IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 28, 2020, accepted May 3, 2020, date of publication May 14, 2020, date of current version June 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2994598

uFETCH: A Unified Searchable Encryption Scheme and Its Saas-Native to Make DBMS Privacy-Preserving

SHEN-MING CHUNG[1], MING-DER SHIEH[1], (Member, IEEE), TZI-CKER CHIUEH[2], CHIA-CHIA LIU[3], AND CHIA-HENG TU[3]

[1]Department of Electrical Engineering, National Cheng Kung University, Tainan 70101, Taiwan
[2]Information and Communications Research Laboratories, Industrial Technology Research Institute, Hsinchu 31040, Taiwan
[3]Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 70101, Taiwan

Corresponding author: Shen-Ming Chung (anton0706@gmail.com)

ABSTRACT As encrypted-search techniques such as Searchable Encryption (SE) were devised for homogeneous data type, i.e. textual or numerical, it is a nature presumption that multiple techniques have to be intertwined to make database management system (DBMS) privacy-preserving. In effect, such a presumption has led to popular designs such as CryptDB, putting efforts on heterogeneous integration. In this paper, an easier option is made available when a unified SE scheme named uFETCH is proposed to accommodate both. Namely, uFETCH is able to build unified SE indexes for both the types while enabling encrypted search even if the SE indexes built for texts and numbers are mingled. To demonstrate how uFETCH can bring up simpler designs, a security agent is exemplified to work with off-the-shelf DBMS while making it privacy-preserving. Since uFETCH transforms the problem of encrypted search into a simple problem of subsequence matching for SaaS-native, it requires only sub-linear search time w.r.t. the volume of indexed items and is secure in the widely-adopted 3-tier cloud structure to help cloud service providers ease regulation compliance with out-sourced repository.

INDEX TERMS Searchable encryption, DBMS, privacy preserving, SaaS-native.

I. INTRODUCTION

As General Data Protection Regulation (GDPR) [1] and similar laws are acting in more countries, people are much aware of their rights on privacy. Because of these laws, companies doing business with private data have no options but to protect them with full strength. With raised awareness, it would make no sense to private data owners if the companies, working as a cloud service provider (CSP) in the era of cloud, further moves private data to yet another CSP for repository. Despite such a practice so common in *3-tier cloud structure* [2] that enables agility and more, as long as GDPR is considered, out-sourcing repository does make CSP harder to claim compliance since the out-sourced repository is at best *semi-trusted* [3]. Therefore, a challenge arises: how can CSP preserve the privacy of its users while being able to out-source repository?

The associate editor coordinating the review of this manuscript and approving it for publication was Gianmaria Silvello.

Intuitively, encryption such as AES seems a solution for CSP to address the challenge. For example, CSP can encrypt private data before storing them upon cloud repository. If CSP only keeps the key(s), virtually neither cloud repository nor CSP can see the private data except the epochs of data processing. This way of managing data not only retains the 3-tier cloud structure but also makes CSP free from escorting private data all the time. Technique as such also eases implementing "the right to be forgotten" in GDPR because simply deleting a key destroys all associated data at once.

However, as CSP has to process data as fast as possible, mere encryption would make CSP a nightmare. That is, all encrypted data have to be retrieved back from cloud repository before decrypting for the "wanted". Though, with traditional encryption, only retrieving all can guarantee a perfect *recall* necessary for business, it does suffer the worst *precision* by metrics of Information Retrieval (IR) [4].

To get decent precision, solutions for commissioned search among encrypted items have been proposed. Among them,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Searchable Encryption (SE) and other encrypted-search techniques stand out to enable efficient retrieval. However, if CSP takes a deeper look at these solutions, obstacles can be found that explain why these ''near-practical'' solutions are still rarely deployed.

A. EXISTING TECHNIQUES AND RESTRICTIONS

Foreseeing privacy issues at the dawn of cloud era, SE has long been proposed to encrypt data flowing to cloud while preserving IR ability for those who have the key. However, the name of SE is often misleading that seems to bring forth special encryption whose ciphertext is searchable. In fact, except the proposal of Song *et al.* [5], almost all SE schemes actually build subsidiary *SE index(es)* for word(s) that represents the data. With SE index(es) associated with encrypted data, retrieving encrypted data becomes a matching problem. That is, with the right key, *SE Trapdoor* can be built for a word to match through SE index(es). Since this index-based approach [6], [7] emerged, a variety of SE schemes have been proposed for single-keyword search, ranging from offering simple *equality* queries [5]–[7] to annexing queries with better expressiveness such as *fuzzy* [8], [9] and *wildcard* [10]–[15]. For multiple-keyword search, *conjunctive* schemes [10], [16]–[18] have been proposed; in case that keywords have to be consecutive, *phrase* [19]–[22] and even *multi-phrase* [23] can be useful. But one thing in common with them is: when talking about SE, schemes are dedicated to matching for *word(s)*.

Besides words, privacy issue also attracts lots of attention on privacy-preserving search for encrypted *numbers*. This line of research is often seen in database community. Such works fill the vacancy in the paradigm of database management system (DBMS) where numbers usually dominate. But unlike SE, this community are more liberal to engage skills beyond cryptography. For example, in 2002 Hacigümüş *et al.* [24] proposed a *bucketing* technique with SQL translations to map range queries onto buckets recalls. Since then, a number of varying proposals appeared with many aiming at *database-as-a-service (DaaS)*. While many works proposed to modify DBMS in order to inject techniques such as *bivariate* [25], *PBtree* [26], *trusted hardware* [27] and *multi-party computation* [28], some preferring not to. Those techniques that requires no server modification, e.g. [24], [29] and [30], are regarded as SaaS-native. But SaaS-native or not, all these techniques are dedicated to looking for *number(s)*.

Of course, neither words nor numbers alone are sufficient to do business as CSP inevitably has to face databases mixed with textual and numerical data. But, with the development of encrypted-search techniques as summarized, it is a nature presumption that heterogeneous techniques across data types have to be intertwined to deal with general databases. Such a presumption is somewhat affirmed after CryptDB [31] integrated techniques of *order-preserving encryption (OPE)* [30], *homomorphic encryption (HE)* [32] and SE [5] to make DBMS privacy-preserving. Remarkably, by picking only SaaS-native techniques, CryptDB is gaining popularity as it fits well with DaaS that forbids any DBMS retrofit. In fact, such an integration even inspired works such as MONOMI [33], L-EncDB [22] and Seabed [34]; all putting efforts on intertwining heterogeneous techniques that we propose to refrain from.

B. CONTRIBUTIONS

This work proposes a solution for CSP to address the challenge of preserving user privacy with out-sourced repository. We introduce a *unified* SE scheme called uFETCH (unified Frequency-Eliminated Trapdoor-Character Hopping) that enables efficient encrypted-search across data types. With the merit of SaaS-native to make off-the-shelf DBMS privacy-preserving, uFETCH brings the following contributions to advocate cloud security and privacy.

- **First Unified SE Scheme**. uFETCH is able to make encrypted data searchable no matter they are textual, numerical or mixed. By a unified search algorithm, it offers efficient encrypted-search even if the SE indexes built for texts are mingled with the SE indexes built for numbers. uFETCH builds SE trapdoors devised to tell nothing about the type of data it is looking for. It features wildcard-based pattern search for encrypted texts while also making range-based search possible for encrypted numbers. To the best of our knowledge, uFETCH is the first SE scheme of this kind.
- **SaaS-Native for Legacy DBMS**. Cloud repository running as DaaS often provides DBMS not open for retrofit. This fact cripples all encrypted-search techniques unless they are SaaS-native, i.e. able to work with DBMS as is. Therefore, we believe a practical SE scheme should be at least SaaS-native. uFETCH is SaaS-native. It builds SE indexes and trapdoors that require no retrofit to off-the-shelf DBMS as long as it serves *subsequence matching*. More than SaaS-native, uFETCH also brings forth a kind of simplicity that heterogeneous integration cannot afford. This simplicity is demonstrated by a security agent that uses only uFETCH to make DBMS privacy-preserving with just few SQL translations.
- **Secure in 3-tier Cloud Structure**. Brought up in cryptography communities, SE schemes are often set upon security notions such as IND-CKA [6], [7]. However, we notice that such notions are actually meant for 2-tier client-server structure, instead of the 3-tier structure recurring in cloud scenarios. In effect, such ''over-spec'' security often leads to inefficient SE schemes whenever better query expressiveness is needed. For example, in Table 1, SE schemes supporting wildcard-based pattern search are often bounded by linear complexity unfit for increasing data in cloud. uFETCH can break that bound with adequate security in 3-tier cloud structure for the sake of commercial-grade performance.

In the following, Section II first reviews some SE primitives devised for SaaS-native, setting a background for the

TABLE 1. Existing SE schemes supporting wildcard-based pattern search are bounded by linear complexity, but our works break that bound by leveraging existing DBMS capability in 3-tier cloud structure (n and m are the number of indexed documents and words respectively).

SE scheme	setting	number indexable	SaaS native	search complexity	best reported experimental results: textual search time	precision/recall	trapdoor/index size	dataset
[12]	public-key	No	No	$\mathcal{O}(n)$	3236ms / 6 words[(1)]	1 / 1	N/A	N/A
[15]	public-key	No	No	$\mathcal{O}(n)$	2294ms / 6 words	1 / 1	25.82KB[(2)] / N/A	N/A
[10]	symmetric	No	No	$\mathcal{O}(n)$	< 1ms / 6 words [(1)]	N/A / 1	13.5B / 19.12B	N/A
[11]	symmetric	No	No	$\mathcal{O}(m)$	< 1ms / 6 words[(1)]	N/A / 1	N/A	N/A
[14]	symmetric	No	No	$\mathcal{O}(m)$	< 1ms / 6 words[(1)]	N/A / 1	N/A	N/A
[13]	symmetric	No	No	$\mathcal{O}(m)$	800ms / 12k words	0.98- / 0.9+	1.25KB / 1.25KB	ACM[(3)]
FETCH [35]	symmetric	No[(4)]	Yes	$\mathcal{O}(\log m)$	< 200ms / 334k words	0.998+ / 1	20B / 200B	Enron

[(1)] reported in [15] [(2)] communication cost of a query [(3)] ACM Digital Library [(4)] Yes, when upgrading to uFETCH (this work)

proposed scheme. The proposed uFETCH and its security discussions for 3-tier cloud structure are given in Section III. The simple security agent enabled by uFETCH is demonstrated in Section IV, followed by experimental results in Section V. Finally, a conclusion is given in Section VI.

II. SE PRIMITIVES FOR SaaS-NATIVE

In this section, SE primitives [35] devised for *textual-only* encrypted search are reviewed as a background. Note that, the SE primitives along are not secure but serves to set up operations and notations to be adopted in the proposed SE scheme. The operations and notations illustrate how the problem of encrypted search (using wildcards in particular) can be transformed into a problem of subsequence matching for SaaS-native.

A. INDEX CONSTRUCTION

Inspired by distorting mirrors in physical world, the SE primitives reflect and project an object (a given word) onto an obfuscating image (an SE index) mathematically; and in case that you cover (wildcardize) a part of the object (the word), the part that is not covered still gets reflected and projected onto another image (an SE trapdoor) which, though still obfuscating, will be in part the same as the uncovered image (the SE index) and thus can be used as a clue for matching. But unlike distorting mirrors, "garbling" is introduced, resulting in three steps of *reflecting*, *projecting* and *garbling* for building an SE index, as shown in Fig. 1.

More precisely, given a word $\boldsymbol{p} = \left[p_1\ p_2\ \ldots\ p_N\right]^\top$, in the step of reflecting, characters of $\boldsymbol{p}$ are "reflected" by multiplying with a matrix $\boldsymbol{R}$ of M-by-N entries defined as

$$\boldsymbol{R} \triangleq \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \boldsymbol{r}_3 \\ \vdots \\ \boldsymbol{r}_M \end{bmatrix} \tag{1}$$

with each $\boldsymbol{r}_{m\in\{1,2,\ldots,M\}}$ defined as a shuffled version of an N-tuple vector $\boldsymbol{z} \triangleq \left[x^0\ x^1\ \ldots\ x^{\alpha-1}\ 0\ 0\ \ldots\ 0\right]$, wherein each non-zero entry is a monomial called a *reflecting pointer* and α defines the number of them in each row of $\boldsymbol{R}$. We denote the

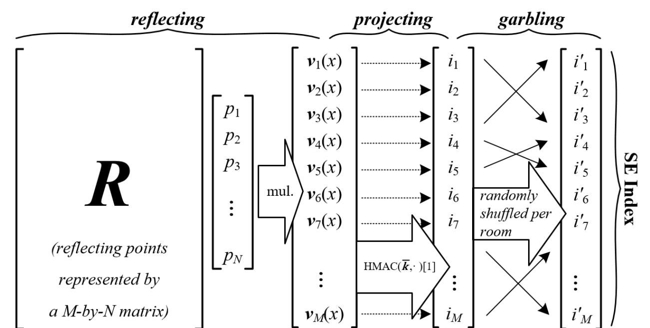

FIGURE 1. Steps to build a SaaS-native SE index from an N-character word.

generation of such a matrix as $\boldsymbol{R} \xleftarrow{\$} \mathbb{R}^{(\alpha)}_{M\times N}$ with $\xleftarrow{\$}$ denoting a random sampling.

Such a reflecting step "reflects" M subsets of characters of $\boldsymbol{p}$ on the coefficients of polynomials $\boldsymbol{v}_1(x), \boldsymbol{v}_2(x), \ldots$ and $\boldsymbol{v}_M(x)$. That is,

$$\boldsymbol{Rp} = \boldsymbol{R}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{v}_1(x) \\ \boldsymbol{v}_2(x) \\ \boldsymbol{v}_3(x) \\ \vdots \\ \boldsymbol{v}_M(x) \end{bmatrix} \tag{2}$$

In the step of projecting, each of the polynomials is projected onto a character. A keyed hashing such as HMAC can be conducted over the coefficients of each polynomial with a secret key $\bar{\boldsymbol{k}} \xleftarrow{\$} \{0, 1\}^n$, and the first character of the output hash is selected as the projected character. Denoting such a projecting as $\succ$, we obtain an image

$$\boldsymbol{i} \triangleq \begin{bmatrix} \boldsymbol{v}_1(x)^\succ \\ \boldsymbol{v}_2(x)^\succ \\ \boldsymbol{v}_3(x)^\succ \\ \vdots \\ \boldsymbol{v}_M(x)^\succ \end{bmatrix} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ \vdots \\ i_M \end{bmatrix} \tag{3}$$

Finally, for the step of garbling, the image $\boldsymbol{i}$ is partitioned into λ "rooms" specified by a sequence $\boldsymbol{b}$ of increasing integers, i.e.

$$\boldsymbol{b} \triangleq (b_1, b_2, \ldots, b_{\lambda+1})|_{b_1=1, b_{\lambda+1}=M+1, b_i-b_{i-1}>2} \tag{4}$$

Assuming $\boldsymbol{b}$ is randomly-generated according to (4) that we denote as $\boldsymbol{b} \overset{\$}{\leftarrow} \mathbb{B}(M, \lambda)$, we can partition $\boldsymbol{i}$ as

$$\boldsymbol{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ \vdots \\ i_M \end{bmatrix} = \begin{bmatrix} \boldsymbol{i}_1 \\ \boldsymbol{i}_2 \\ \vdots \\ \boldsymbol{i}_\lambda \end{bmatrix} \tag{5}$$

such that

$$\boldsymbol{i}_l = \begin{bmatrix} i_{b_l} \\ i_{b_l+1} \\ i_{b_l+2} \\ \vdots \\ i_{b_{(l+1)}-1} \end{bmatrix} \tag{6}$$

Aligned with the partitioned $\boldsymbol{i}$, a *Garbler* $\boldsymbol{G}$ is defined as

$$\boldsymbol{G} \triangleq \begin{bmatrix} \boldsymbol{\pi}_1 \\ \boldsymbol{\pi}_2 \\ \boldsymbol{\pi}_3 \\ \vdots \\ \boldsymbol{\pi}_\lambda \end{bmatrix} \tag{7}$$

where each $\boldsymbol{\pi}_{l \in \{1,2,\ldots,\lambda\}}$ is a randomly-generated *permutation matrix* [36] of dimension $(b_{l+1} - b_l) \times (b_{l+1} - b_l)$.

With $\boldsymbol{G}$, an SE index is obtained by

$$\boldsymbol{idx} \triangleq \boldsymbol{G}^\top \boldsymbol{i} \triangleq \begin{bmatrix} \boldsymbol{\pi}_1 \boldsymbol{i}_l \\ \boldsymbol{\pi}_2 \boldsymbol{i}_2 \\ \boldsymbol{\pi}_3 \boldsymbol{i}_3 \\ \vdots \\ \boldsymbol{\pi}_\lambda \boldsymbol{i}_\lambda \end{bmatrix} \tag{8}$$

Note that, the resulted $\boldsymbol{idx}$ is SaaS-native because it can be stored as a *string*, i.e. the most common type in DBMS.

B. TRAPDOOR CONSTRUCTION

Before showing how SE trapdoors can be built for wildcard queries, we first define what a wildcard query is:

- **Wildcard Query**. $\tilde{\boldsymbol{q}} = \begin{bmatrix} \tilde{q}_1 \ \tilde{q}_2 \ \ldots \ \tilde{q}_N \end{bmatrix}^\top$ is a wildcard query if it is made by replacing zero or more characters of a certain word $\boldsymbol{p} = \begin{bmatrix} p_1 \ p_2 \ \ldots \ p_N \end{bmatrix}^\top$ with wildcard Δ, that represents any single character. And we denote $\tilde{\boldsymbol{q}} \trianglerighteq \boldsymbol{p}$ if $\tilde{\boldsymbol{q}}$ matches $\boldsymbol{p}$. More precisely,

$$\tilde{\boldsymbol{q}} \trianglerighteq \boldsymbol{p} \Leftrightarrow (\tilde{q}_i \neq \Delta \Rightarrow \tilde{q}_i = p_i) \forall i \in \{1, 2, \ldots, N\} \tag{9}$$

By this definition and the notation, wildcard query is seen as a generalization of equality query. Given a wildcard query $\tilde{\boldsymbol{q}} = \begin{bmatrix} \tilde{q}_1 \ \tilde{q}_2 \ \ldots \ \tilde{q}_N \end{bmatrix}^\top$, to conduct encrypted-search through SE indexes, an SE trapdoor can be built with the same $\{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}\}$. As illustrated in Fig. 2, the procedure of building SE trapdoor for the wildcard query also comprises three steps: reflecting, projecting and *sifting*. In reflecting,

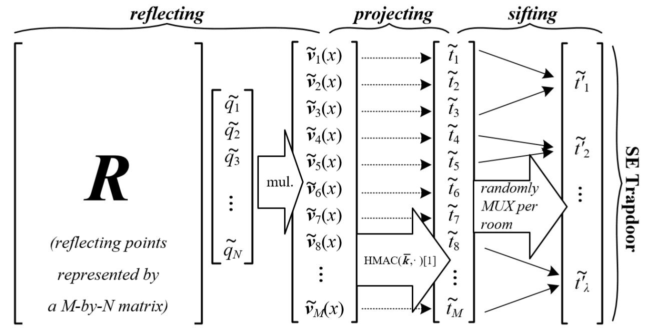

FIGURE 2. Steps to build a SaaS-native SE trapdoor from a N-character (wildcardized) query word.

M polynomials are resulted, i.e.

$$\boldsymbol{R}\tilde{\boldsymbol{q}} = \boldsymbol{R} \begin{bmatrix} \tilde{q}_1 \\ \tilde{q}_2 \\ \tilde{q}_3 \\ \vdots \\ \tilde{q}_N \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{v}}_1(x) \\ \tilde{\boldsymbol{v}}_2(x) \\ \tilde{\boldsymbol{v}}_3(x) \\ \vdots \\ \tilde{\boldsymbol{v}}_M(x) \end{bmatrix} \tag{10}$$

If there is any Δ in $\tilde{\boldsymbol{q}}$, it would get reflected among the coefficients of $\tilde{\boldsymbol{v}}_1(x), \tilde{\boldsymbol{v}}_2(x), \ldots, \tilde{\boldsymbol{v}}_M(x)$. For each polynomial containing at least one Δ, we force the result of projecting $\succ$ to be a Δ. In general, any operation dealing with Δ is forced to output Δ, resulting in "wildcard propagation". Therefore, one might obtain a Δ-involved $\boldsymbol{t} = \begin{bmatrix} \tilde{t}_1 \ \tilde{t}_2 \ \ldots \ \tilde{t}_M \end{bmatrix}^\top$ that, like (5), is partitioned into λ rooms according to $\boldsymbol{b}$, i.e.

$$\boldsymbol{t} \triangleq \begin{bmatrix} \tilde{\boldsymbol{v}}_1(x)^\succ \\ \tilde{\boldsymbol{v}}_2(x)^\succ \\ \tilde{\boldsymbol{v}}_3(x)^\succ \\ \vdots \\ \tilde{\boldsymbol{v}}_M(x)^\succ \end{bmatrix} = \begin{bmatrix} \tilde{t}_1 \\ \tilde{t}_2 \\ \tilde{t}_3 \\ \vdots \\ \tilde{t}_M \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{t}}_1 \\ \tilde{\boldsymbol{t}}_2 \\ \vdots \\ \tilde{\boldsymbol{t}}_\lambda \end{bmatrix} \tag{11}$$

Aligned with the partitioned $\boldsymbol{t}$, a *Sift* $\boldsymbol{S}$ is defined as

$$\boldsymbol{S} \triangleq \begin{bmatrix} \boldsymbol{\sigma}_1 \\ \boldsymbol{\sigma}_2 \\ \boldsymbol{\sigma}_3 \\ \vdots \\ \boldsymbol{\sigma}_\lambda \end{bmatrix} \tag{12}$$

wherein each $\boldsymbol{\sigma}_{l \in \{1,2,\ldots,\lambda\}}$ is a randomly-generated *selection vector* [36] of dimension b_l that has exactly one entry of 1 and 0's elsewhere. By $\boldsymbol{\sigma}_{l \in \{1,2,\ldots,\lambda\}}$, one character per the room $\tilde{\boldsymbol{t}}_{l \in \{1,2,\ldots,\lambda\}}$ is selected (i.e. mux). Thereby, an *SE trapdoor* is built as

$$\boldsymbol{trap} \triangleq \boldsymbol{S} \cdot \boldsymbol{t} \triangleq \begin{bmatrix} \boldsymbol{\sigma}_1 \tilde{\boldsymbol{t}}_l \\ \boldsymbol{\sigma}_2 \tilde{\boldsymbol{t}}_2 \\ \boldsymbol{\sigma}_3 \tilde{\boldsymbol{t}}_3 \\ \vdots \\ \boldsymbol{\sigma}_\lambda \tilde{\boldsymbol{t}}_\lambda \end{bmatrix} \tag{13}$$

Denoting $\boldsymbol{trap} \sqsupseteq \boldsymbol{idx}$ if $\boldsymbol{trap}$ is a subsequence of $\boldsymbol{idx}$, it can be observed that

$$\tilde{\boldsymbol{q}} \trianglerighteq \boldsymbol{p} \Rightarrow \boldsymbol{trap} \sqsupseteq \boldsymbol{idx} \tag{14}$$

as proved in [37]. With this property, cloud repository can conduct a textual search simply by checking whether a given $\boldsymbol{trap}$ is a subsequence to any of stored $\boldsymbol{idx}$'s. That is, with no $\tilde{\boldsymbol{q}}$ nor $\boldsymbol{p}$ ever exposed to, cloud repository can be commissioned to search while remaining ignorant. The resulted trapdoor $\boldsymbol{trap}$ is SaaS-native because it only requires subsequence matching, a well-supported feature in most DBMS.

III. PROPOSED UNIFIED SE SCHEME - uFETCH

The property of (14) indicates the reviewed SE primitives as SaaS-native. However, the SE primitives can only serve words of fixed-length of N, that is too restricted for most scenarios. Besides, when feeding a word of same characters, the index construction will lead to an $\boldsymbol{idx}$ of same character. For example, a word "`000...0`" would lead to an $\boldsymbol{idx}$ like "`xxx...x`" that leaks the word pattern.

In this section, the SE primitives are extended and formalized into the proposed uFETCH of four algorithms $\{KeyGen, BuildIndex, Trapdoor, Match\}$. uFETCH frees the word-length restriction and leaks no word pattern. Being a unified SE scheme, uFETCH also builds indexes for numbers with range queries enabled.

Our initial idea is to "normalize" all words to N-character long, and if N is configured large enough, the scheme can then cover most words. Interestingly, it happens that such normalization can even apply to numbers as long as one can find ways that not only transform any number $\boldsymbol{d} \in [min, max]$ into a word $\boldsymbol{p} = \left[p_1\ p_2\ \ldots\ p_N\right]^\top$, but also any range $\boldsymbol{r} \subset [min, max]$ into $\tilde{\boldsymbol{q}} = \left[\tilde{q}_1\ \tilde{q}_2\ \ldots\ \tilde{q}_N\right]^\top$ such that if $\boldsymbol{d} \in \boldsymbol{r}$ then $\tilde{\boldsymbol{q}} \trianglerighteq \boldsymbol{p}$. Then, leveraging (14), a perfect recall for numerical encrypted search is assured. We found one feasible way to achieve it is by N projections that project $\boldsymbol{r}$ and $\boldsymbol{d}$ onto $\tilde{\boldsymbol{q}}$ and $\boldsymbol{p}$ respectively.

A. uFETCH SCHEME KEY

Algorithm 1 ***KeyGen***

Input : a unary 1^n, *min* and *max*
Output: a uFETCH scheme key $\boldsymbol{k}$

1. $\boldsymbol{R} \xleftarrow{\$} \mathbb{R}^{(\alpha)}_{M\times N}$
2. $\bar{\boldsymbol{k}} \xleftarrow{\$} \{0,1\}^n$
3. $\boldsymbol{b} \xleftarrow{\$} \mathbb{B}(M, \lambda)$
4. $\boldsymbol{n} \leftarrow \{n_1, n_2, \ldots, n_N | min < n_i < max\}$
5. $\boldsymbol{k} \leftarrow \{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}, \boldsymbol{n}\}$

An instance of uFETCH is defined by a scheme key $\boldsymbol{k} \leftarrow \{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}, \boldsymbol{n}\}$ with public configuration of N, M, α, λ, L and (min, max). Besides the ingredients of the reviewed SE primitives, $\boldsymbol{n}$ is introduced as a vector containing N secret numbers $n_1, n_2, \ldots, n_N$ within (min, max) that configure N projections in algorithms *BuildIndex* and *Trapdoor* and are used exclusively for numbers.

Note that, these secret numbers may or may not be randomly generated, as they can be specified per a known distribution to conceal it, as detailed in Section III-E4.

B. uFETCH INDEX

The algorithm *BuildIndex* embodies our idea to "normalize" a datum. That is, when a textual word is given, steps are taken to make it a word of N-character long. But if it is a number, N projections are instead conducted to make it a N-character word too. Thereby, regardless of the data type, the same procedure of projecting, reflecting and garbling can follow up to build uFETCH indexes.

Algorithm 2 ***BuildIndex***

Input : a scheme key $\boldsymbol{k} = \{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}, \boldsymbol{n}\}$, a datum $\boldsymbol{d}$
Output: a uFETCH index $\boldsymbol{idx}$

```
   // S1. project onto a N-char binary word
1  if d is a number then
2      w ← ∅
3      for i ← 1 to |n| do
4          β ← (d > n_i) // project onto {0,1}
5          w ← w‖β
6  else
7      w ← d
   // S2. pad short words to L-char long
8  h ← f(k̄, |w|)
9  if |w| < L then
10     w ← w‖h[(|w| + 1)...L]
   // S3. pattern conceal and make N-char long
11 p ← ∅
12 for i ← 1 to ⌈N/|w|⌉ do
13     ẇ ← w ⊕ h[(i × |w| + 1)...((i + 1) × |w|)]
14     p ← p‖ẇ
15 p ← p[1...N]
   // S4. build an index
16 i ← (Rp)^≻ with (·)^≻ ≜ f(k̄, ·)[1...1]
17 randomly get a Frequency Garbler G
18 idx ← G^⊤ i
```

BuildIndex comprises stages of `S1` to `S4`. `S1` checks if a given datum $\boldsymbol{d}$ is a number. If asserted, it transforms $\boldsymbol{d}$ into a N-character word. That is, N binary comparisons are conducted with their binary results concatenated ($\|$) into a word $\boldsymbol{w}$. Thereby, after `S1`, $\boldsymbol{w}$ is obtained even if it is a number. In `S2`, by a pseudo random-random function (PRF) $f(\bar{\boldsymbol{k}}, \cdot)$, a string of PRF characters is generated with the length of $\boldsymbol{w}$ as the seed. The first part of the string is used to pad short words to make them at least L-character long. The remaining PRF characters are used in `S3` to wipe out the pattern of $\boldsymbol{w}$ by XOR. `S3` repeats XORing until a N-character $\boldsymbol{p}$ is concatenated. Finally, the background primitives are engaged in `S4` to build the corresponding SE index. Note that, the PRF $f(\bar{\boldsymbol{k}}, \cdot)$ can also be implemented by keyed-hashing such as HMAC, and

we index the characters of a PRF string by $[a \ldots b]$, stating a-th to b-th character is used.

It can be observed that `S1` is so simple that, when a given datum $\boldsymbol{d}$ is a large (small) number, `S1` could result in a word $\boldsymbol{w}$ comprising many `1`'s (`0`'s) as $\boldsymbol{d}$ is larger (smaller) than most of $n_1, n_2, \ldots, n_N$. In an extreme case, $\boldsymbol{w}$ could be a string of "`111...1`" or "`000..0`" that we cautioned. This, however, is fine because in `S3` such patterns will be wiped out after `XOR`ing with the PRF sequence $\boldsymbol{h}$.

C. uFETCH TRAPDOOR

To build a trapdoor that can identify uFETCH indexes under the same scheme key $\boldsymbol{k}$, *Trapdoor* first checks whether a given query $\boldsymbol{q}$ is a word or a range. When a word is given, similar steps are taken to normalize it a word of N-character long, except any wildcard Δ will "propagate" through normalization.

Algorithm 3 *Trapdoor*

Input : a scheme key $\boldsymbol{k} = \{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}, \boldsymbol{n}\}$, a query $\boldsymbol{q}$
Output: a uFETCH trapdoor $\boldsymbol{trap}$

```
   // S1. project onto a N-char ternary word
1  if q is a range then
2  |  w ← ∅
3  |  for i ← 1 to |n| do
4  |  |  τ ← (q > n_i) // project onto { 0,1,Δ}
5  |  |  w ← w‖τ
6  else
7  |  w ← q
   // S2. pad short query words to L-char long
8  h ← f(k̄, |w|)
9  if |w| < L then
10 |  w ← w‖h[(|w| + 1)...L]
   // S3. pattern conceal and make N-char long
11 q̃ ← ∅
12 for i ← 1 to ⌈N/|w|⌉ do
13 |  ẇ ← w ⊕ h[(i × |w| + 1)...((i + 1) × |w|)]
14 |  q̃ ← q̃‖ẇ
15 q̃ ← q̃[1...N]
   // S4. build a trapdoor
16 t ← (Rq̃)^≻ with (·)^≻ ≜ f(k̄, ·)[1...1]
17 randomly get a Frequency Sift S
18 trap ← S · t
19 remove each Δ from trap , if any
20 if |trap| > ⌊λ/2⌋ then
21 |  randomly remove (|trap| − ⌊λ/2⌋) character(s) from
   |  trap
```

Aligned with the pseudo codes of *BuildIndex*, when a numerical range is given, $\boldsymbol{q}$ has to be first projected onto a N-character word so as to comply with `S2`. Since a range is not a number but stands for a set of numbers usually bounded by two numbers, say (q_1, q_2), the comparison in the line `4` requires special treatment. Specifically, the condition $>$ is `true` and symbolized as `1` if and only if both q_1 and q_2 are larger than n_i. In case that both q_1 and q_2 are NOT larger than n_i, the result is `false` and symbolized as `0`. Otherwise, Δ is designated. For example, because `(10,20]>15` is neither true nor false, Δ is designated for that comparison. Thereby, after `S1`, a range is made into a N-character *ternary* word of characters $\{0, 1, \Delta\}$.

If a wildcard query is given, `S1` is skipped and `S2` takes place to pad the query to be at least L-character long if it is not. But padded or not, `S3` follows up to make all words to be N-character long while using `XOR`ing with PRF sequence $\boldsymbol{h}$ to conceal original word pattern. Note that, the protocol of wildcard propagation will make each wildcard Δ propagated through XOR. Thereby, if $\boldsymbol{q}$ contains Δ, the word $\tilde{\boldsymbol{q}}$ input to `S4` will also contain propagated Δ('s).

In `S4`, after the steps of reflecting, projecting and sifting (in lines `16`∼ `18`), each Δ is removed from $\boldsymbol{trap}$. If $\boldsymbol{trap}$ is longer than the threshold $\lfloor \lambda/2 \rfloor$, the remaining characters are randomly trimmed out. The trimming tries to make the resulted uFETCH trapdoor to be of the same length of $\lfloor \lambda/2 \rfloor$ if possible. It helps conceal search pattern as to be discussed.

D. uFETCH ENCRYPTED SEARCH

What makes uFETCH extraordinary and attractive is that it transforms the problem of encrypted search into the simple problem of subsequence matching, with trapdoors telling nothing about the data type being searched.

Algorithm 4 *Match*

Input : an SE trapdoor $\boldsymbol{trap}$, an SE index $\boldsymbol{idx}$
Output: True or False

1 *output* True *if* $\boldsymbol{trap} \sqsupseteq \boldsymbol{idx}$*; otherwise, output* False

As shown in the algorithm *Match*, to check if a query identifies any encrypted item, cloud repository can simply check if a given SE trapdoor $\boldsymbol{trap}$ is a subsequence to any stored SE index $\boldsymbol{idx}$, as denoted by the operator $\sqsupseteq$. With a typeless trapdoor, cloud repository is made ignorant of the data type when it is commissioned to search, not even the number of wildcards in use if any. Thanks to the implication of (14), under the same scheme key $\boldsymbol{k} = \{\boldsymbol{R}, \bar{\boldsymbol{k}}, \boldsymbol{b}, \boldsymbol{n}\}$, uFETCH brings a perfect recall rate of 1 regardless of the configuration of $N, M, \alpha, \lambda, (min, max)$ and L. Though, for a good precision, configuration does matter as exemplified in Section V.

E. SECURITY DISCUSSIONS

Typically, CSPs provide their services running in the middle of a 3-tier structure as shown in Fig. 3, wherein private data fed from the tier-1 are processed in tier-2 but stored in the tier-3. Such a 3-tier structure is widely-adopted because the tier-3 cloud repositories can free CSP from tedious-yet-serious storage management.

In the 3-tier cloud structure, repository providers see each CSP as a tenant regardless how many users it has. In fact,

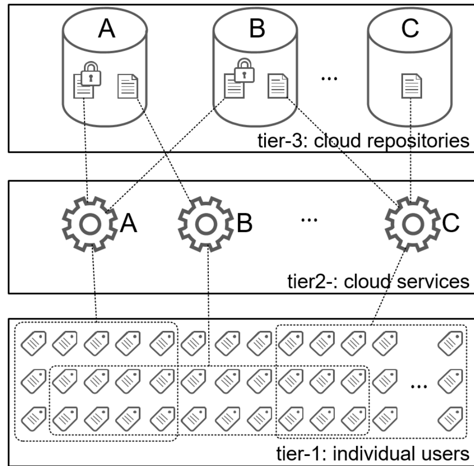

FIGURE 3. The widely-adopted 3-tier cloud structure with examples of repository providers seeing services A, B and C as tenants, but not able to identify users of service A because of encryption.

assuming no collusion between the CSPs and repository providers, repository providers cannot know the number nor the identity of users in tier-1 if CSP enforces encryption, just as what cloud service A does in Fig. 3. Please note that, to utilize any security proxy/agent such as CryptDB [31] or the one we are about to propose in tier-2, it is assumed that CSP is allowed to see private data in plaintext. Though CSP is entrusted by tier-1 users to see data in order to process them as fast as possible for the service it offers, CSP has to make tier-3 cloud repository ignorant of all the (processed) data it uploads This assumption is generic to SE schemes and existing SE-integrated tools such as CryptDB [31], MONOMI [33], L-EncDB [22] and Seabed [34] as long as they are used in tier-2 to help CSP protect tier-1 user privacy against adversarial DBMS in tier-3.

With the 3-tier cloud structure, security notions such as IND-CKA [6], [7] seem ''over-spec'' to assure each SE index indistinguishable from one another. Though notions as such are tough, they are actually meant for 2-tier client-server structure, wherein user identity is often known. As user identity can be exploited to apply domain-knowledge or side-channel to compromise the meaning of certain SE index, IND-CKA prevents any compromised SE index from breaching another.

However, it is not the case with the 3-tier cloud structure, assuming all data are anonymised by encryption and put under tenants' umbrella. Thereby, it is arguable that SE indexes have no need to be indistinguishable from one another, as long as they properly conceal the data.

1) CFA-IMMUNITY IN 3-TIER CLOUD STRUCTURE

Assuming encryption is enforced in the 3-tier cloud structure, with no user identity to exploit, adversaries inside cloud repository can be modelled as accessible to some open dictionaries. However, this alone could lead to the notorious *Character Frequency Attack* (CFA).

uFETCH is immune to CFA. By reflecting, projecting and engaging Garbler $\boldsymbol{G}$ and Sift $\boldsymbol{S}$, it makes character

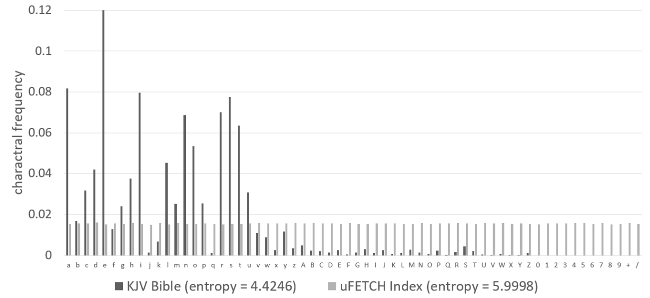

FIGURE 4. Character distributions before and after uFETCH ***BuildIndex*** under an example configuration, wherein the typical English distribution of KJV Bible is transformed into one that is close to White Noise.

frequency eliminated (FE). Namely, character frequencies are ''merged'' after reflecting-and-projecting, with the merged frequencies further ''blended'' by Garbler $\boldsymbol{G}$ and Sift $\boldsymbol{S}$ before producing index and trapdoor respectively. From uFETCH indexes and trapdoors, adversaries are only left with severely distorted frequencies that are useless to launch CFA.

Furthermore, uFETCH can conceal *search pattern* [7] to prevent user's query behaviour from being analyzed. That is, given the same query repeatedly, *Trapdoor* is able to build different trapdoors with all reaching the same goal. *Trapdoor* also tries to make all trapdoors the same length to conceal the wildcard usage in a textual query or the wideness of a range query. By randomly punching out character(s), *Trapdoor* builds dynamic trapdoors that seem to have *trapdoor characters hopping* (TCH) around. Combining the techniques of FE and TCH, uFETCH protects its indexes and trapdoors from CFA.

2) MAKING INDEX ENTROPY CLOSE TO WHITE NOISE

Whenever necessary, uFETCH can actually be better than CFA-immunity. As a case study, using a configuration of $N = 40, M = 200, \alpha = 16, \lambda = 40, L = 7, (min, max) = (0, 100)$ and HMAC-SHA256 as the PRF f with `base64` characters output, the entropy of built indexes can be made quite close to that of White Noise. Though this configuration pulls down precision (comparing to the instance in Section V), it can even be considered for the 2-tier client-server structure.

Using KJV Bible as an example, we first check the ensemble character distribution along with the entropy of all unique words in KJV Bible. Then, after *BuildIndex*, the ensemble character distribution and the entropy of the corresponding uFETCH indexes are examined. As the perfect entropy of `base64` is 6, uFETCH can make characters distributed very close to White Noise, with an near-perfect entropy of 5.9998 as shown in Fig. 4.

3) MAKING INDEX INDISTINGUISHABLE

To check how indistinguishable uFETCH indexes can be, under the same configuration, uFETCH indexes are built for alphabetically-similar words and put along with the uFETCH indexes for numbers within `(0,100)`. We check if similar words or nearby numbers will have their uFETCH indexes related to each other.

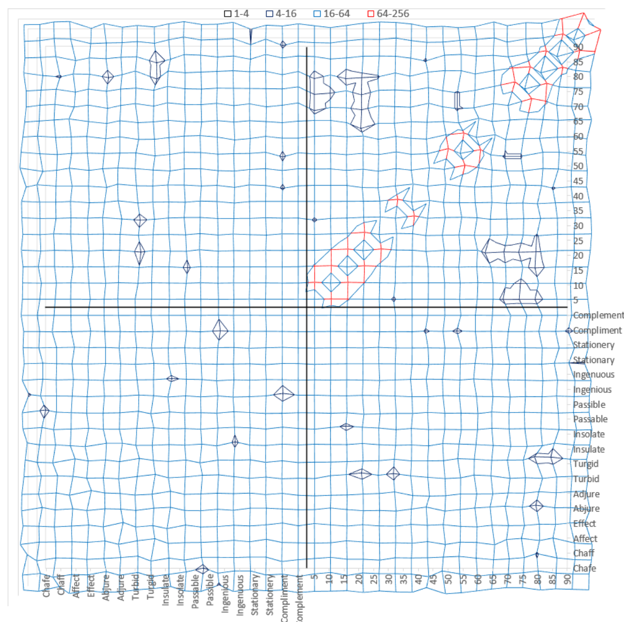

FIGURE 5. Measurement of LCS length as a distance between uFETCH indexes built for words and numbers.

Why similar words? Intuitively, with the same uFETCH scheme key $\boldsymbol{k}$, it can be anticipated that similar words will get reflected-and-projected similarly. Thus, uFETCH indexes built for similar words might have same characters spread as the common subsequence between uFETCH indexes. There is the same concern for nearby numbers. Therefore, we are curious about whether uFETCH can even eliminate such a clue, if necessary.

As shown Fig. 5, longest common subsequence (LCS) length between FETCH indexes is measured to see if it distinguishes similar words or nearby numbers. Because of the noise resulted from "redundant" characters prepared in rooms for TCH, the LCS length between similar words and not-similar words can be quite indistinguishable. However, nearby numbers would have indexes with longer LCS. Though this clue could be used to "group" nearby numbers, it tells no relative magnitude as OPE [30] leaks.

4) FLATTENING NUMERICAL INDEXES

uFETCH can be seen as a variant to the bucketing technique proposed by Hacigümüş *et al.* [24] because numbers are virtually "bucketed" by uFETCH indexes, with each uFETCH index designated as an identifier. Therefore, the distribution of uFETCH indexes reveals the numerical distribution.

To conceal the distribution, as proposed by Hore *et al.* [38], it is good to adjust bucketing so that each bucket has roughly the same quantity of items. uFETCH offers an option to do so. By generating the secret numbers $n_1, n_2, \ldots, n_N$ according to the known distribution to be concealed, the index distribution can be flattened. For example, given a dataset of Gaussian distribution, secret numbers with Gaussian distribution can be generated and used in the scheme key $\boldsymbol{k}$ accordingly.

IV. SECURITY AGENT FOR OFF-THE-SHELF DBMS

uFETCH can bring up simpler designs that make off-the-shelf DBMS privacy-preserving. To demonstrate the simplicity, a security agent is built that relies only on the uFETCH for encrypted-search across data types, instead of integrating heterogeneous techniques.

As shown in Fig. 6, the security agent resides with CSP in tier-2 to let it move private data to tier-3 DBMS with privacy preserved. Whenever data are fed from users in tier-1, CSP stores the (processed) data to the tier-3 DBMS using SQL statements as usual but only via the security agent. CSP can of course query and update data by SQL as well. With the keys specified and kept by CSP, the security agent "translates" all SQL statements into SQL-aware encrypted statements so that all data are concealed before landing on tier-3. The security agent processes SQL statements with three overhead levels: `L1`, `L2` and `L3`.

A. L1: SIMPLE TRANSLATION

As a database has at least one table, `CREATE TABLE` is always the first SQL statement that CSP has to begin with. `CREATE TABLE` configures a new table by pairs of {*fieldname*, *type*}. For example, to create a table `club` with four fields of `ID`, `name`, `age` and `weight`, four types of `INT`, `CHAR(40)`, `INT` and `FLOAT` are paired respectively.

Given one `CREATE TABLE` statement, the security agent actually derives two `CREATE TABLE` statements, with one to ask the tier-3 DBMS to create a new table (e.g. `club`) for storing encrypted data, and the other to create a new table (e.g. `club_cnf`) in a local DBMS to keep the pairs of {*fieldname*, *type*}. While the table in tier-3 DBMS would grow and become enormous, the local one is static and small.

To derive the statement to tier-3 DBMS, simple translation is conducted over all fields specified by `CREATE TABLE` (except field(s) of no privacy concern such as auto-serialized `ID`). Thus, for each pair of {*fieldname*, *type*}

- *type* is changed to `VARCHAR`
- an extra pair of {*fieldname*`_idx`, `VARCHAR`} is added

Then, with a table created as specified in tier-3 DBMS, CSP can populate it by `INSERT` statements. As `INSERT` specifies pairs of {*fieldname*, *datum*}, simple translation is conducted for each pair such that

- *datum* encrypted by encryption such as AES
- an extra pair of {*fieldname*`_idx`, *BuildIndex*($\boldsymbol{k}$, *datum*)} is added, with $\boldsymbol{k}$ a uFETCH scheme

Note that, uFETCH builds an ***idx*** for a number regardless it is of `INT`, `FLOAT` or types of other resolution, and a unified field, e.g. `VARCHAR`, suffices to accommodate it as a string.

B. L2: SECOND-STAGE FILTERING

After a table is populated with encrypted data, CSP can selectively retrieve them back by `SELECT` statements via the security agent. However, because of possible false-positives, such a selective retrieval requires a *second-stage filtering* that slightly pushes the overhead to `L2`.

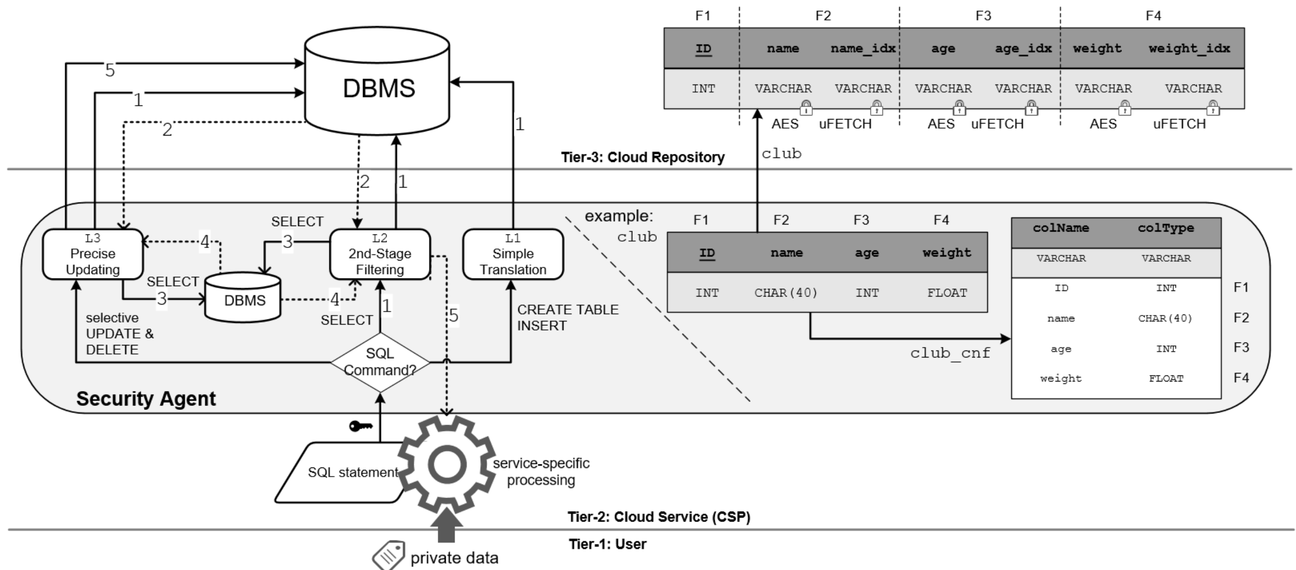

FIGURE 6. Using uFETCH, the security agent can be built at ease with three levels of overhead, i.e. L1, L2 and L3.

A SELECT statement often comes with WHERE followed by conditions of {*fieldname operator operand*}. Translation is enforced on each condition such that

- *fieldname* replaced by *fieldname*_idx
- *operator* replaced by LIKE
- *operand* replaced by *Trapdoor*($\boldsymbol{k}$, *operand*) with the SQL wildcard % put in front of, in rear of and in between all trapdoor characters

The translated SELECT asks tier-3 DBMS to conduct subsequence matching on the index field, instead of the field of encrypted data, and is sent as L2:1 in Fig. 6. After a result set is back (L2:2), the security agent decrypts it and put the plaintext one into the local DBMS for second-stage filtering by the original SELECT (L2:3−4). Then the result set with a perfect precision is returned to CSP (L2:5) with each *type* restored according to the local table (e.g. club_cnf).

Note that, the security agent recognizes and combines two conditions into one if they constitute a range query. For example, SELECT * from club WHERE age > 16 and age < 28 will be translated into SELECT age from club WHERE age_idx LIKE *Trapdoor*($\boldsymbol{k}$, (16, 28)), with SQL wildcard % put in front of, in rear of and in between trapdoor characters for subsequence matching. However, for a single-ended comparison, *min* or *max* will be used to complete a range. For example, *Trapdoor*($\boldsymbol{k}$, (16, *max*)) is used to build a trapdoor for age > 16.

C. L3: SELECTIVE UPDATING

Besides SELECT statements, UPDATE and DELETE can also come with WHERE followed by conditions. In such cases, similar translation can also be conducted over the conditions. However, to avoid corrupting data because of false-positives, the security agent first sends SELECT instead.

That is, given a UPDATE or DELETE statement, a SELECT statement with WHERE appended with translated conditions (pruned from UPDATE or DELETE) is sent as L3:1 in Fig. 6. After the tier-3 DBMS returns a result set L3:2, the security agent decrypts it and put the plaintext one into the local DBMS for second-stage filtering (L3:3−4) by the SELECT except this time all conditions are in plaintext. With all false-positives filtered out, the result set gives the security agent a result set with precise identifiers $id_1, id_2, \ldots$ (e.g. of the field ID) to update or delete. Thereby, as shown by L3:5, an UPDATE or DELETE statement is sent to tier-3 DBMS with WHERE followed by a list of ID $=$ id_1 OR ID $=$ id_2 OR …to update affected records precisely, with affected fields "simple translated" as L1 does.

V. EXPERIMENTAL RESULT

A computer with an Intel i7-7700 CPU running at 3.6GHz with 16GB RAM is used in tier-2 to run both a security agent and a CSP client. The CSP client creates the example table club in tier-3 and populates it with 2^{10} to 2^{19} records. Each record has a name randomly-imported from a name dataset [39] along with age, weight uniform-distributed in (0,100).

We set up tier-3 DBMS, i.e. a MySQL server, within the same computer to eliminate network inference while our security agent can be replaced by CryptDB [31] for comparison. To leverage MySQL internal index for better search speed, an index is created for each field of club. The security agent is equipped with a uFETCH instance configured with $N = 40$, $M = 200$, $\alpha = 7$, $\lambda = 40$ and $L = 16$ with (*min*, *max*) set to (0, 100). If not explicitly stated, each of measured points are an average of 1,000 randomly-generated samples.

Note that, there are works also putting efforts on heterogeneous integration such as MONOMI [33], L-EncDB [22] and

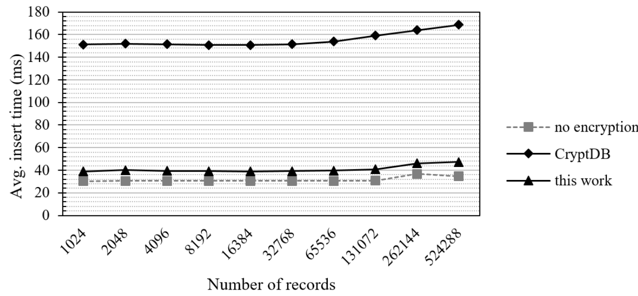

FIGURE 7. L1 overhead of SQL INSERT in average of all inserted records.

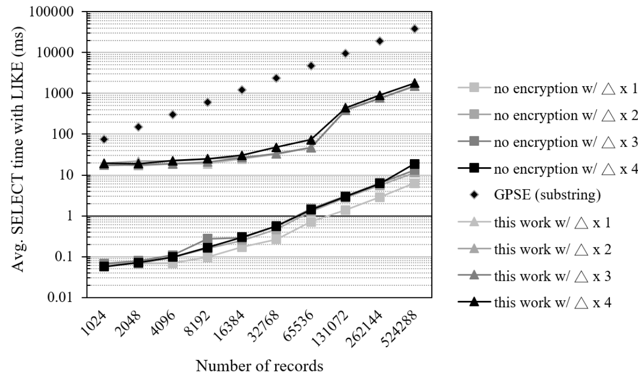

FIGURE 8. L2 overhead of SQL SELECT with LIKE for wildcard query.

Seabed [34]. However, only CryptDB is open to be evaluated and put along with this work.

A. L1 OVERHEAD

There are two SQL statements, i.e. `CREATE TABLE` and `INSERT`, that require only simple translation and thus bring `L1` overhead. Among them, only `INSERT` is worthy of evaluating because the overhead of `CREATE TABLE` is very little and once for all.

As shown in Fig. 7, after CSP incrementally `INSERT` 2^{10} to 2^{19} records, we observe the average insert time of the security agent very close to that of no encryption, i.e. executing SQL statements in plaintext.

However, given the same dataset, CryptDB requires about at least 150ms to make each `INSERT` done in average. Its slower performance might be due to the heterogeneous integration of SE, OPE and HE. Besides, in our experiments, CryptDB actually failed to complete all operations because of out-of-memory. Specifically, CryptDB crashes when dealing with 2^{18} inserted records. Therefore, points for 2^{18} and 2^{19} records are extrapolated from the measured points for 2^{16} and 2^{17} for CryptDB in Fig. 7 and Fig. 9.

B. L2, L3 OVERHEAD

Because of the second-stage filtering, we categorize the SQL statements of `SELECT` with a `WHERE` clause as having `L2` overhead. However, there are two cases to be observed, namely, `SELECT` by searching among encrypted texts, and `SELECT` by searching among encrypted numbers.

In case of searching among encrypted texts, uFETCH enables the security agent to retain the handy SQL operator

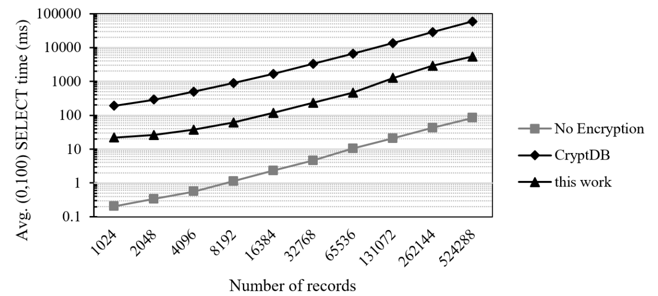

FIGURE 9. L2 overhead of SQL SELECT conditioned with range.

`LIKE` for wildcard queries. To measure the overhead of such `L2` operations, CSP feeds the security agent with the statements of `SELECT * FROM club WHERE name LIKE q`, with `q` randomly-picked from inserted names but with zero to four characters replaced by wildcards Δ. (Note that, it is actually the _ used in SQL. Δ is used here for symbol consistency.) For each of the statements, the security agent conducts SQL translation as depicted in Section IV-B and we observe that the steps `L2:1∼5` would bring about 100-time slowdown, i.e. 1.6 second, comparing to no encryption as shown in Fig. 8 when facing 2^{19} records. As CryptDB does not support wildcard query, the best-reported wildcard SE speed, i.e. GPSE [13], is put in for comparison, even though it is not SaaS-native and thus cannot be used in off-the-shelf DBMS. However, given the same 2^{19} records, GPSE needs extra 38 seconds to just complete search, i.e. a time that not yet includes record retrieval and decryption.

In case of searching among encrypted numbers, uFETCH enables the security agent to support range query also by the operator `LIKE` as explained in subsection IV-B. To measure the overhead, CSP produces statements of `SELECT * FROM club WHERE age > r1 AND age < r2`, with `r1` smaller than `r2` and both randomly-generated within `(0, 100)`. As shown in Fig. 9, in average, the `SELECT` with range conducted by the security agent is about 65-time slower than no encryption when facing 2^{19} records. However, it is nevertheless about 14-time faster than CryptDB given the same amount of records. Note that, each measured point is an average of 5,000 `SELECT` statements with randomly-generated ranges of 10%, 20%, 30%, 40% and 50% wide to the maximum `(0, 100)`, wherein each wideness contributes one-fifth of the samples.

We do not plot the `L3` overhead because SQL statements of `L3` dynamically depends on how many records are actually updated or deleted. However, since such an overhead can be seen as `L2 SELECT` plus `L1` update or delete, i.e. `UPDATE` or `DELETE` with precise identifiers, we consider Fig. 7, 8 and 9 are sufficient for case-by-case `L3` estimation.

C. TEXTUAL IR PERFORMANCE

Though *externally*, CSP always receives from the security agent a result set with perfect IR [4] as expected, it is important to know how the *internal* IR performs because it implies additional costs of communication, memory and latency for

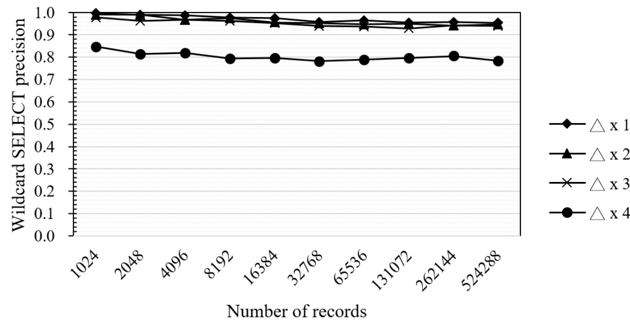

FIGURE 10. Precision of SQL SELECT of wildcardized textual queries, with zero to four wildcards in use.

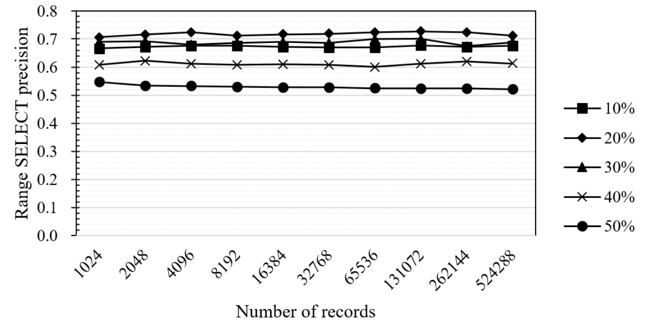

FIGURE 11. Precision of SQL SELECT with queried ranges that are 10% to 50% wide to the configured (*min*, *max*).

the sake of privacy-preserving. Since uFETCH guarantees a perfect recall, we only have to measure the precision after `L2:2`.

For searching among encrypted texts, this IR experiment is targeting the field `name` and is conducted along with the `L2` overhead experiment. The underlying dataset [39] provides names with average length of 6.19 characters, i.e. about 2-character shorter than English words. Though uFETCH gives better precision with longer words, the name dataset can be seen as a corner case. However, because we will measure the precision of wildcard queries with up to four wildcards in use, names shorter than five characters are not used as the `q` to be made into wildcard queries.

As shown in Fig. 10, given 2^{10} to 2^{19} randomly-picked names, the precision is checked with respect to the number of wildcards Δ in use. Wildcard queries with less than four wildcards are very close to perfect, i.e. 1. However, when the number of wildcards is set to four, the precision drops substantially. It is due to the wildcard propagation mentioned in Section II-B and limits how the security agent can support SQL wildcard `%` that represents arbitrary number of characters. For example, as `%` can be implemented by `OR`ing wildcard queries with zero to multiple Δ, the security agent can only support up to four wildcards Δ under this uFETCH instance if a decent precision is required.

D. NUMERICAL IR PERFORMANCE

As uFETCH enables numerical search by means of wildcard queries, its numerical IR precision is related to it. Intuitively, the precision of a range query would drop if the range is too wide as it leads to more wildcards in use.

As shown in Fig. 11, the best precision of queried ranges is about 0.7 with the uFETCH instance. The precision drops when queried range is getting wider. When the queried range reaches 50% of the (*max*, *min*), the precision reduces to 0.5. This is actual the lower bound for *wide* range queries due to too many wildcards in use. In fact, in the case of 50%, all numbers are retrieved back and the precision of 0.5 is the worst one can expect for uniform-distributed numbers.

One might come across an anti-intuition that the precision of 10%-wide range queries is worse than that of 20%. It is actually a phenomenon due to *narrow* range queries. Their precision cannot be properly measured because it highly depends on the configured N and the volume of numbers being indexed. That is, since uFETCH virtually partitions numbers into $N + 1$ buckets with N secret numbers, nearby numbers tend to be bucketed together. When the trapdoor of a range query matches a bucket, all numbers in the bucket will be retrieved. However, as the query can be made very narrow comparing to the "nearby" distance, the precision can be made very poor. Fortunately, this poor precision can be improved by an option depicted below.

E. OPTION FOR NUMERICAL IMPROVEMENT

For clarity, in this work we evaluated the performance of encrypted numerical search by only picking fields of similar ranges, i.e. `age` and `weight`. However, for *not*-similar-range fields such as height and income, the security agent has to set a different (*min*, *max*) for each field, leading to multiple uFETCH instances in use. Nevertheless, it is not hard to keep just one uFETCH instance. That is, one can first represent any number by its floating-point representation with *mantissa* and *exponent* separately normalized to (*min*, *max*) and both indexed by an uFETCH instance. Though this will require two index fields (instead of one) and a bit more complicated SQL translation addressing two fields at the same time, it is worthy because it also improves the precision of the mentioned narrow range queries.

VI. CONCLUSION

This work proposes a unified SE scheme named uFETCH for both textual and numerical data. Using unified index structure and search algorithm, uFETCH enables encrypted-search even if SE indexes across types are mingled. uFETCH offers efficient selective retrieval by transforming the problem of encrypted-search into a simple problem of subsequence matching for SaaS-native, regardless the encountered dataset contains texts, numbers or both. uFETCH is built for efficiency with a security dedicated to the widely-deployed 3-tier cloud structure.

To show how uFETCH can bring up simplicity, a security agent is demonstrated that translates SQL into three levels of SQL-aware encrypted statements, making existing DBMS privacy-preserving. Our experimental results affirm

low overhead and decent IR precision. In fact, comparing with CryptDB, i.e. a popular SQL security proxy, uFETCH brings not only a simpler design, but also an even lower overhead. With adequate search speed and the adequate security, uFETCH is a practical means for CSP to ease compliance with privacy regulations.

REFERENCES

[1] *General Data Protection Regulation*. Accessed: Mar. 25, 2020. [Online]. Available: https://gdpr-info.eu

[2] J. Mei, K. Li, A. Ouyang, and K. Li, "A profit maximization scheme with guaranteed quality of service in cloud computing," *IEEE Trans. Comput.*, vol. 64, no. 11, pp. 3064–3078, Nov. 2015.

[3] O. Goldreich, *Foundations of Cryptography: Basic Applications*, vol. 2. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[4] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York, NY, USA: ACM Press, 1999.

[5] D. Xiaoding Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy. (S&P)*, May 2000, pp. 44–55.

[6] E.-J. Goh, "Secure indexes," in *Proc. IACR Cryptol. ePrint Arch.*, 2003, pp. 1–18.

[7] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. CCS ACM*, 2006, pp. 79–88.

[8] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–5.

[9] A. Awad, A. Matthews, Y. Qiao, and B. Lee, "Chaotic searchable encryption for mobile cloud storage," *IEEE Trans. Cloud Comput.*, vol. 6, no. 2, pp. 440–452, Apr. 2018.

[10] C. Bösch, R. Brinkman, P. Hartel, and W. Jonker, "Conjunctive wildcard search over encrypted data," in *Proc. SDM*, 2011, pp. 114–117.

[11] T. Suga, T. Nishide, and K. Sakurai, "Secure keyword search using Bloom filter with specified character positions," in *Proc. Int. Conf. Provable Secur.*, 2012, pp. 235–252.

[12] S. Sedghi, P. V. Liesdonk, S. Nikova, P. H. Hartel, and W. Jonker, "Searching keywords with wildcards on encrypted data," in *Proc. SCN*, in Lecture Notes in Computer Science, vol. 6280, 2010, pp. 138–153.

[13] D. Wang, X. Jia, C. Wang, K. Yang, S. Fu, and M. Xu, "Generalized pattern matching string search on encrypted data in cloud systems," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 2101–2109.

[14] C. Hu and L. Han, "Efficient wildcard search over encrypted data," *Int. J. Inf. Secur.*, vol. 15, no. 5, pp. 539–547, 2016.

[15] Y. Yang, X. Liu, R. H. Deng, and J. Weng, "Flexible wildcard searchable encryption system," *IEEE Trans. Services Comput.*, early access, Jun. 12, 2017, doi: 10.1109/TSC.2017.2714669.

[16] P. Golle, J. Staddon, and B. Waters, "Secure conjunctive keyword search over encrypted data," in *Applied Cryptography and Network Security* (Lecture Notes in Computer Science), vol. 3089. Berlin, Germany: Springer, 2004, pp. 31–45.

[17] L. Ballard, S. Kamara, and F. Monrose, "Achieving efficient conjunctive keyword searches over encrypted data," in *Information and Communications Security*, (Lecture Notes in Computer Science), vol. 3783. Cham, Switzerland: Springer, 2005, pp. 414–426.

[18] Y. Yang and M. Ma, "Conjunctive keyword search with designated tester and timing enabled proxy re-encryption function for E-health clouds," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 4, pp. 746–759, Apr. 2016.

[19] Y. Tang, D. Gu, N. Ding, and H. Lu, "Phrase search over encrypted data with symmetric encryption scheme," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst. Workshops*, Jun. 2012, pp. 471–480.

[20] Z. A. Kissel and J. Wang, "Verifiable phrase search over encrypted data secure against a semi-honest-but-curious adversary," in *Proc. IEEE 33rd Int. Conf. Distrib. Comput. Syst. Workshops*, Jul. 2013, pp. 126–131.

[21] S. Zittrower and C. C. Zou, "Encrypted phrase searching in the cloud," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2012, pp. 764–770.

[22] J. Li, Z. Liu, X. Chen, F. Xhafa, X. Tan, and D. S. Wong, "L-EncDB: A lightweight framework for privacy-preserving data queries in cloud computing," *Knowl.-Based Syst.*, vol. 79, pp. 18–26, May 2015.

[23] C. Guo, X. Chen, Y. Jie, F. Zhangjie, M. Li, and B. Feng, "Dynamic multi-phrase ranked search over encrypted data with symmetric searchable encryption," *IEEE Trans. Services Comput.*, early access, Oct. 30, 2017, doi: 10.1109/TSC.2017.2768045.

[24] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 216–227.

[25] P. Chen, W. Zeng, Y. Zhu, and Y. Gu, "Secure range query based on privacy-preserving function in two-tiered sensor networks," in *Proc. Int. Conf. Secur. Smart Cities, Ind. Control Syst. Commun. (SSIC)*, Jul. 2016, pp. 1–6.

[26] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar, "Fast range query processing with strong privacy protection for cloud computing," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1953–1964, Oct. 2014.

[27] S. Bajaj and R. Sion, "TrustedDB: A trusted hardware-based database with privacy and data confidentiality," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 752–765, Mar. 2014.

[28] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers, "SMCQL: Secure querying for federated databases," *Proc. VLDB Endowment*, vol. 10, no. 6, pp. 673–684, Feb. 2017.

[29] J. Li and E. R. Omiecinski, "Efficiency and security trade-off in supporting range queries on encrypted databases," in *Proc. Working Conf. Data Appl. Secur. (DBSec)*, 2005, pp. 69–83.

[30] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 563–574.

[31] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proc. 23rd ACM Symp. Oper. Syst. Princ. (SOSP)*, 2011, pp. 85–100.

[32] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. EUROCRYPT*, 1999, pp. 223–238.

[33] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 289–300, Mar. 2013.

[34] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (SDI)*, Oct. 2016, pp. 587–602.

[35] S.-M. Chung, M.-D. Shieh, and T.-C. Chiueh, "A SaaS-native wildcard searchable encryption scheme for protecting privacy in cloud services," in *Proc. IEEE Int. Conf. Big Data Intell. Comput.*, 2019, pp. 178–184.

[36] I. N. Bronshtein, K. A. Semendyayev, G. Musiol, and H. Muehlig, *Handbook of Mathematics*, vol. 2. New York, NY, USA: Springer-Verlag, 2004.

[37] S.-M. Chung, M.-D. Shieh, and T.-C. Chiueh, "FETCH: A cloud-native searchable encryption scheme enabling efficient pattern search on encrypted data within cloud services," *Int. J. Commun. Syst.*, p. e4141, Dec. 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.4141

[38] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proc. 13th Int. Conf. Very Large Data Bases*, vol. 30, 2004, pp. 720–731.

[39] *Popular Baby Names, National Data*. Accessed: Mar. 25, 2020. [Online]. Available: https://www.ssa.gov/OACT/babynames/names.zip

SHEN-MING CHUNG received the B.S. and M.S. degrees in electronic engineering from the National Yunlin University of Science and Technology, Taiwan, in 2001 and 2003, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering, National Cheng Kung University, Taiwan. He joined the Industrial Technology Research Institute (ITRI), Taiwan. He worked as an Engineer. In recent years, he has focused on the area of searchable encryption tailored for industrial scenarios in order to address regulation-induced issues. His research interests include networking, communication, and the security-related issues thereof.

MING-DER SHIEH (Member, IEEE) received the B.S. degree in electrical engineering from National Cheng Kung University, Tainan, Taiwan, in 1984, the M.S. degree in electronic engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1986, and the Ph.D. degree in electrical engineering from Michigan State University, East Lansing, MI, USA, in 1993. From 1988 to 1989, he was an Engineer with United Microelectronic Corporation, Hsinchu. From 1993 to 2002, he was with the Faculty of the Department of Electronic Engineering, National Yunlin University of Science and Technology (NYUST), Douliu, Taiwan. From 1999 to 2002, he was the Department Chairman with NYUST. Since 2002, he has been with the Department of Electrical Engineering, National Cheng Kung University. From 2010 to 2014, he was the Deputy General Director of Information and Communications Research Laboratories, Industrial Technology Research Institute (ITRI), Taiwan. From 2014 to 2017, he was the Department Chairman with National Cheng Kung University, where he is currently a Full Professor. His current research interests include very-large-scale integration (VLSI) design and testing, VLSI for signal processing, and digital communication. He was a technical committee member in several international conferences. He received the Teaching Award from NYUST, in 1998. He was the Program Co-Chair and General Co-Chair of the Asian Test Symposium, in 2004 and 2009, respectively, and the Chairman of the Tainan Chapter of the IEEE Circuits and Systems Society, from 2009 to 2010. From 2010 to 2012, he served as an Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS - PART I and the Lead Guest Editor of a special issue of *Computers Engineering and Electrical* journal, in 2012.

TZI-CKER CHIUEH received the Ph.D. degree in computer science from the University of California at Berkeley. He was a General Director of the Cloud Computing Center for Mobile Applications, Industrial Technology Research Institute (ITRI), Taiwan. He was a Professor with the Computer Science Department, Stony Brook University, Stony Brook, NY, USA. From 2007 to 2009, he was the Director of the Core Research, Symantec Research Labs. He is currently the Vice President and the General Director of Information and Communications Research Laboratories. He has published over 250 technical articles in refereed conferences and journals.

CHIA-CHIA LIU received the bachelor's degree in computer science from Tunghai University, Taichung, Taiwan, in 2018. She is currently pursuing the degree with the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. Her research interests include information security and cryptography.

CHIA-HENG TU received the Ph.D. degree from National Taiwan University (NTU), in 2012. From 2012 to 2015, he was a Research and Development Manager with the Institute for Information Industry. He worked as a Postdoctoral Researcher with MediaTek-NTU Advanced Research Center, NTU, in 2015. He is currently an Assistant Professor with the Department of Computer Science and Information Engineering (CSIE), National Cheng Kung University (NCKU). His research interests are in developing tools, such as computer architecture simulators, performance analyzers/optimizers, and parallelizing compilers for designing/optimizing specialized computer systems.