I. INTRODUCTION

With the proliferation of e-health technology, enormous amount of multimedia-based health data is being exponentially generated from medical devices and sensors. Generally, healthcare systems demand continuous and routine monitoring to capture a deluge of information for high quality of images, pulses, and motions [1], [2]. For example, a transceiver on a body sensor network monitoring physiological signal of an individual can produce up to 31 GB data per day [3]. To manage and process such a massive scale of data, next generation e-health industry (e.g., Industry 4.0 [4]) is already turning to cloud-based services [5], linking millions of devices and sensors to economically attractive public cloud platforms [2]. On the other hand, to reduce the complexity and the energy consumption during data acquisition, unified data sampling and compression techniques like compressive sensing [1], [6]–[8] are being increasingly adopted into hardware sensors [9], [10], accelerating the generation of digital health data and saving the related resource consumption.

However, for such a promising paradigm of cloud-based health monitoring to become truly successful, there are still fundamental and critical challenges yet to be fully addressed. Firstly, health data is personal and sensitive in nature. Directly exposing them in the public cloud environment may raise concerns on possible privacy regulation violations [5], [11], since unexpected data breach incidents happen from time to time [12], [13]. Thus, strong data protection mechanisms, such as encryption, need to be enforced throughout the whole service flow for guaranteed privacy assurance.

Secondly and more critically, with advanced sampling technologies and continuously decreased hardware cost, we must be able to deal with both the large volume of health data acquired and the high velocity of data generation. For example, a wireless body sensor with compressive sensing support can compress an ECG vector in 25ms [9]. With 1,000 such sensors, 40,000 compressed samples can be easily generated in a second. Therefore, how to quickly index those samples and make them promptly available for search and utilization becomes a critical challenge. Ideally, the indexing procedure should be designed to avoid being the performance bottleneck in the overall service flow. But given the security protection, the large volume, and the common high-dimensionality of multimedia-based health data, such a practical requirement makes the index design even more challenging.

To securely index the encrypted data, one promising technique in the literature is searchable symmetric encryption (SSE) [14]–[16] (to just list a few). However, nearly all existing constructions do not consider our targeted challenge on index build efficiency when data is continuously generated at high rates. In the typical scenarios they studied, usually the encrypted index is built only once, and a few updates
are performed subsequently. Even if the one-time building cost is not very economical, it could still be tolerated. As evaluated in [16], it takes nearly 2 hours to index 1 million encrypted data records. If applied directly, the resulting indexing throughput can hardly support the incoming data sample rates originated from even one single sensor in the aforementioned wireless body network example. Besides, to ensure low latency retrieval processing and service quality, achieving good index space efficiency and good retrieval quality are also important yet practical design requirements.

**Contributions:** In this paper, we present a secure cloud-based framework for healthcare monitoring systems with privacy guarantees. The proposed design exploits advancements from different domains, including compressive sensing with unified data acquisition and compression, practical encrypted search techniques, high-performance content-based indexing, and novel concurrent programming algorithms. Our system framework and the novel indexing design enable secure and very fast indexing of continuously generated compressed data samples for health monitoring, and facilitate high-performance encrypted search over compressed samples. Both these salient features are practically desirable.

First of all, our crucial observation is to build the secure healthcare monitoring system via the generic compressive sensing framework [9], [10] for time and energy efficient data acquisition. As one remarkable feature of compressive sensing is that the Euclidean norm of raw data is still preserved in the measurement space of compressed samples [1], users such as doctors now are able to utilize and query over those compressed medical data samples for high-value results, e.g., similar samples and Top-K [17]–[19]. Another benefit of focusing on compressed samples is for reduced storage cost at cloud [17], [20]. This is because the samples alone are sufficient for high-quality data recovery after selected retrieval [19], saving the extra cost of storing raw data. For protection of the sensitive health data, the compressed data samples must be first transmitted to a trustworthy gateway server for encryption before being outsourced to the cloud.

Moving to the encrypted domain, to securely index and query the high-dimensional encrypted compressed samples, we note recent efforts, e.g., [21]–[23], on the combination of SSE with locality-sensitive hashing (LSH), a randomized approximation algorithm for similarity search in high-dimensional space [24]. Those studies leverage the fact that SSE can be useful as long as one can associate the datasets with corresponding keywords. Despite the functionality, none of them are able to achieve all the performance requirements on securely indexing a large amount of encrypted medical data samples generated in high rates, because these studies on searching encrypted data focus on application scenarios, where building the encrypted index is a one-time setup and a long building procedure is acceptable. Although recent efforts on the combination of LSH with advanced hash-based structures [22], [23] show promising results on index space efficiency, the encrypted index building algorithm without concurrent support does not adequately achieve the desired performance, i.e., fast turnaround between the time newly encrypted data samples arrive, and the time they become available for secure selected retrieval.

In order to accomplish secure and fast indexing of encrypted health data samples, we propose a new algorithm to build a high performance encrypted index with non-trivial multi-threading support. In particular, we explore a well-known lock-free technique called optimistic locking [25], [26] to accomplish high throughput of indexing, i.e., enabling multiple insertion in parallel while guaranteeing thread safety. We show how the design logic of optimistic locking can carefully be brought into the security framework of SSE and the advanced LSH indexing techniques. Essentially, our proposed concurrent index building algorithm achieves the optimal number of locks for write operations via embedding optimistic locking for read operations, which avoids intensive use of locks for the read when seeking multiple buckets in each insertion. The resulting encrypted index achieves provable security, superior load factors, and fast building speed.

Based on the construction of the encrypted index, we propose a secure retrieval algorithm that enables the cloud server to securely return encrypted high-value data samples, such that neither the sample set nor the request of retrieval are revealed. Meanwhile, it also supports concurrent lookup. For practical considerations, we improve the index building performance via caching, reduce the bandwidth cost of secure retrieval algorithm, and present a secure bulk update mechanism to handle continuously generated medical data samples. Regarding the quality of services, we further perform parameter analysis on the tradeoff between accuracy and space efficiency. The retrieval accuracy can be controlled for applications with different requirements.

We implement the proposed system, utilizing a widely recognized algorithm CoSaMP [6] in our compressive sensing module for data acquisition and recovery, and adopting standard encryption schemes for data sample encryption and index encryption. We conduct comprehensive experiments on Amazon Cloud. The performance evaluation shows that it takes 12 minutes to index 1 billion data samples, achieving a throughput of indexing 1.4 million samples per second. The accuracy and visual evaluation on a real healthcare related image set shows good quality of retrieval for high-value results over encrypted image samples.

**Organization.** Section II articulates the system architecture and threat models. Section III elaborates the proposed design. Section IV presents the implementation and experimental evaluation. We discuss the related work in Section V and
conclude in Section VI.

II. PROBLEM STATEMENT

A. System Architecture and Threat Model

The envisioned system architecture is depicted in Fig. 1. There is a gateway server operated by the data owner. It continuously collects data samples from sensing devices within the generic compressive sensing framework [9], [10], where the raw data is acquired and compressed in a uniform way with low energy cost. To relieve the local storage overhead, data owners, such as hospitals and healthcare monitoring applications, choose to outsource these compressed data samples to the cloud service provider. Because all pairwise distances between raw data objects are well preserved in the measurement space of compressed samples [1], data retrieval services can directly be enabled over the compressed data for high-value results [17]–[19]. Data users such as doctors can send the request of interests to retrieve similar samples and reconstruct the raw data objects at their local side for further investigation and analytics like disease diagnosis.

In our model, the security threats primarily come from the cloud environment. We assume that the cloud service providers will honestly follow our proposed service protocol, but they would be interested in the health data. Besides, once the cloud servers are hacked, the confidentiality of health data would also be compromised. As known, health data usually contains data sensitive or personal private information, and the compressed data still captures the information of original health data, which can further be reconstructed. Therefore, data samples should be first collected by the data owner for encryption, and then outsourced to cloud servers.

From a high level point of view, our system operates in the following procedure. The gateway server receives each incoming data sample \( s \) acquired from medical sensors or devices via compressive sensing. Then it encrypts the samples with the private key \( k \) of the data owner, and sends them to the cloud. To enable secure retrieval over encrypted samples, the gateway server builds an encrypted searchable index \( I \) for a bunch of data samples. For secure retrieval of interests, authorized data users generate tokens \( t \) from the query sample \( s_q \), and the cloud processes these tokens over the encrypted index to obtain the \( ids \) of matching samples. To handle continuously collected samples, the above index building procedure is invoked periodically. In order to make the large amount of newly generated samples promptly available for private healthcare related services, the speed of building encrypted indexes must practically be fast.

If data reconstruction is further demanded, we assume that the data user performs the reconstruction after decryption at the local side. We are aware that existing techniques of privacy-preserving data reconstruction via compressed sensing [18], [20], [27] can be integrated in our framework to remove the local computational burden of data recovery. More details will be discussed in Section V. Note that our proposed system framework adopts compressive sensing for efficient data acquisition and compressed storage, while preserving high-value result retrieval over compressed samples. Known compressive sensing algorithms that are embedded on medical devices can readily be utilized in our framework. Our design mainly focuses on, after the samples are collected, how to enable the gateway server to index those samples securely and promptly, and how to provision a controlled ability for the cloud to leverage those encrypted samples.

We also remark that cryptographic tools like broadcast encryption [14] and oblivious pseudo-random functions [28] can be used for authorizing data users. And encryption-based access control like attributed-based encryption [29] can also be utilized for data sample encryption with enforced access control. Those studies are not the focus in this work.

B. Design Goals

The proposed design is to provide the strongest possible protection on the data privacy while simultaneously maintaining the service efficiency and quality for huge amount of increasingly generated health data in healthcare monitoring systems. Our design goals are listed as follows:

1) Security guarantees: the system should ensure strong protection on data samples and query samples during the service flow. The cloud should never learn the content of data samples from the views it observes.

2) Fast index building speed: the encrypted index should be built in a reasonably short time even for billions of data samples. Namely, the throughput of indexing new samples should be at least comparable to the throughput of data acquisition from healthcare monitoring devices.

3) Space efficiency: the encrypted index should achieve optimal space complexity, compact index size, and high load factors, e.g., 90%, so that the resulting index can relatively be easy to be fit into physical memory of commodity servers for low latency query processing.

4) Query efficiency: the complexity of the proposed secure retrieval algorithm should be sublinear, and the query latency and bandwidth should be bounded.

5) Controllable accuracy: the tradeoffs between accuracy and efficiency should be understood. The retrieval accuracy should be controlled via tunable parameters.

C. Preliminaries

Cryptographic primitive: A private-key encryption scheme is a tuple of probabilistic polynomial-time algorithms (\( \text{KGen, Enc, Dec} \)). The key generation algorithm \( \text{KGen} \) takes a security parameter \( \lambda \) to return a secret key \( k \) that satisfies \(|k| > \lambda \). The encryption algorithm \( \text{Enc} \) takes a key \( k \) and a message \( m \in \{0,1\}^* \) to return a ciphertext \( c \in \{0,1\}^* \). The decryption algorithm \( \text{Dec} \) takes \( k \) and \( c \) to return \( m \). Define a family of pseudo-random functions (PRF) \( F : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \), if for all probabilistic polynomial-time distinguishers \( \mathcal{D} \), \( \Pr[\mathcal{D}^F(k,\cdot)(1^n) = 1] - \Pr[\mathcal{D}^{\mathcal{F}^{\perp}}(1^n) = 1] < \text{negl}(n) \), where \( \text{negl}(n) \) is a negligible function in \( n \), \( k \) is a uniform choice of \( \text{Func}_n \).

Compressive sensing: Many data objects acquired from the physical world are sparse when expressed in the proper orthonormal basis. Compressive sensing is a new sampling/sensing paradigm that exploits the data sparsity [1].
Considering an $n \times 1$ sparse data object $b$, the sampling process, i.e., taking compressed samples, is done by multiplying an $m \times n$, $m \ll n$, selecting matrix $R$ with full row rank to $b$ that derives an $m \times 1$ sample vector $s = Rb$. For $b = Vx$, where $V \in \mathbb{R}^{m \times n}$ is the basis and $x$ are the coefficients, we have $s = Ax$ for $A = RV$. Compressive sensing also attempts to recover $b$, or equivalent $x$, from $s$. According to [1], $\delta_{2e}$ is the $2e$-th restricted isometry constant of a matrix $A$. For $\forall \|x\|_0 \leq 2e$, one can have $(1-\delta_{2e})\|x_1-x_2\|_2^2 \leq \|Ax_1-Ax_2\|_2^2 \leq (1+\delta_{2e})\|x_1-x_2\|_2^2$. This condition implies that when $\delta_{2e} \ll 1$, all pairwise $\ell_2$ distances between sparse data are well-preserved in the measurement space. As a result, the similarity between original data objects can be effectively evaluated on the compressed data samples. Based on this property, content-based retrieval can directly be performed over the compressed data samples, as shown in plaintext designs [17], [19]. We note that such treatment is also consistent with a line of work on using random projection for multimedia retrieval [30], [31].

**Locality-sensitive hashing**: Generally, content-based retrieval is supposed to return similar data points to the query point. Since compressed data samples are high-dimensional, we resort to LSH, a randomized approximation algorithm for efficient similarity retrieval in high-dimensional spaces [32]. The idea is to hash data samples via a family of “distance-preserving” functions, where similar ones have hash collisions with a higher probability than those that are far apart.

**Definition 1** (Locality-sensitive Hashing). Let $\mathcal{S}$ be the domain of data points and $\text{dist}$ be the distance function. Given distance $R_1$, $R_2$, where $R_1 < R_2$, and probability $p_1$, $p_2$, where $p_1 > p_2$, a hashing function family $\mathcal{H} = \{h : \mathcal{S} \rightarrow \mathcal{U}\}$ is $(R_1,R_2,p_1,p_2)$-locality-sensitive if for any $s_i, s_j \in \mathcal{S}$:

- if $\text{dist}(s_i, s_j) \leq R_1$ then $\Pr[h(s_i) = h(s_j)] \geq p_1$;
- if $\text{dist}(s_i, s_j) > R_2$ then $\Pr[h(s_i) = h(s_j)] \leq p_2$.

Given a set of LSH functions, one can amplify the gap between the high probability $p_1$ and the low probability $p_2$ by concatenating several functions [32]. Accordingly, the composite LSH function family is defined as $\mathcal{G} = \{g : \mathcal{S} \rightarrow \mathcal{U}^m\}$ such that $g(s) = (h_1(s), \ldots, h_m(s))$, where each function contains $m$ random LSH functions. In general, $l$ composite LSH functions will be selected: $\{g_1, \ldots, g_l\}$. Consequently, similar data points can be pre-grouped and later retrieved based on the matching of composite LSH hash values.

### III. THE PROPOSED SYSTEM

In this section, we will propose a privacy-assured healthcare system that supports secure and fast indexing on encrypted health data samples. As mentioned before, our design is based on salient properties of compressive sensing. That is, the Euclidean norm of raw data is preserved in the compressed sample space [1], [17]–[19]; besides, compressed samples allow for high quality health data recovery after selected retrieval. In our system, we treat compressive sensing in a blackbox manner. Common compressive sensing algorithms such as ROMP [1] and CoSaMP [6] (just to list a few) can readily be implemented to acquire and generate data samples for the gateway server of data owner.

To effectively utilize those encrypted samples, we first summarize a number of common techniques that have been adopted by existing studies to construct encrypted content-based indexes, and point out their insufficiencies for our targeted problem on fast index build. Then we propose a novel index algorithm that facilitates very fast building time with non-trivial concurrency support, while preserving security guarantees. Such optimization greatly improves the indexing throughput, and paves the way to handle incoming health data samples acquired from healthcare sensors in high rates.

After that, we propose a parallel secure retrieval protocol, and show how to handle continuously generated health data with practical considerations. To ensure the service quality, we further perform analysis on retrieval accuracy and index efficiency, and figure out the method to control the accuracy. Formal security definitions and proofs are also given to demonstrate the strong protection on sensitive health data.

### A. Overview of Design Challenges

Compressive sensing greatly improves the efficiency of health data acquisition and enables healthcare sensors to generate compressed data samples at high rates. Correspondingly, healthcare systems are demanded to fast index new samples and make them become available for retrieval and utilization in a short time, while respecting user’s data privacy. In addition, the encrypted index should have advantages such as space and time efficiency to serve a large amount of health data. Note that compressed data samples are generally high-dimensional data points, and thus we focus on LSH-based indexes for efficient similarity retrieval in high-dimensional spaces. Before we introduce our proposed designs, we quickly overview a number of common techniques utilized by recent works for efficient content-based retrieval on both encrypted [22], [23] and non-encrypted domains [33], [34]. Understanding their insufficiencies will guide us to an optimized design with non-trivial concurrency support.

Specifically, for index load balancing, multi-choice hashing is shown to be working well with LSH-based similarity retrieval [34]. In particular, multiple hash tables are created, and each of them is associated with a random composite LSH function. Since each LSH hash value may match a large number of data samples, probing is applied to seek available buckets in each of these hash tables [33]. For further improved space efficiency, cuckoo hashing, a variant of multi-choice hashing, can also be employed, which relocates those samples across different hash tables to make the index achieve even higher occupancy, e.g., 90%.

In the encrypted domain, to insert a sample in a secure manner, all the possible probing buckets are determined via one-way transformed LSH hash values of the corresponding data sample [21]–[23]. After all the samples are inserted, the entire index is encrypted and uploaded for use. Usually for a compact size, $\text{id}s$ of data samples are stored in the index, and encrypted data samples are stored separately. Each $\text{id}$ can be a pointer or a pseudonym that links to an encrypted sample.

**Importance of fast indexing**: As mentioned, those common techniques in the literature do not necessarily achieve fast
ideally, the indexing throughput should be at least comparable to the incoming data sample rates, so as to avoid being the bottleneck of the overall service flow. But quickly building the high-performance encrypted index is a non-trivial task. Challenges exist when space efficiency and fast indexing should both be achieved. In particular, cuckoo hashing used for high index load factors will relocate {	extit{id}s} frequently. And multiple-choice hashing and random probing techniques require each {	extit{id}} insertion to access multiple buckets. Therefore, the direct adoption of multiple threads for the building procedure fail to achieve the thread safety, because different threads for insertion will possibly access the same index buckets.

Designing an efficient and concurrent index building algorithm encounters the following issue. During the insertion for a given {	extit{id}}, a set of probing buckets are read for available buckets. But meanwhile, another thread may write one of the buckets. Once the bucket is empty, it could be updated twice, and this insertion will fail since the {	extit{id}} inside is overwritten. And when some {	extit{id}} is to be reinserted in cuckoo hashing, a set of probing buckets are read for re-insertion as well. Therefore, different threads would also access and write a certain bucket at the same time.

The basic approach is to utilize locks to synchronize different threads. As long as each bucket is accessed exclusively, the correctness can be guaranteed. However, the number of buckets that need locking could be very large. Take the previous work [22] for example. In their approach, each insertion trial will read up to several hundreds buckets empirically, for just 1 million dataset. As a result, the locks will be used intensively, incurring substantial overheads.

Optimistic locking: We note that it is inevitable to use locks to make the write operation atomic. However, we observe that it is not necessary to lock the buckets being read. As long as the thread does not read the dirty data, then the overwrite will be avoided. If the bucket is currently being updated, the read thread does not read the dirty data, then the overwrite will be avoided. If the bucket is currently being updated, the read thread does not read the dirty data, then the overwrite will be avoided. If the bucket is currently being updated, the read thread does not read the dirty data, then the overwrite will be avoided.

B. Secure and Concurrent Index Building Algorithm

ideally, the indexing throughput should be at least comparable to the incoming data sample rates, so as to avoid being the bottleneck of the overall service flow. But quickly building the high-performance encrypted index is a non-trivial task. Challenges exist when space efficiency and fast indexing should both be achieved. In particular, cuckoo hashing used for high index load factors will relocate {	extit{id}s} frequently. And multiple-choice hashing and random probing techniques require each {	extit{id}} insertion to access multiple buckets. Therefore, the direct adoption of multiple threads for the building procedure fail to achieve the thread safety, because different threads for insertion will possibly access the same index buckets.

Designing an efficient and concurrent index building algorithm encounters the following issue. During the insertion for a given {	extit{id}}, a set of probing buckets are read for available buckets. But meanwhile, another thread may write one of the buckets. Once the bucket is empty, it could be updated twice, and this insertion will fail since the {	extit{id}} inside is overwritten. And when some {	extit{id}} is to be reinserted in cuckoo hashing, a set of probing buckets are read for re-insertion as well. Therefore, different threads would also access and write a certain bucket at the same time.

The basic approach is to utilize locks to synchronize different threads. As long as each bucket is accessed exclusively, the correctness can be guaranteed. However, the number of buckets that need locking could be very large. Take the previous work [22] for example. In their approach, each insertion trial will read up to several hundreds buckets empirically, for just 1 million dataset. As a result, the locks will be used intensively, incurring substantial overheads.

Optimistic locking: We note that it is inevitable to use locks to make the write operation atomic. However, we observe that it is not necessary to lock the buckets being read. As long as the thread does not read the dirty data, then the overwrite will be avoided. If the bucket is currently being updated, the read operation will skip this bucket and continue to the next bucket. Accordingly, we propose the following idea, inspired by the technique called optimistic locking [25], [26], a well-known lock-free technique in concurrent programming, to eliminate

Algorithm 1: Concurrent index building function Build()

<table>
<thead>
<tr>
<th>Data:</th>
<th>{k_1, k_2}: private keys;</th>
<th>s = {s_1, \ldots, s_n}: the set of data samples;</th>
<th>{g_1, \ldots, g_l}: LSH functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>I: the encrypted index.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I ← create_shared_memory([n/l/load_factor]);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion Thread 1:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>i ← 0 to n/p_w − 1 do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert(k_1, s_i);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread p_w:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>i ← (p_w − 1)n/p_w to n − 1 do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert(k_1, s_i);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>∀A_j ∈ I for j = 1 to l:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>i ← 0 to A_j.size() − 1 do</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A_j[i] ← Enc(P(k_2, g_j(s)), id)/bucket encryption</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2: One {	extit{id}} insertion: this example index includes 3 hash tables A_1, A_2, A_3 associated with l = 3 composite LSH functions g_1, g_2, g_3 respectively. F(t_g) denotes the one-way transformed composite LSH value. The probing step d is set as 2, which determines that each distinct LSH value maps to 2 random buckets, i.e., A_j[F(t_g, d)] for d from 1 to 2, where F is a secure PRF. Note that before the final update on a bucket, tag of this bucket will be validated for thread safety.
lock the bucket; 2
lock the bucket; 0
increment tag; 0
increment tag; 0
find a bucket; 1
determine tag; 1
increment tag; 1
place id; 1
increment tag; 1
unlock the bucket; 2

(a) Case one: when thread \( j \) accesses the same bucket, \( \text{tag} \) is found as an odd number, which indicates that the bucket is currently updated by another thread. Thus, thread \( j \) skips the bucket.

<table>
<thead>
<tr>
<th>Step</th>
<th>Thread ( i )</th>
<th>Thread ( j )</th>
<th>( \text{tag} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>find a bucket;</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>validate ( \text{tag} );</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \text{vtag} \leftarrow \text{tag} );</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>lock the bucket;</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>( \text{tag} \leftarrow \text{vtag} );</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>increment ( \text{tag} );</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>find a bucket;</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>validate ( \text{tag} );</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>place ( \text{id} );</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>increment ( \text{tag} );</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>unlock the bucket;</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

(b) Case two: after \( \text{tag} \) is checked and before the bucket is locked by thread \( j \), thread \( i \) finishes the update of this bucket. To avoid overwriting, \( \text{tag} \) is checked again. As the update in thread \( i \) increments \( \text{tag} \), \( \text{vtag} \) is not equal to \( \text{tag} \). Thread \( i \) will unlock and skip the bucket.

TABLE I: Flow of concurrent insertion. This example shows that how thread-safe write operations are performed when two threads \( i \) and \( j \) access the same bucket.

a large number of threads for read.

Our design is to use a variable \( \text{tag} \) to track the state of each bucket, i.e., checking whether the bucket is currently updated by another thread or not. Each \( \text{tag} \) is initialized as 0. When a thread begins to write a bucket, its \( \text{tag} \) will be incremented by 1. After the write, it will be incremented by 1 again. Then for each bucket read, the \( \text{tag} \) of the bucket will be checked. If it is an odd number, there is an on-going write operation on this bucket, and the read operation will skip this bucket. And if it is an even number, the bucket will be locked. In addition, the \( \text{tag} \) will be double checked to guarantee the thread-safe write operation, in case that another thread updates the bucket before the lock. When the write operation is completed, the bucket will be unlocked. Through this technique, each thread only creates one single lock for the bucket to be written, and the overhead is minimized.

**Construction:** The detailed encrypted index construction is presented in Algorithm 1, Algorithm 2, and Algorithm 3. And an illustrative example is also shown in Fig. 2. In Algorithm 1, we first create a piece of shared memory which can concurrently be accessed by multiple threads. Then we evenly split the input data samples \( \{s_1, \ldots, s_n\} \) into partitions, and assign each partition to total \( p_n \) threads respectively for insertion.

After all \( \text{ids} \) are inserted, the buckets are then encrypted.

Algorithm 2 shows the procedure of one \( \text{id} \) insertion of a certain data sample \( s \). In the shared memory, \( l \) hash tables \( \{A_1, \ldots, A_l\} \) are initialized, where each of them is associated with a LSH composite hash function in \( \{g_1, \ldots, g_l\} \) respectively. The possible probing buckets for \( \text{id} \) in a certain hash table \( A_i \) are located via \( F(t_{g_i, j}) \) for \( j \) from 1 to \( d \), where \( t_{g_i} = P(k_1, g_i(s)) \), \( d \) is the probing step, \( P \) and \( F \) are secure PRFs. Those probing buckets in each hash table will be scanned until an empty bucket is found.

When an empty bucket is found, its \( \text{tag} \) will be checked as seen in Line 6 of Algorithm 2. If it indicates that there exists another thread writing the bucket, it will skip the bucket and keep on scanning the next probing bucket. Otherwise, the bucket will be locked for the current thread. Before the final write operation, \( \text{tag} \) will be checked again, comparing its previously copy \( \text{vtag} \), to ensure that the bucket is not modified by any other thread before the lock. If \( \text{tag} \) and \( \text{vtag} \) are same, \( \text{tag} \) will be incremented by 1 to notify other threads that the bucket is currently updated. Then the write operation will be executed, and \( \text{tag} \) will be incremented again. Otherwise, if \( \text{tag} \) and \( \text{vtag} \) are different, which indicates that the bucket is updated by another thread, the bucket will be skipped, and the lock on this bucket will be released. In Table I, we summarize the flow of concurrent insertion.

As long as the corresponding probing buckets in every hash table are occupied, the CuckooKick operation presented in Algorithm 3 will be invoked. In particular, each insertion scans \( d \) probing buckets in \( l \) hash tables. Then one of these probing buckets is randomly selected for relocation. Before replacing the \( \text{id} \) inside with the \( \text{id} \) to be inserted, \( \text{tag} \) will also be checked twice to make sure that this bucket is not updated by any other thread.

After all samples are indexed, the entire index will be encrypted so that all buckets are protected and indistinguishable. This step is similar to all existing SSE schemes. Here, each \( \text{id} \) in the bucket is encrypted by using the corresponding one-way transformed LSH hash value \( P(k_2, g_j(s)) \) as the encryption key,i.e., \( \text{Enc}(P(k_2, g_j(s)), \text{id}) \), which allows the cloud server to securely and correctly obtain the matching \( \text{ids} \) in the subsequent retrieval.

**Caching the probing step:** We further utilize caching to reduce security overhead during the insertion. In one \( \text{id} \) insertion, \( d \) probing buckets will be checked in each hash table via computing \( \{F(t_{g_1, 1}), \ldots, F(t_{g_l, d})\} \) as shown in Line 3 of Algorithm 2. Such random probing technique ensures the protection of LSH hash values, but introduces expensive computational cost, because those probing buckets will be scanned from the beginning in every insertion trial. When the load of the index grows, seeking available buckets consumes redundant computation, since the first few buckets are occupied. Moreover, CuckooKick forces the \( \text{id} \) re-insertion, and thus the growing index load will greatly increase the overheads.

To speed up the insertion, we propose to cache the current probing step for every distinct \( t_{g_i} \) in a dictionary, so the time complexity of seeking an available bucket achieves optimal, \( O(1) \). We note that the cache size scales with the number of
the distinct LSH values, which is data dependent, and related to the density of datasets and the parameters of LSH functions. In our experiment shown in Section IV, the cache consumes less than 6MB for 1 million data samples.

Avoiding deadlocks: There exists a condition that the corresponding buckets for a certain write operation are all locked by other threads. In that case, the thread will wait for one of those buckets to be unlocked. And the thread might keep waiting if all those buckets are continuously locked, which results in a deadlock. Given total \( p_w \) threads and \( ld \) buckets in the read operation of each thread, we observe the fact that if the number of threads is less than the number of accessed buckets in each read, i.e., \( p_w < ld \), the deadlock will not happen because each thread only locks one bucket at a time, and there must exist at least \( ld - p_w + 1 \) buckets without locks. Thus, the write operation can either find an empty bucket or select a bucket for CuckooKick within those unlocked buckets. In the implementation, we follow the above observation by setting \( p_w < ld \) to avoid deadlocks.

C. Secure Retrieval and Secure Bulk Update

Secure retrieval: When encrypted health data samples and encrypted indexes are uploaded to the cloud for use, our system should enable the users to utilize those samples while providing privacy assurance. For example, a doctor wants to find similar health data of a patient in a cloud-based healthcare database without exposing the patient’s data and the database.

Based on the construction of the encrypted index, we propose the secure retrieval operation in Algorithm 4 to serve the above purpose on usability and security. For a given data sample \( s_q \) as the retrieval of interests, the authorized user first computes two sets of tokens \( \{t_{g_i}, r_{g_i}\} \) shown in line 2 and line 3. \( t_{g_i} \) is used for the cloud server to find the buckets in \( A_i \) with matching \( ids \) inside. \( r_{g_i} \) is used to decrypt the buckets. Because these two tokens are generated from PRFs on the input of composite LSH values of \( s_q \), the correctness is guaranteed. The retrieved encrypted data samples have matched LSH values with \( s_q \). And the security is also ensured, since the cloud server learns no useful information about the user request and the content of returned samples. Only with valid tokens, matched \( ids \) can be located and decrypted.

Since cuckoo hashing is a variant of multi-choice hashing, the encrypted index readily supports concurrent lookup. In the implementation, we assign total number of \( p_r \) threads to access the hash tables in parallel. We note that no lock is required in the parallel retrieval operation. Once an encrypted index is built, no more updates will be performed on this index. Besides, each secure retrieval operation accesses at most \( l \times d \) buckets, so the time complexity and bandwidth complexity are bounded, \( O(ld) \).

In our design, the user client is required to validate the results by checking if the Euclidean distances between the returned data samples and the query sample are within the pre-defined distance threshold. The goal is to filter the false positives of LSH. Here, the distance computation cost is marginal, since the number of matched result samples is bounded, e.g., several hundreds of samples as shown later in Section IV-B. Besides, the user client is asked to recover the medical data if the application further demands the reconstruction. We note that existing privacy-preserving image recovery techniques [18], [20], [27] can readily be adopted to reduce the client reconstruction cost, which enable the cloud server to securely recovery the original data from compressed samples without learning either the data samples or the original data. More discussion is presented in Section V.

Secure bulk update: For newly collected health data samples, we adopt secure bulk update, building new indexes in a streaming manner. For security purposes, fresh keys will be
Algorithm 4: Secure retrieval operation Retrieval()

Data: $s_q$: the data sample of the interest of retrieval;
Result: $\{s^*\}_i$: the encrypted high-value data samples.

\begin{algorithmic}
\State \textbf{AUTHORIZED USER:}
\For{$i \leftarrow 1$ to $l$}
\State $t_{g_i} = P(k_1, g_i(s_q));$
\State $r_{g_i} = P(k_2, g_i(s_q));$
\EndFor
\State \textbf{CLOUD SERVER:}
\State \textbf{Thread 1:}
\For{$i \leftarrow 0$ to $l/p_r - 1$}
\For{$j \leftarrow 1$ to $d$}
\State $id \leftarrow \text{Dec}(r_{g_i}, A_i[F(t_{g_i}, j)]);$
\EndFor
\State $\cdots$
\EndFor
\State \textbf{Thread $p_r$:}
\For{$i \leftarrow (p_r - 1)/p_r$ to $n - 1$}
\For{$j \leftarrow 1$ to $d$}
\State $id \leftarrow \text{Dec}(r_{g_i}, A_i[F(t_{g_i}, j)]);$
\EndFor
\EndFor
\State // fetch ciphertext from $ids$
\State return $\{s^*\}_i$;
\end{algorithmic}

used, i.e., $\{k^1_i, k^2_i\}$ for the $i$th index. Specifically, we have to consider the storage capacity of the gateway server to avoid the storage overload during the bulk update.

In our implementation, a threshold $\alpha$ is pre-defined. When the number of incoming data samples reaches $\alpha$, Build function will be invoked. In practice, $\alpha$ can be determined by $\alpha \leq \beta \gamma$, where $\beta$ denotes the throughput of $id$ insertion, and $\gamma$ denotes the time limit to make new data samples available for selected retrieval. $\gamma$ can be determined by $\gamma = \tau/\sigma$, where $\tau$ denotes the storage capacity of the gateway server, and $\sigma$ denotes the rate of data sample acquisition. Then we have:

$$\alpha \leq \frac{\beta \tau}{\sigma} \quad (1)$$

As a result, the gateway server will be fully utilized if $\beta$ and $\sigma$ are comparable. In addition, if some healthcare applications have specific requirements on $\gamma$, i.e., fast turnaround on new samples, one can add more gateway servers to increase the indexing throughput $\beta$ incrementally. Here, we consider the index uploading time to be marginal, since the connection between the gateway server and the cloud server can be dedicated and high-speed, e.g., AWS Direct Connect [35].

Reducing bandwidth: Recall that processing each index requires sending two set of tokens $\{t_{g_i}, r_{g_i}\}$ to the cloud server. Because the bulk update uses fresh keys for new indexes, each index will be accessed with different tokens for one given request. Accordingly, accessing $C$ encrypted indexes introduces the bandwidth cost of tokens which is equal to $C \{l + \tau \}$. As shown, the cost will scale with $C$ and $l$. To save the bandwidth, we use the same key to generate $\{t_{g_i}\}$ across different indexes, but still use fresh keys to generate $\{r_{g_i}\}$. Therefore, only one copy of $\{t_{g_i}\}$ is sent, and the bandwidth cost is reduced to $l + \tau C l$. As shown later in Section IV-B, nearly half of the bandwidth cost is saved.

From the security perspective, $\{t_{g_i}\}$ for locating buckets in each index are now the same, and thus the cloud server knows that the buckets at the same position of different indexes contain $ids$ that map to the samples with matched LSH hash values. Yet, such leakage does not appear to be harmful, because those buckets are encrypted with different tokens $\{r_{g_i}\}_C l$. Without proper tokens, they will not be decrypted.

D. Accuracy Control

We next conduct probability analysis on the accuracy of our retrieval design. It will serve as the guideline for choosing related system parameters to strike a flexible balance between space efficiency, time efficiency, and retrieval quality. We note that using cuckoo hashing makes the encrypted index very compact, but similar data samples could be kicked apart with a probability such that subsequent retrieval can no longer get them back. To reach better accuracy, we first derive the probability $P$ on the above case that a similar sample is kicked apart, and then propose mechanisms by accessing more buckets or consuming more space to reduce that probability.

Based on the LSH definition in Section II-C, and the bucket selection procedure on the kicked $id$ in CuckooKick, we have the above probability $P$ as follows:

$$P = \frac{1 - p^n_i}{ld(l-1)} \quad (2)$$

Since one of $ld$ buckets is selected for relocation, the probability for the kicked $id'$ is $1/l$. And it will be relocated to one of the rest $l-1$ hash tables. Assume that the kicked $id'$ and the inserted $id$ links to two similar data samples, then the probability of being kicked apart is $1 - p^n_i$, where $p_1$ is the probability for two similar samples with matched LSH hash value, and $m$ is the number of LSH function in each composite LSH function. From Equation 2, one can increase the probing step $d$ to reduce the probability $P$, and this approach also increases query latency and bandwidth.

Alternatively, we insert duplicate $ids$ to improve the accuracy. The probability for kicking apart total number of $c$ copies is derived as:

$$P_c = \left(1 - \frac{p^n_i}{l-1}\right)^c \prod_{i=0}^{c-1} \frac{1}{ld - i} \quad (3)$$

As shown, $P_c$ is much smaller than $P$ for fixed $l$ and $d$. But the tradeoff is that more space consumption is introduced, i.e., the size of the index will be $c$ times larger than the size of the original index. During the implementation, we can further check whether the kicked $id$ is the last copy or not in each CuckooKick operation. If it is the case, this $id$ will be skipped, and another bucket will be selected.

E. Security Analysis

To demonstrate the strong privacy protection on health data, we follow the security definition in the framework of SSE [14], [15], proving that the proposed secure retrieval operation over encrypted indexes is secure against adaptive chosen-keyword attacks (CKA2); that is, the cloud server can learn no partial information about the health data samples and the queries
during secure retrieval and secure update. Encrypted matching samples will be found only with the corresponding and valid query tokens. In particular, we quantify the views of the cloud server, and present a simulation-based security definition. The objective is that even with the views, the cloud server cannot differentiate a real game and a simulated game in a polynomial number of adaptive queries.

For each encrypted index, the cloud server knows the index size, but not the underlying content. The view on each index is defined as \( L_{\text{index}} = (N, |e|) \), where \( N \) is the index capacity and \( |e| \) is the bit length of a bucket. For a given query, the views are defined as \( L_{\text{retrieval}} = \{(t_{gi}, r_{gi}), l, \{e, id\}_id\} \). We note that \( \{t_{gi}\}_i \) on different indexes are the same for bandwidth saving, but \( \{r_{gi}\}_i \) generated from fresh private keys are different for preserving the security strength. During retrieval, the cloud server sees the accessed buckets and the matching ids, i.e., \( \{e, id\}_id \). Given the above definitions of cloud views, the security definition is given as follows:

**Definition 2.** Given functions \( \text{Build}, \text{Retrieval} \) with views \( L_{\text{index}}, L_{\text{retrieval}} \), and a probabilistic polynomial time (PPT) adversary \( A \) and a PPT simulator \( S \), we define the probabilistic games \( \text{Real}_A(\lambda) \) and \( \text{Ideal}_{A,S}(\lambda) \):

\[ \text{Real}_A(\lambda): \text{a challenger C calls KGen}(1^\lambda) \text{ to get a private key } k. \text{ A selects a sample set } s \text{ and asks C to build } I. \text{ Then } A \text{ adaptively conducts a polynomial number of queries with the tokens } t, r \text{ generated from } C. \text{ Finally, } A \text{ returns a bit as the game’s output.} \]

\[ \text{Ideal}_{A,S}(\lambda)(k): A \text{ selects } s, \text{ and } S \text{ generates } \tilde{I} \text{ based on } L_{\text{index}}. \text{ Then } A \text{ adaptively conducts a polynomial number of queries. From } L_{\text{retrieval}} \text{ in each query, } S \text{ generates the corresponding } t, \tilde{r}, \text{ which are processed over } \tilde{I}. \text{ Finally, } A \text{ returns a bit as the game’s output.} \]

Our proposed design is \((L_{\text{index}}, L_{\text{retrieval}})\)-secure against adaptive chosen-keyword attacks (CKA2) if for all PPT adversaries \( A \), there exists a PPT simulator \( S \) such that

\[ \Pr[\text{Real}_A(\lambda) = 1] - \Pr[\text{Ideal}_{A,S}(\lambda) = 1] \leq \epsilon(\lambda) \]

where \( \epsilon(\lambda) \) is a negligible function in \( \lambda \).

**Theorem 1.** Our proposed design is CKA2 secure in the random oracle model if the private-key encryption scheme (KGen, Enc, Dec) is CPA-secure, and \( F, P \) are secure PRF.

**Proof.** From \( L_{\text{index}} \), the simulator \( S \) creates a random index \( \tilde{I} \), which has the same size as the real encrypted index \( I \). It contains \( N \) buckets, where each bucket is an \( |e| \)-bit random string. Due to the semantic security of private-key encryption, the adversary \( A \) cannot differentiate \( I \) from \( \tilde{I} \).

When the first query sample is sent, \( S \) generates random strings as simulated tokens \( \{t_{gi}, r_{gi}\}_i \) with the same length to the real tokens \( \{t_{gi}, r_{gi}\}_i \). After that, a random oracle \( H_1 \) is operated in the way of \( \{H_1(t_{gi}), j\}_j \) for \( j \) from 1 to \( d \) to find related buckets in all \( l \) hash tables respectively. For each accessed bucket, another random oracle \( H_2 \) is operated to obtain the \( id \) inside such that \( id = H_2(r_{gi}, |j|) \oplus e \), where \( ids \) in \( I \) and \( \tilde{I} \) are identical. And from \( L_{\text{retrieval}} \), the matching \( \{id\}_id \) are identical in the two games. In the subsequent queries, if \( t_{gi}, r_{gi} \) appears repeatedly, \( S \) will choose the same tokens simulated before, and obtain the repeated matching \( ids \). Otherwise, \( S \) will generate random tokens and operate random oracles to get the results like the procedure in the first query. Due to the semantic security of PRF, \( A \) cannot differentiate simulated tokens from real tokens. Because the matching \( ids \) in simulated results from real results are identical, \( A \) cannot differentiate them either.

**IV. EXPERIMENTAL EVALUATION**

**A. Experiment Setup**

We implement the proposed design in C++ with O3 optimization, and perform evaluation on an AWS EC2 large instance “r3.8xlarge” with 32 vCPUs, 244GB memory, and Ubuntu Server 15.10 installed. For cryptographic primitives, we use OpenSSL toolkit (version 1.0.1a) to implement the symmetric encryption AES128 and pseudo-random function SHA256. First, we generate a large dataset for performance demonstration of the proposed encrypted index. In particular, we select 1 billion webpages from a public dataset called “Common Crawl Corpus” [36]. For each web page, we generate a 1,000-dimensional Bag-of-Words (BoW) vector according to a dictionary with 1,000 top frequent words. The above high-dimensional vectors are used to simulate high-dimensional data samples in healthcare applications for the purpose of performance evaluation especially on very large scale datasets.

For accuracy evaluation, we select a real healthcare dataset, i.e., the International Skin Imaging Collaboration (ISIC) archive [37] with \( 10^4 \) images. And we use a C++ compressive sensing library called KL1p [38] to derive the samples from raw data, and utilize the MIT E2LSH library [39] to train and obtain the optimized LSH parameters for the selected healthcare dataset. The recovery algorithm CoSaMP [6] implemented in KL1p is applied for visual evaluation.

**B. Performance Evaluation**

**Index building performance:** To demonstrate the superior performance of our proposed encrypted index, we first evaluate the index building time. As shown in Fig. 3-(a), when the number of threads increases, the building time reduces accordingly. The result shows that it takes roughly 2.5 hours to index 1 billion data samples with one single thread. Increasing to 16 threads, it just takes 12 minutes, achieving an average throughput of indexing \( 1.4 \times 10^6 \) encrypted samples per second. Using
a wireless body sensor with compressive sensing support as an example [9], it can generate 40 ECG compressed vectors per second. According to the above throughput, a single gateway server is able to serve up to $3.5 \times 10^3$ such sensors ideally.

The concurrent index building algorithm utilizes multiple threads effectively, since it achieves the optimal number of locks and avoids a large number of locks for the read. Fig. 3-(a) depicts that the building time decreases linearly in the number of threads, which demonstrates that the proposed algorithm maximizes the processing ability of multiple cores. In this experiment, we set the maximum number of threads as 16, i.e., the number of CPU cores. Note that each vCPU of an EC2 instance is a hyperthread of an Intel Xeon core in AWS [40]. Namely, the instance we use with 32 vCPUs actually has 16 physical cores. We remark that as the encrypted index is allocated in the shared memory, adding more threads would possibly lead to performance degradation due to thread synchronization, scheduling, context switch, etc.

For comparison, we implement an encrypted dictionary proposed in [16]. For fairness, we also allocate the encrypted dictionary in the shared memory, and store one-way transformed LSH hash values and encrypted sample ids as key-value pairs. To our best knowledge, this construction is the simplest and one of the most efficient SSE based index in the literature. We use the bucket hash as the underlying dictionary, and the indexing throughput is $1.37 \times 10^5$ samples per second. The result shows that the indexing throughput of our design with 16 threads is $9.2 \times$ higher than the above single thread design. We emphasize that none of existing encrypted index designs can achieve such a high speed of indexing, because they do not explicitly optimize their construction and implementation for fine-grained concurrency support.

To further improve the building performance, we cache the current probing step $d$ of one-way transformed LSH values, as shown in Line 3 of Algorithm 2, to avoid redundant computation during the insertion. Here, we report the building time from 0.25 million samples to 1 million data samples in Fig. 3-(b), the saving with caching varies from 1 times to 3 times. We note that the size of cache is data dependent. If the dataset is dense, the number of distinct LSH values will be small, and so will be the cache size. For this dataset, total 5.7 millions of distinct LSH hash values are derived for 1 million data samples with $l = 20$. We use a 8-bit integer for each probing step, so the cache consumes 5.7MB memory, introducing $31\% = 5.7/18.3$ space overhead in addition to the encrypted index.

In Fig. 4-(a), we explore the insertion cost during the index building procedure. For experimentation, 1000 *ids* of data samples are inserted under different index loads. As seen, the average latency per *id* insertion ranges from 6µs to 28µs, which is extremely fast. The latency greatly increases when the load is over 75%. It is because when more and more buckets are occupied, CuckooKick operations shown in Algorithm 3 will be invoked more frequently, which makes each *id* insertion take longer time. For the same reason, the throughput of insertion decreases when the load of index becomes heavy, as depicted in Fig. 4-(b).

**Memory consumption:** Our encrypted index construction minimizes the security overhead on index memory consumption. Each bucket is encrypted with a AES128 cipher, and a 4-bit tag is attached to realize optimistic locking. As we can see, only 3% = 4/128 space overhead is introduced to have concurrency support for the encrypted index. For 1 billion dataset, the encrypted index consumes 18.3GB with a 90% load factor. For a “r3. 8xlarge” instance with 244GB memory, it can hold indexes for up to 13 billion data samples. According to the achieved indexing throughput and index space efficiency, our design is shown to be capable of maximizing the resource utilization of cloud instances, while preserving the strong protection on medical data samples.

**Secure retrieval performance evaluation:** For the performance evaluation of secure retrieval, we report the query latency and throughput. Recall that the secure retrieval algorithm asks the cloud server to locate and process $d$ buckets in each of hash tables. Thus, the cost of each retrieval is bounded, i.e., $ld$ buckets are processed. In Fig. 5-(a), the query latency increases linearly in the number of processed buckets. For $l = 20$ and $d = 25$, it takes less than 2.5s to obtain 500 result *ids*. The proposed retrieval algorithm also supports concurrent processing over the encrypted index. Those hash tables can be accessed in parallel. In Fig. 5-(b), the more threads one uses, the more query throughput one can have. But we note that when the number of probing step increases, the throughput will decrease because each secure retrieval operation takes more time on bucket processing. For 16 threads and $d = 5$, the query throughput reaches over $2.5 \times 10^5$ queries per second.

We next evaluate the bandwidth cost for tokens generated in each query sample. Recall that processing each index requires the data user to send two sets of tokens $\{t_{gi}, r_{gi}\}$ to the cloud.
server. A new index from bulk update is generated with fresh keys. For \( C \) encrypted indexes, the bandwidth cost is equal to \( CI(|t_{g_1}| + |r_{g_1}|) \). Then we use the same key to generate \( \{t_{g_1}\} \) across different indexes to save the cost, but still use fresh keys to generate \( \{r_{g_1}\} \) for security. As a result, the bandwidth cost is equal to \( |t_{g_1}| + CI|r_{g_1}| \). Here, SHA256 is used to implement PRFs, and thus the size of \( |t_{g_1}| \) and \( |r_{g_1}| \) is equal to 32 bytes. When all indexes are accessed by different tokens, 12.8KB bandwidth is introduced in a given query for \( l = 20 \) and \( C = 10 \). If they are accessed by the same \( \{t_{g_1}\} \), 7.04KB bandwidth is introduced, i.e., 45\% saving. We can see that the more indexes there are, the more saving there will be.

C. Accuracy Evaluation

To understand the retrieval quality of high-value results, we conduct accuracy and visual evaluation on a real healthcare image set ISIC [37]. Our work focuses on the retrieval of compressed image samples, rather than the reconstruction of raw images. Thus, we primarily show the quantitative results for the retrieval. In particular, we use the metric for Top-K precision in [41], defined as \( \frac{1}{K} \sum_{i=1}^{K} |s_i - r_i| \), where \( s_i \) is the query sample and \( s'_i \) and \( s_i \) are the \( i \)-th closest samples from the ground truth and the retrieval results from our encrypted index respectively.

As illustrated in Fig. 6, the precision of Top-10 reaches over 95\%. Recall that in Section III-D, we propose two methods that improve accuracy. First, one can increase the probing step \( d \) to reduce the probability that a sample \( id \) is kicked away as shown in Eq. 2. In Fig. 6-(a), for a \( l = 20 \), the precision for \( d = 15 \) is lower than the precision for \( d = 30 \). And for Top-50 results, the precision has 4\% loss. We note that as total \( id \) buckets are accessed in each query, the query latency will increase as \( d \) increases. The second method is to store multiple copies of each \( id \) for better accuracy. As indicated in Eq. 3, the probability that all copies for a given \( id \) are kicked apart is lower if there are more copies in the index. In Fig. 6-(b), for \( l = 10 \), \( d = 20 \), the precision of storing 2 \( id \) copies is higher than the precision of storing only 1 copy per \( id \). For Top-50 results, the precision of storing 1 copy per \( id \) has 3\% loss compared to the precision of storing 2 copies per \( id \). As a tradeoff, storing 2 copies introduces twice as much space consumption.

The above results indicate that there is a tradeoff between accuracy and efficiency in our design. Proper parameters should be selected based on the requirements of certain medical applications. If an application demands better accuracy, a larger \( d \) can be chosen, or multiple \( id \) copies can be indexed. If an application tolerates a certain level of accuracy loss while demanding time and space efficiency, one may choose a smaller \( d \) for lower query latency, and store only 1 copy per \( id \) for optimal space consumption.

In Fig. 7, visual evaluation is depicted for two query medical images. In this experiment, we adopt a conservative compression ratio 25\% for demonstration. As illustrated, the images reconstructed from retrieved samples are closely related to the query image. Recall that the correctness of our design is based on two facts. First, distances between raw data are still preserved in the measurement space of compressed samples. As the compressed samples are high-dimensional data points, applying LSH enables fast and effective similarity retrieval over those compressed samples. Second, the one-to-one mapping property of secure PRFs ensures that tokens generated from same composite LSH hash values are deterministic, and thus similar samples can still be grouped together and later be retrieved by corresponding secure tokens. Our security design will not affect the effectiveness of LSH. On the other hand, one may also use an aggressive compression ratio for more communication and storage saving, but the recovery quality could be reduced. We note that the methodologies related to data reconstruction are independent to our work.

V. RELATED WORK

To utilize encrypted medical data samples, one prerequisite is to securely index those encrypted data samples while allowing the cloud server to match similar data samples without knowing the queries and the underlying content of samples. Thus, our proposed designs are closely related to a line of work on searchable symmetric encryption (SSE). In [21], Kuzu et al. first combine LSH and SSE to enable secure similarity search over encrypted data. Prior SSE schemes [14]–[16], [42] (to just list a few) can also be adopted to serve for the same purpose as
long as the LSH hash values are treated as distinct keywords. The above schemes do not consider to address the imbalance of LSH, and using those schemes will result in encrypted indexes with low memory efficiency. More importantly, those schemes fall short of reaching high indexing throughput. They focus on application scenarios such that the building procedure of the encrypted index is a one-time setup. Once the index is built, new data will rarely be added. Thus, they can hardly fulfill the performance requirement in our context.

To achieve high indexing throughput, space efficiency, and guaranteed security for sensitive medical data simultaneously, we start from our prior LSH based encrypted index constructions [22], [23] with optimal space consumption. Yet, we realize that utilizing multiple threads to build such a memory efficient encrypted index is a non-trivial task. Then we explore efficient concurrent programming techniques, and devise an algorithm that systematically integrates optimistic locking, LSH, SSE, and high performance hashing techniques all together. Moreover, we apply the above core building block to our proposed secure cloud-based framework for healthcare monitoring systems, and integrate it with compressive sensing for reduced storage cost, energy efficient data acquisition, and high-value retrieval with privacy preservation.

Another loosely related work focuses on privacy-preserving image recovery services [18], [20], [27]. The above designs enable the cloud server to securely recovery images from compressed samples without learning information from either the data samples or the underlying images. The recovery problem is formulated as $\ell_1$ minimization problem. And a secure transformation mechanism is proposed to map the original problem to a random one so as to hide the information of the underlying image. We note that the above studies are orthogonal to our design, but can readily be integrated into our secure retrieval service to develop a full-fledged system. The encrypted high-value data samples can be located and then recovered in a privacy-preserving fashion.

VI. Conclusion

In this work, we propose a privacy-aware healthcare system that supports secure and fast indexing of encrypted medical data. Unlike existing approaches, we start from the novel data acquisition/sample technique, i.e., compressive sensing, in a way that the content-based retrieval can be allowed on compressed data. Because medical data is naturally sensitive, we then apply encrypted search techniques to achieve secure retrieval over encrypted data samples. To address the challenges for continuously generated medical data samples at high rates and large volumes, we propose an encrypted high-performance index which can be fast built via concurrent insertion threads. For practical considerations, we further improve the building performance via caching, reduce the bandwidth of secure retrieval, and explore the relationship between accuracy and efficiency. Experimental evaluation on Amazon Cloud demonstrates practical performance of the proposed index designs and good quality of retrieval.

ACKNOWLEDGMENT

This work was supported in part by Research Grants Council of Hong Kong (Projects No. CityU 138513 and No. CityU 11276816), the Natural Science Foundation of China under Project 61572412, Innovation and Technology Commission of Hong Kong under ITF Project ITS/307/15, and the AWS Education Research Grant. Portions of Dr. Kui Ren’s research was supported by US National Science Foundation under grant CNS-1262777. Portions of Dr. Jian Weng’s work was supported by National Science Foundation of China under Project 61272413, 61133014, 61272415 and 61472165, Research Fund for the Doctoral Program of Higher Education of China under grant 20134401110011, and the 2016 special fund for Applied Science & Technology Development and Transformation of Major Scientific and Technological Achievements.

REFERENCES


Xinyu Wang is currently a Ph.D. student at at the Department of Computer Science, City University of Hong Kong. He received his B.E. degree in Software Engineering from East China Jiaotong University in 2011. He worked for Tencent as a software engineer from 2011 to 2013. His research interests include cloud computing, network security, and big data.

Cong Wang (M’11) is an assistant professor in the Computer Science Department at City University of Hong Kong. He received his B.E. and M.E. degrees from Wuhan University in 2004 and 2007, and the Ph.D. degree from Illinois Institute of Technology in 2012, all in Electrical and Computer engineering. He worked at the Palo Alto Research Center in the summer of 2011. His research interests are in the areas of cloud computing security.

Jian Weng (M’15) received the M.S. and B.S. degrees in computer science and engineering from the South China University of Technology, Guangzhou, China, in 2004 and 2000, respectively, and the Ph.D. degree in computer science and engineering from Shanghai Jiao Tong University, Shanghai, China, in 2008. From April 2008 to March 2010, he was a Postdoc with the School of Information Systems, Singapore Management University, Singapore. Currently, he is a Professor and Executive Dean with the School of Information Technology, Jinan University, Guangzhou, China. He has authored or coauthored more than 60 papers in cryptography conferences and journals.

Prof. Weng has served as PC Co-Chair or PC Member for more than 20 international conferences. He was the recipient of the 2014 Cryptographic Innovation Award from the Chinese Association for Cryptographic Research, the Best Paper Award from SCIS 2011, and the Best Student Award from ProvSec 2014.

Kui Ren (F’16) received the Ph.D. degree from the Worcester Polytechnic Institute, Worcester, MA, USA. He is currently a Professor of Computer Science and Engineering and the Director of the Ubisec Laboratory with the University at Buffalo, Buffalo, NY, USA. His research has been supported by the NSF, DoE, AFRL, MSR, and Amazon. He has authored or coauthored 160 peer-reviewed journal and conference papers. His current research interests include cloud and outsourcing security, wireless and wearable system security, and human-centered computing.

Prof. Ren is a Member of the Association for Computing Machinery, and a Distinguished Lecturer of the IEEE Vehicular Technology Society. He was a recipient of the NSF CAREER Award in 2011 and the Sigma Xi/IIT Research Excellence Award in 2012. He was the recipient of several Best Paper Awards, including the IEEE ICNP 2011. He serves as associate editor for IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Mobile Computing, IEEE Transactions on Information Forensics and Security, IEEE Wireless Communications Magazine, IEEE Internet of Things Journal, IEEE Transactions on Smart Grid, Elseviers Pervasive and Mobile Computing, and Oxfords The Computer Journal.

Xingliang Yuan is currently a Ph.D. student at the Department of Computer Science, City University of Hong Kong. He received his B.S. degree from Nanjing University of Posts and Telecommunications in 2008, and the M.S. degree from Illinois Institute of Technology in 2009, both in Electrical Engineering. His research interests include cloud computing security, mobile security, and multimedia security.