GridSE: Towards Practical Secure Geographic Search via Prefix Symmetric Searchable Encryption
Ruoyang Guo, Jiarui Li, and Shucheng Yu, Stevens Institute of Technology
https://www.usenix.org/conference/usenixsecurity24/presentation/guo-ruoyang

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA
978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
GridSE: Towards Practical Secure Geographic Search via Prefix Symmetric Searchable Encryption

Ruoyang Guo  
Stevens Institute of Technology

Jiarui Li  
Stevens Institute of Technology

Shucheng Yu  
Stevens Institute of Technology

Abstract

The proliferation of location-based services and applications has brought significant attention to data and location privacy. While general secure computation and privacy-enhancing techniques can partially address this problem, one outstanding challenge is to provide near latency-free search and compatibility with mainstream geographic search techniques, especially the Discrete Global Grid Systems (DGGS). This paper proposes a new construction, namely GridSE, for efficient and DGGS-compatible Secure Geographic Search (SGS) with both backward and forward privacy. We first formulate the notion of a semantic-secure primitive called symmetric prefix predicate encryption (SP²E), for predicting whether or not a keyword contains a given prefix, and provide a construction. Then we extend SP²E for dynamic prefix symmetric searchable encryption (pSSE), namely GridSE, which supports both backward and forward privacy. GridSE only uses lightweight primitives including cryptographic hash and XOR operations and is extremely efficient. Furthermore, we provide a generic pSSE framework that enables prefix search for traditional dynamic SSE that supports only full keyword search. Experimental results over real-world geographic databases of sizes (by the number of entries) from $10^3$ to $10^7$ and mainstream DGGS techniques show that GridSE achieves a speedup of $150 \times - 5000 \times$ on search latency and a saving of $99\%$ on communication overhead as compared to the state-of-the-art. Interestingly, even compared to plaintext search, GridSE introduces only $1.4 \times$ extra computational cost and $0.9 \times$ additional communication cost. Source code of our scheme is available at https://github.com/rykieguo1771/GridSE-RAM.

1 INTRODUCTION

With the increasing deployment of Internet of Things (IoT) devices and the advent of next-generation wireless and mobile communications, the demand for Geographic Information Systems (GIS), particularly Location-Based Services (LBS), and consequently Geographic Searches (GS), has never been greater in various context-aware applications. To deliver accurate LBS services, GS apps usually need to collect fine-grained personal information such as users’ real-time locations and interests. For privacy concerns, data stored in GS servers, which are usually hosted in the cloud, shall be well protected, e.g., via encryption, as a preemptive measure in light of data breaches. Secure Geographic Search (SGS) thus emerges as a promising data protection technique that supports GS over encrypted geographic data.

Existing GS systems, driven by various functionalities (not limited to searches), have been deeply rooted on a so-called Discrete Global Grid (DGG) [1] technique for structuring and indexing two-dimensional spatial data (or higher dimension extension). DGG uses hierarchical tessellation of grids to recursively partition the Earth’s surfaces into geometrical cells which are then coded. A sequence of DGG cell codes describe a hierarchical partition with progressively finer resolution. As shown in Fig. 1, a cell code “dr5r7” represents a cell (sub-region) inside the grid of “dr5r” around the Statue of Liberty. DGG systems (DGGS) can also provide indexes/APIs to support various GIS/LBS services such as geographic search, distance comparison and spatial visualization. In real-world systems, DGGS technologies such as Niemeyer’s Geohash [2], Google S2 [3] and Uber H3 [4] have been widely adopted.

Figure 1: “dr5r” tags an area around the Statue of Liberty. Cell code “dr5r7” tags an area inside “dr5r”.

With DGGS, a GS database is indexed by the hierarchical cell codes. A geographic search is essentially to find a prefix match between the indexed cell codes and the search request. To support SGS wherein both index and search request
are encrypted, the problem can be formulated as the general problem of secure prefix search.

One outstanding challenge with the design of SGS, however, is to assure near latency-free instant responses for pleasant user experiences while compatible with existing GIS/LBS techniques especially DGGS. Theoretically, many advanced techniques, such as Fully Homomorphic Encryption (FHE) [5, 6] and Multi-Party Computation (MPC) [7], could be employed to realize SGS by performing prefix search over ciphertexts. However, current implementations of these techniques are still far from practical when real-time performances are of concern. For example, MPC requires multiple rounds of interactions between client and server for every single query. Current FHE-based secure search is relatively more efficient but still requires thousands of seconds per 10,000 search tuples and a heavy communication cost [5]. On the other hand, emerging privacy-enhancing techniques such as Differential Privacy (DP) can be leveraged for geo-indistinguishability [8] and hence a more efficient SGS. However, such techniques are intrinsically lossy and will cause the so-called lack-in-accuracy issue in geographic services. One branch of privacy-preserving retrieval relies on anonymous technologies like k-anonymity [9], which are constrained by data distribution.

Aiming at secure search over encrypted data, searchable symmetric encryption (SSE) provides a generic but viable solution. However, constructing SGS from existing SSE is by no means trivial, especially considering the dynamics of GS databases (e.g., insertion and deletion of entries) and the privacy implications caused by such dynamics. Specifically, additional privacy, including forward privacy [10–13] and backward privacy [12, 14, 15], shall be considered for dynamic databases. Forward privacy is to ensure newly added data cannot be linked to previous queries while backward privacy further eliminates the link between a query and previously deleted data. These metrics and their definitions construct a strict framework for provable dynamic searchable security.

To our knowledge, the majority of existing SSE techniques focused on keyword search rather than prefix search and thus cannot support SGS, not to mention forward and backward privacy. Few works such as Moataz et. al. [16] enable substring search and provide a semantic-secure SSE but fail to address forward privacy or backward privacy. Moreover, their approach leverages Oblivious RAM (ORAM) which incurs a high communication complexity (quadratic logarithmic). For instance, considering a GS dataset of $10^5$ entries, conducting a single search under this scheme needs at least hundreds of interactions which is far from practical for GIS/LBS applications. Another line of research [17–21] build a specialized index for SGS and all exhibit trade-offs between efficiency, security and false-positive rates.

In this paper, we construct an efficient, secure and DGGS-compatible SGS scheme that supports both forward privacy and backward privacy. To this end, we first formally formulate a novel cryptographic primitive symmetric prefix predicate encryption (SP\textsuperscript{2}E) for predicating a prefix, i.e., evaluating whether a keyword contains a given prefix, and provide a concrete construction. Based on SP\textsuperscript{2}E, we design a new cryptographic algorithm prefix symmetric searchable encryption (pSSE), namely GridSE, for secure geographic search that supports both forward and backward privacy. Interestingly, the construction of GridSE involves only cryptographic hash and XOR operations and is extremely light-weight. Regarding the search complexity, GridSE only introduces an additional cost linearly proportional to the number of DGGS cells, which is practically a constant in real systems and independent to the global GS database size. Thus, GridSE is highly applicable to large-scale GIS/LBS systems.

Our main contributions are summarized as follows:

- We construct an efficient, scalable and DGGS-compatible SGS scheme GridSE that supports both forward and backward privacy. Based on lightweight cryptographic primitives, GridSE is able to offer near real-time latency for large-scale geographic searches.
- We formulate and design a novel cryptographic primitive - symmetric prefix predicate encryption (SP\textsuperscript{2}E) - for predicating prefixes. We prove that SP\textsuperscript{2}E is secure under the random oracle model. Based on SP\textsuperscript{2}E, we construct a new prefix SSE primitive pSSE which is forward and backward secure. Both SP\textsuperscript{2}E and pSSE are generic primitives and can be of independent interests.
- We propose a generic dynamic pSSE framework based on the key idea of GridSE. It supports forward and backward privacy and is also compatible with traditional dynamic SSE that supports only full keyword search.
- Experiments on geographic databases with $10^3 - 10^7$ entries and major DGGS systems (including Niemeyer’s Geohash, Google S2 and Uber H3) show that GridSE achieves a speedup of $150 \times - 5000 \times$, respectively, in search latency and a saving of 99% in communication overhead as compared to the state-of-the-art [16]. Even compared with plaintext geographic search, GridSE only introduces $1.4 \times$ additional computational cost and $0.9 \times$ additional communication cost.

The rest of this paper is organized as follows: Section 2 presents the background. Section 3 lists related works which are followed by preliminaries in Section 4. The SP\textsuperscript{2}E primitive is introduced in Section 5. Section 6 presents our construction of GridSE. The experimental evaluation is shown in Section 7. We conclude this paper in Section 8.

2 BACKGROUND

2.1 Geographic Search Framework

We consider a typical GIS/LBS application scenario with three major entities - users, client and server. A server (e.g.,
AWS) is a cloud provider for computing or storage services; a client is a GIS/LBS provider (e.g., Yelp) that outsources its data (after encryption) to the server while keeping some local computation and storage capacity; users (e.g., mobile users) are consumers who make geographic search requests to the client to enjoy GIS/LBS services. We assume the client pre-computes encrypted indexes and uploads them to the server to assure search efficiency.

In real GIS/LBS systems, both SQL/relational and NoSQL/non-relational databases are widely adopted. To ensure compatibility, we consider secure geographic search scenarios with both types of databases. In SQL databases, data is organized in tables of columns (i.e., attributes) and rows (i.e., records). A record would be contributed by a user. NoSQL databases, on the other hand, store data as JSON documents which are composed of key-value pairs. For brevity, we use block to denote either a record or a JSON document since both of them can be indexed by a certain attribute. Each block is assigned a unique identifier, as shown in Fig. 2.

We logically divide a database into two pools, data and their indexes. Data (namely blocks) are indexed by their locations, and those sharing the same location are included under the same key. To avoid ambiguity, we call a key for indexing locations as index-key. For example, assume the index-key w1 in Fig. 2 is associated with a certain geographic area on Earth (e.g., New York City). When searching for w1, the data such as id1, id2 located in this area will be returned. In this work, we only focus on the location attribute (e.g., DGGS-compatible geographical cell codes) and how to retrieve an exact index-key over its encrypted form such that the queried encrypted blocks will be returned correctly, securely and efficiently.

2.1) to facilitate geographical search. The recursive partition means that the grid is hierarchical, so are the cell codes. A longer cell code means a more precise geographical location and its prefix represents a broader area around it. Different DGGS systems may have different resolutions based on how and the granularity of the partition.

Widely adopted DGGS techniques include Niemeyer’s Geohash [2], Google S2 [3] and Uber H3 [4], to name a few. They all decompose the globe into a hierarchy of regular geometric cells. For example, GeoHash partitions Earth into rectangle cells while Google S2 uses square cells and Uber H3 is based on hexagons. GeoHash simply follows the hierarchical partition where the grid is recursively divided into smaller rectangular ones. S2 adopts so-called “roads” (for logical parent-child relationship between nodes) inside a cell such that points geographically close to each other are still close in the database. When a point is accessed via its index, its neighbours can also be accessed. These “roads” are coded in the prefix of a cell code. A similar strategy is adopted by H3. Taking the Statue of Liberty (Fig. 1) for instance, GeoHash uses cell code “dr5r7” for the sub-area (the right sub-figure in Fig. 1) with a precision of 20km$^2$. To search the similar area, S2 uses “b0fd9d”, and it is 8a2a10 in H3. In GeoHash, the cell code of the highest resolution of 6 cm$^2$ in that area is “dr5r7kkkek9p” with 12 levels (i.e., 12 letters) of precision. While in S2, it is “b0fd9d0806a6787” for the highest resolution of 0.74cm$^2$ with a maximum of 16 levels. H3 uses “8a2a10fffffff” with a maximum of 15 levels for a precision of 8950cm$^2$. In essence, geographic search in DGGS is implemented through prefix search no matter what partitioning and cell coding methods are used.

2.3 Geographic Search Through Prefix Search

Consider an index-key w which is a cell code in a geographical database. Geographical search using another cell code w returns a match if and only if $w_p \subseteq w$, i.e., $w_p$ is the prefix of w. For example, when a tourist searches “best places to visit in New York”, it will return a list of specific places such as “Empire State Building” and “Statue of Liberty”. Since the query location “New York” (i.e., $w_p$) is broader than “Empire State Building” or “Statue of Liberty” (i.e., w that is used as an index-key in the geographical database), the former is a prefix of the latter. Evaluating whether a search query is a prefix of an index-key is straightforward and can be illustrated as follows.

![Figure 3: Prefix search for index-key “dr5r7kkkek9p”](image)

Assume an index-key “dr5r7kkkek9p”. Checking whether dr5r7 is its prefix is simply to compare each character from

This precision indicates the average hexagon area of H3 grids.
left to right in the order of \( d \), then \( r \), ..., and then \( 7 \), one by one, as shown in Fig 3. It shall be noted that in SGS where both search queries and index-keys (which are usually called “keywords”) are encrypted, the same prefix search process as in Fig. 3 is performed but over encrypted characters.

3 RELATED WORK

Secure Geographic Search (SGS). The notion of Secure Geographic Search (SGS) was coined in 2014 by Ghinita and Rughinis [18] for searching encrypted geographic data. This work exploits Hidden Vector Encryption (HVE) to hierarchically encode and then encrypt data and queries to vectors, each of which is of high dimension. Thereafter, variant solutions [17, 19–23] have been proposed with improved efficiency. For instance, ref. [19] is constructed on a domain-based binary tree, and [20] on a Huffman encoding tree, both for faster retrieving of HVE [24] instances. Ref. [17] leverages Shen-Shi-Waters (SSW) encryption [25] to check and verify inner products over vectors converted from data and queries. However, the intrinsic complexities of HVE and SSW hinder the run-time efficiency despite of the theoretical faster-than-linear [17] and logarithmic [18–20] retrieve complexities, respectively. For retrieve efficiency, ref. [21] introduces a multi-level index mechanism based on bloom filters. However, this work also inherits the security-efficiency trade-off with bloom filters and offers a weaker security when high efficiency is preferred. Ref. [22] exhibits the similar issue and provides at most an equivalence of 17-bit security though only lightweight primitives such as cryptographic hash functions (based on the structure of [21]) and bloom filters are used. Besides, both [21] and [22] require one extra interaction for a single query. The above solutions consider a single server and follow the single server-client framework. Ref. [23] adopts two non-colluding servers but overlooks the leakage risk by simply clustering points by distance. Among these solutions, only [21] and [22] consider compatibility with the DGGS technologies. However, none of the existing solutions simultaneously satisfies the requirements of near real-time search latency, data dynamics, and forward/backward privacy.

SSE and Dynamic SSE. The problem of SSE was first formulated by Song et al. [26]. Since then, extensive research has been focused on various aspects including query expressiveness [27, 28], search security with controlled leakage [29] and efficiency optimization [30, 31]. This family of SSE schemes allow leakage that is modeled under the notion of access pattern and search pattern, where the former reveals the actually retrieved items of a query, and the latter leaks previous queries related to the current one.

Stefanov et al. [32] first elaborated the notion of forward and backward privacy for so-called dynamic SSE, i.e., SSE that support data dynamics (i.e., data insertion and deletion). After that, a number of dynamic SSE schemes [10–13] were proposed but only supporting forward privacy, i.e., breaking the link between newly added data and previous queries. Zhang et. al. [33] further demonstrated a successful file-injection attack when forward privacy is not assured. The notion of backward privacy for dynamic SSE was first defined by Bost et al. [12], i.e., limiting what the server can learn between newly deleted data and queries afterwards. It regulates three different levels of leakages in the incremental (inclusive) order, from the least of Type-I to the most of Type-III. All these types allow the leakage of identifiers of documents that match the keyword \( w \) that is currently being searched and the time when they were inserted. Under this backward privacy framework, Bost et al. [12] and Ghareh Chamami et al. [14] proposed dynamic SSE constructions with all the three types of backward privacy, respectively. Sun et al. [34] proposed a Type-III backward privacy scheme. Recently, three Type-II schemes were proposed by Demertzis et al. [15, 35] and Sun et al. [36]. In addition, Hoang et al. [37] proposed a dynamic SSE framework particularly towards cloud data storage-as-a-service infrastructures. All these schemes focus on whole keyword search without supporting prefix search.

ORAM-TEE-based SSE. The leakage of access pattern has drawn increasing attention as discussed by Garg et. al [38]. Although this issue can be mitigated by Oblivious Random Access Machine (ORAM) [39] and its variants [40–42], deploying ORAM in the network environment, especially for searching data from remote cloud servers, incurs a significant bandwidth cost of at least poly-logarithmic complexity [43, 44]. To address this challenge, Hoang et. al. [45] combine ORAM with trusted execution environments (TEEs) to alleviate the bandwidth constraints. This approach leverages the trusted enclaves to store the oblivious structure and secret keys. Consequently, the decryption of the data during searches is executed within the enclave through I/O operations, rather than via communication with clients, reducing the bandwidth overhead by around \( 100 \times \) [45]. Another work [46] that adopts a similar methodology focuses on hiding the leakage caused by dynamic updates through utilizing the trusted components provided by TEE. Despite the great advancements, these works rely on a stronger security assumption on the TEE and only support search on the whole keywords.

Secure Substring Search (Prefix Search). Substring SSE was first studied by Shen et al. [47] in the context of static SSE. This construction is based on a suffix tree in which a large extent of the tree structure will be disclosed. Following this initial attempt, ref. [48] further reduces the storage cost of [47] and enhances its security. Leontiadis et. al. [49] then adopted Order-Preserving Encryption (OPE) for substring search. Subsequently, ref. [50] utilizes the SGX technology for substring search. The security of this approach hinges on the security of trusted hardware. Notably, all these constructions are for static SSE and leaks statistical information, e.g., the relation between suffix and prefix data. To support dynamic SSE, Moataz et. al. [51] utilize structure encryption and inner product predicate for substring search and updates.
While accommodating updates, this method also leaks statistical information, such as the frequency of the same letters. In contrast, without compromising statistical leakage, ref. [16] utilizes a hierarchical ORAM tree structure to support substring search. Also, oblivious suffixes are organized by this structure to simultaneously support data update operations. However, this update introduces additional heavy bandwidth overhead due to the ORAM environment. In addition, Mainardi et al. [52] exclusively supports offline model for secure substring search. Other than these works, to our best knowledge, the research on the problem of secure substring or prefix search for dynamic SSE is very limited.

### 4 PRELIMINARIES

In this section, we define the basic cryptographic primitives used in this work, including the syntax and security definitions of dynamic prefix symmetric searchable encryption (pSSE) and forward/backward privacy.

#### 4.1 Dynamic pSSE

Different from SSE, dynamic prefix symmetric searchable encryption (pSSE) extends the capabilities of SSE by enabling prefix search, alongside the inherent properties of SSE. Essentially, SSE can be viewed as a special case of pSSE where the prefix is the entire keyword. Similar to dynamic SSE, dynamic pSSE is pSSE that supports data dynamics such as insertion and deletion. A dynamic pSSE scheme \( \Sigma = (\text{Setup}, \text{Search}, \text{Update}) \) is comprised of one algorithm and two protocols:

- **Setup**: \( 1^k \rightarrow K, \sigma, EDB \): It takes as input the security parameter \( \lambda \) and outputs \( K, \sigma \) and \( EDB \), where \( K \) is a secret key, \( \sigma \) is the client’s local state, and \( EDB \) is the (empty) encrypted database that is to be sent to the server.

- **Search**: \( (K, \sigma, q; EDB) \rightarrow DB(w_p) \): It is a protocol for searching the database by the server. It gets inputs \( (K, \sigma, q) \) from a client and \( EDB \) (including index) from the server. The server outputs \( DB(w_p) \) in the form of a result list \( I_{w_p} \). The result would be empty if \( w_p \not\in \) \( w \) for any index-key \( w \). In this paper, we only consider search queries for a single prefix \( w_p \) of possibly multiple keywords (i.e., index-keys) \( w \).

- **Update**: \( (K, \sigma, op, in; EDB) \rightarrow K, \sigma, EDB \): It is a protocol for inserting an entry to or deleting an entry from the database. It takes as inputs \( EDB \) from a server and \( (K, \sigma, op, in) \) from a client. Operation \( op \) can be add or del and the input \( in \) is composed of a block \( B \) and its identifier \( id \). After the protocol, the input block \( B \) is added to or (logically) removed from \( EDB \) while its identifier \( id \) is added to or (logically) removed from its corresponding index. The protocol may modify \( K, \sigma \) and \( EDB \) as needed.

Above APIs follow the definition of [12] with some minor modifications. In this paper, we consider a database (SQL or NoSQL) composed of index and data, where the data is in the format of blocks. As discussed in Section 2.1, a block is either a record in SQL database or a JSON document in NoSQL database. In this work, \( EDB \) consists of two encrypted pools - the encrypted indexes and the encrypted blocks. We implicitly assume that after receiving the result identifiers \( DB(w_p) \), the client performs an additional round to retrieve the actual blocks. For brevity, we omit this step in our construction.

**Correctness**: A dynamic pSSE scheme \( \Sigma = (\text{Setup}, \text{Search}, \text{Update}) \) is correct if it always returns the correct results, i.e., indices, for each query. The definition aligns with the formal correctness definition of SSE in [29, 53].

**Security**: The security of pSSE is captured by using a real-world versus ideal-world experimentation [29, 32, 53]. An intuition is that an adversary can not distinguish between the experiments REAL and IDEAL. The security model is parameterized by a leakage function \( L = \{ L_{\text{Srch}}, L_{\text{Up}} \} \) which captures the information learned by an adversarial server. \( L_{\text{Srch}} \) and \( L_{\text{Up}} \) correspond to leakage during setup, search and updates in respective. Informally, a secure pSSE scheme with leakage \( L \) should reveal nothing about the database \( DB \) except for this explicit leakage.

**Definition 4.1 (Adaptive Security of Dynamic pSSE)**. A dynamic pSSE scheme \( \Sigma \) is \( L \)-adaptively secure (with respect to leakage function \( L \)) iff for all PPT adversary \( A \) that makes polynomial number of queries \( q \), there exists a stateful PPT simulator \( \text{Sim} \) such that

\[
|Pr[\text{REAL}\Sigma_A(\lambda, q) = 1] – Pr[\text{IDEAL}\Sigma_{\text{Sim}}(\lambda, q) = 1]| \leq \text{negl}(\lambda),
\]

where \( \text{REAL}\Sigma_A(\lambda, q) \) and \( \text{IDEAL}\Sigma_{\text{Sim}}(\lambda, q) \) are defined as follows:

- **REAL\Sigma_A(\lambda, q)**: The adversary \( A \) initially chooses a database \( DB \) and gets back \( EDB \) by calling Setup(\( 1^k \)). Then \( A \) is allowed to adaptively perform search (resp. update) queries with input \( q \) (resp. input \( (op, in) \)) and receives transcript generated by calling Search(\( K, \sigma, q; EDB \)) (resp. Update(\( K, \sigma, op, in; EDB \))). Given these real transcripts, the adversary \( A \) finally outputs a bit \( b \).

- **IDEAL\Sigma_{\text{Sim}}(\lambda, q)**: The adversary \( A \) initially selects a database \( DB \) and gets back \( EDB \) generated by the simulator \( S(L_{\text{Srch}}(DB)) \). Then \( A \) is allowed to adaptively perform search (resp. update) queries with input \( q \) (resp. input \( (op, in) \)) and receives transcript generated by running \( S(L_{\text{Up}}(K, \sigma, q, EDB)) \) (resp. \( S(L_{\text{Up}}(K, \sigma, op, in; EDB)) \)). Given these simulated transcripts, the adversary \( A \) finally outputs a bit \( b \).

### 4.2 Forward and Backward Privacy of pSSE

Forward and backward privacy are two important properties of dynamic SSE that restrict what information is leaked
by the Update query, so are they in dynamic pSSE. The definition of forward and backward privacy of SSE was first proposed in [32] and then formulated by Bost et al. [11, 12]. In their works, the definition only considers queries on keyword/document pairs, i.e., searching to find out whether a keyword belongs to a certain document. In this paper, we first formulate the definition of forward and backward privacy for a SQL (or NoSQL) database structures of index and data. A search is to find out the indices that are related to an index-key w which is subject to a certain attribute. In brief, forward privacy ensures previous search queries do not reveal any information about the retrieved blocks to be updated. It hides whether an addition is about a new block or one that has been previously searched for. Backward privacy limits the information that the server can learn about blocks that are added previously but deleted later during searches related to them. It guarantees that the adversary cannot obtain extra information about the deleted blocks. For space limit, we include the full definitions of forward/backward privacy for keyword/block pairs in Appendix A, following similar definitions in [12].

4.3 Pseudorandom Function

Let $G : \mathcal{K} \times X \rightarrow \mathcal{Y}$ be a function defined from $X$ to $\mathcal{Y}$. $G$ is a secure pseudorandom function (PRF) if for all PPT adversaries $A$, its maximum advantage as $\text{Adv}^{\text{PRF}}_{A,G}(\lambda) = |Pr[A^{G(k)}(1^\lambda) = 1] − Pr[A^{F}(1^\lambda) = 1]|$ is negligible in $\lambda$, where $F$ is a random function from $X$ to $\mathcal{Y}$, and $k \xleftarrow{} \mathcal{K}$.

5 SYMMETRIC PREFIX PREDICATE ENCRYPTION

In this section, we present a novel cryptographic primitive, symmetric prefix predicate encryption (SP²E). We begin with the building block $f$-bit bounded symmetric encryption (SE-f) followed by the syntax, security, and concrete construction of SP²E.

5.1 $f$-bit Bounded Symmetric Encryption

Let $\lambda$ be a security parameter and $G$ denote a pseudorandom function with an output length $l(\lambda)$. An $f$-bit bounded symmetric encryption scheme SE-f with message space $\mathcal{M}$ can be described as a 3-tuple $(SE-f.Gen, SE-f.Enc, SE-f.Dec)$. While the proposed scheme is derived from standard symmetric encryption (SE), the difference is in SE-f, only a $f$-bit interval is decrypted instead of the whole message. We formalize the definition of SE-f as follows:

SE-f.Gen($1^\lambda$) → $k_1, k_2$: On input a security parameter $\lambda$, it outputs two random secret keys $k_1, k_2$ uniformly sampled from key space $\mathcal{K}$.

SE-f.Enc($k_1, k_2, \text{sub}(\cdot)$): Given $k_1, k_2$, it encrypts a $f$-bit message $\text{sub}(m)$.

SE-f.Dec($k_1, k_2, \text{sub}(m)$): It decrypts a $f$-bit message $\text{sub}(m)$.

5.2 Syntax

Syntax of SE-f:

- $\text{Enc}(k_1, m)$:
  - $m \in \mathcal{M}$:
  - $ct \leftarrow \text{SE-f.Enc}(k, m)$
  - Return $ct$.

Figure 4: IND-CPA security of SE-f

SE-f.Enc($k_1, m$) → $ct$: On input a secret key $k_1$ and message $m \in \mathcal{M}$, it computes and outputs ciphertext $ct$ as:

$$ct = G(k_1, m) \oplus m.$$ (2)

where $G(k_1, m)$ is a PRF output that XOR the message $m$. Note that $|m| = |G(k_1, m)|$, $ct \in \{0, 1\}^{l(\lambda)}$.

SE-f.Dec($k_1, k_2, ct$) → $m'$: On input $k_1, k_2$ and $ct$, it decrypts some chosen consecutive $f$ bits among a length $l(\lambda)$ of the message $m$ with the bit position starting from $pos_1$ to $pos_2$, where $0 \leq pos_1 < pos_2 \leq l(\lambda)$, $pos_2 - pos_1 = f$. The algorithm computes and outputs $m'$ as:

$$m' = ct \oplus$$

$$G(k_2, m).\text{sub}(\text{pos}_1, \text{pos}_2) \parallel G(k_1, m).\text{sub}(\text{pos}_1, \text{pos}_2)$$

(3)

where the method $ct \oplus \text{sub}(\text{StartInx}, \text{EndInx})$ extracts the bits from the StartInx-th position to the EndInx-th position in a “left-closure right-open” manner, e.g. for $ct = 100100$, $x.\text{sub}(2, 4) = 01$.

Correctness:

$$m'.\text{sub}(\text{pos}_1, \text{pos}_2)$$

$$= \text{SE-f.Dec}(k_1, k_2, \text{SE-f.Enc}(k_1, m)) \text{.sub}(\text{pos}_1, \text{pos}_2)$$

$$= (ct \oplus G(k_1, m)) \text{.sub}(\text{pos}_1, \text{pos}_2)$$

(4)

$$= G(k_1, m) \oplus m \oplus G(k_1, m) \text{.sub}(\text{pos}_1, \text{pos}_2)$$

$$= m.\text{sub}(\text{pos}_1, \text{pos}_2).$$

Security: The security of SE-f is defined by an IND-CPA experiment presented in Fig. 4. The security definition of SE-f is similar to the standard indistinguishability definition of symmetric encryption (SE) except for decryption, where a segment of the message is decrypted in SE-f rather than a complete message.

Definition 5.1 (Semantic Security of SE-f). A symmetric encryption scheme $SE-f = (SE-f.Gen, SE-f.Enc, SE-f.Dec)$ is IND-CPA secure if for any security parameter $\lambda \in \mathbb{N}$ and PPT (probabilistic polynomial time) adversary $A$, the advantage of $A$ is negligible in $\lambda$. That is,

$$\text{Adv}^{\text{IND-CPA}}_{A,SE-f}(\lambda) = \left| Pr[A^{\text{SE-f.Enc}}(\lambda) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda).$$ (5)
5.2 Syntax and Security of SP$^2$E

A symmetric prefix predicate encryption scheme SP$^2$E is defined over a keyword space $\Lambda'$ where each character of a keyword belongs to the alphabet $\Lambda$, and $t$ is the maximum length of a keyword. SP$^2$E consists of a quintuple of PPT algorithms ($\text{KeyGen}, \text{PreEnc}, \text{Enc}, \text{TKGen}, \text{PrefDec}$). The preliminary encryption algorithm PreEnc encodes the keyword as a fixed-length hash value which serves as a message, an input of the encryption algorithm Enc, to compute the ciphertext $ct$. The algorithm TKGen outputs a token, namely the encrypted prefix. The prefix decryption algorithm PrefDec identifies if the ciphertext $ct$ contains a specific prefix without revealing the keyword nor message. The definition is formalized as follows:

$\text{KeyGen}(1^\lambda, t) \rightarrow SK, MSK$: On input a security parameter $\lambda$ and a maximum length $t$ of a keyword, it generates a secret key $SK$ for encryption and a master secret key $MSK$ that is used as seed.

$\text{PreEnc}(MSK, seq, w) \rightarrow m$: On input $MSK$, a string keyword $w$ and a number $seq$ that indicates an identical sequence of $w$ among keywords, the algorithm outputs a fixed-length hash value $m$.

$\text{Enc}(SK, m) \rightarrow ct, \delta$: On input $SK$ and $m$, the algorithm outputs a ciphertext $ct$ and a local parameter $\delta$ that is stored in the local state.

$\text{TKGen}(SK, MSK, \delta, seq, w_p) \rightarrow k'$: On input secret keys $SK, MSK$, a local parameter $\delta$, a number $seq$ and a prefix $w_p$, it outputs a token $k'$.

$\text{PrefDec}(k', ct) \rightarrow \perp$ or $w_p \sqsubseteq w$: On input a token $k'$ and ciphertext $ct$, it outputs $\perp$ or $w_p \sqsubseteq w$.

**Correctness:** For security parameter $\lambda$, integer $t \in \mathbb{N}^*$, $(SK, MSK) \leftarrow \text{KeyGen}(1^\lambda, t)$ and a keyword $w \in \Lambda'$, SP$^2$E is correct if

$$
Pr \left[ \text{PrefDec}(k', ct) \rightarrow \perp : w_p \sqsubseteq w \right] = 1.
$$

**Security:** The security of SP$^2$E is defined by two experiments, an IND-CPA experiment $\text{Exp}_{\Lambda, \text{SP}^2\text{E}, \text{Enc}}^{\text{IND-CPA}}(\lambda)$ for encryption and an IND-f-CPA experiment $\text{Exp}_{\Lambda, \text{SP}^2\text{E}, \text{TKGen}}^{\text{IND-f-CPA}}(\lambda)$ for token generation. The security definition of IND-f-CPA is similar to the standard IND-CPA indistinguishability definition except for the specific $f$ bits. Concretely speaking, a PPT adversary has the ability to access and query both oracles $\Omega_{\text{Enc}}(w)$ and $\Omega_{\text{TKGen}}(w_p)$ with unlimited times after the adversary submits an input $w$ (or $w_p$). The details of experiments are presented in Fig. 10 in Appendix B. The formal definition of encryption and token security of SP$^2$E is presented below.

**Definition 5.2 (Encryption Security of SP$^2$E).** A symmetric prefix predicate encryption scheme SP$^2$E is IND-CPA encryption secure, if for all PPT adversary $\mathcal{A}$, the advantage pf $\mathcal{A}$ is negligible in $\lambda$. That is:

$$
\text{Adv}_{\mathcal{A}, \text{SP}^2\text{E}, \text{Enc}}^{\text{IND-CPA}}(\lambda) = \left| Pr[\text{Exp}_{\mathcal{A}, \text{SP}^2\text{E}, \text{Enc}}^{\text{IND-CPA}}(\lambda) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda).
$$

**Definition 5.3 (Token Security of SP$^2$E).** A symmetric prefix predicate encryption scheme SP$^2$E is IND-f-CPA token secure, if for all PPT adversary $\mathcal{A}$, the advantage pf $\mathcal{A}$ is negligible in $\lambda$. That is:

$$
\text{Adv}_{\mathcal{A}, \text{SP}^2\text{E}, \text{TKGen}}^{\text{IND-f-CPA}}(\lambda) = \left| Pr[\text{Exp}_{\mathcal{A}, \text{SP}^2\text{E}, \text{TKGen}}^{\text{IND-f-CPA}}(\lambda) = 1] - \frac{1}{2} \right| \leq \text{negl}(\lambda).
$$

According to the encryption and token security defined in Def. 5.2& 5.3, we further define the adaptive security for SP$^2$E as follows:

**Adaptive Security:** We define the adaptive security for SP$^2$E scheme since a PPT adversary $\mathcal{A}$ is allowed to submit the chosen challenged keyword strings $w_0, w_1$ (or prefixes $w_{p_0}, w_{p_1}$) adaptively at any point in time while the security is guaranteed.

**Definition 5.4 (Adaptive Security of SP$^2$E).** A symmetric prefix predicate encryption scheme SP$^2$E is IND-f-CPA secure, if for all PPT adversary $\mathcal{A}$, it has at most a negligible advantage

$$
\text{Adv}_{\mathcal{A}, \text{SP}^2\text{E}}^{\text{IND-f-CPA}}(\lambda) = \max \left( \left| Pr[\text{Exp}_{\mathcal{A}, \text{SP}^2\text{E}, \text{Enc}}^{\text{IND-CPA}}(\lambda) = 1] - \frac{1}{2} \right|, \left| Pr[\text{Exp}_{\mathcal{A}, \text{SP}^2\text{E}, \text{TKGen}}^{\text{IND-f-CPA}}(\lambda) = 1] - \frac{1}{2} \right| \right) \leq \text{negl}(\lambda).
$$

5.3 Bit-wise Prefix Recognition SP$^2$E from PRF

In this part, we present the details of SP$^2$E based on a pseudorandom function (PRF) $G : \mathcal{K} \times X \rightarrow \mathcal{Y}$, a cryptographic hash function $H : \mathcal{K} \times \mathbb{N}^* \rightarrow \mathcal{K}$ and an $f$-bit bounded symmetric encryption scheme $\text{SE}_f = (\text{SE}_f, \text{Gen}, \text{SE}_f, \text{Enc}, \text{SE}_f, \text{Dec})$. Specifically, the proposed scheme SP$^2$E = $(\text{KeyGen}, \text{PreEnc}, \text{Enc}, \text{TKGen}, \text{PrefDec})$ is constructed as follows:

$\text{KeyGen}(1^\lambda, t) \rightarrow SK, MSK$: On input a security parameter $\lambda$ and a parameter $t$ that denotes the maximum length of a keyword, it uniformly samples secret keys $sk_1, sk_2, sk_{k_1}, ..., sk_{k_{|\Lambda|}}$ at random from the key space $\mathcal{K}$. Each secret key $sk_{c_i}$ uniquely associates with a character $c_i$ in the textual alphabet $\Lambda$, for $1 \leq i \leq |\Lambda|$, and $|\Lambda|$ is the total number of distinct characters. A keyword is a string and all of its characters are from $\Lambda$. This algorithm specifies a parameter $f$ for indicating the number of “certain bits” that are indicators of a queried result. $f$ is subject
to $0 < f \leq \frac{|\lambda|}{2^l}$. For simplicity, we denote the outputs with $SK = \{sk_1, sk_2\}, MSK = \{sk_c, \ldots, sk_c|\}, f$.

PreEnc$(MSK, seq, w) \rightarrow m$: It takes as input $MSK$, a keyword $w$ and a number $seq$ that is the sequence of this $w$ among (distinct) keywords. It outputs the fixed-length hash value $m$ computed by following steps:

1. Compute the length of a keyword $w$, $|w|$, i.e., the total number of its characters.
2. Pick up the associated key $sk_w$ from $MSK$ for each character of $w$ in sequence. For example, if the $i$-th character is a textual letter '$a'$, then its associated key is $sk_{a'}$, namely $sk_{a'}$ refers to $sk_{a'}$ in this case.

3. Output $m$ as

$$m = \bigoplus_{i=1}^{|w|} \left( (0^{i-1}) \parallel H(sk_{w_i} \parallel seq \parallel i) \right) . \text{sub}(0, l(\lambda)). \tag{10}$$

$\text{Enc}(SK, m) \rightarrow ct, \delta$: On input $SK$ and $m$, by using the secret key $sk_1$ in $SK$, it computes the ciphertext as

$$ct = \text{SE-f.Enc}(sk_1, m) = m \oplus G(sk_1, m), \tag{11}$$

When the encryption is invoked, $\delta = G(sk_1, m)$ must be recorded in the local state. Note that $\delta$ is not public. Instead, it is secretly held by the entity executing this algorithm, usually the client.

$\text{TKGen}(SK, MSK, \delta, seq, w_p) \rightarrow k'$: On input $SK, MSK$, a local parameter $\delta$, an identifier $seq$ and a queried prefix $w_p$, it outputs a token $k'$ that will be used to evaluate if the message of a ciphertext $ct$ contains prefix $w_p$. With both secret keys $sk_1$ and $sk_2$ in $SK$, the algorithm computes $k'$ as:

1. Calculate the length of the prefix $w_p$, $|w_p|$.
2. Pick up the associated key $sk_{w_p}$ from $MSK$ in sequence for each character of $w_p$.
3. Set $pos_1 = (|w_p| - 1) \cdot f$, $pos_2 = |w_p| \cdot f$, and we have $pos_2 - pos_1 = f$. The parameters $pos_1$ and $pos_2$ are the respective start and end bit-position of an $f$-bit segment.

4. Output $k'$ as

$$k' = \bigoplus_{i=1}^{|w_p|} \left( (0^{i-1}) \parallel H(sk_{w_{p_i}} \parallel seq \parallel i) \right) . \text{sub}(0, l(\lambda)) \oplus G(sk_{seq}, \text{sub}(0, pos_1)) \parallel G(sk_{seq}, \text{sub}(pos_2, l(\lambda))). \tag{12}$$

$\text{PrefDec}(k', ct) \rightarrow \perp$ or $w_p \subseteq w$ \footnote{This decryption here is logically equivalent to the decryption $\text{SE-f.Dec}(sk_1, sk_2, ct) \rightarrow m$ in Sec. 5.1. The message carried by a token $k'$ corresponds to the mentioned $m$, that is $\bigoplus_{i=1}^{|w_p|} \left( (0^{i-1}) \parallel H(sk_{w_{p_i}} \parallel seq \parallel i) \right) . \text{sub}(0, l(\lambda)).$}:

On input a token $k'$ and a ciphertext $ct$, it calculates that

$$r' = k' \oplus ct, \text{ and } 0^f \overset{?}{=} r'. \text{sub}(pos_1, pos_2) \tag{13}$$

The algorithm evaluates whether there are consecutive $f$ bits valued with zero located in the segment $[pos_1, pos_2)$ of the result $r'$. We denote these $f$ bits with $0'$. In brief, $0'$ is an indicator for a keyword. By verifying $0' \overset{?}{=} r'. \text{sub}(pos_1, pos_2)$, the algorithm PrefDec can identify whether the message $m$ contains the prefix $w_p$. The algorithm outputs $\perp$ if $w_p$ is not a prefix of $m$. Otherwise, it outputs $w_p \subseteq w$.\footnote{In Fig. 5 (c)-(d) we omit “sub” for simplicity, i.e., $\delta(pos_1, pos_2) = \delta.\text{sub}(pos_1, pos_2); \eta(0, pos_1) = \eta.\text{sub}(0, pos_1); etc.}\

Correctness: The correctness of $\text{SP}^2E$ is derived from that of $\text{PRF} G$ and the symmetric encryption $\text{SE-f}$. That is

"$w_p \subseteq w$: $w_p$ is a prefix of $w$" or
"$w_p \nsubseteq w$: $w_p$ is not a prefix of $w".$

In the first case, the bits valued with zero (i.e., $0'$) in the output of $\text{SP}^2E$ decryption should be revealed such that a keyword $w$ having the queried prefix $w_p$ can be identified correctly. To validate this, we expand the output of PrefDec, namely $k' \oplus ct$. As shown in Fig. 11. If $w_p$ is a prefix of the keyword, the $f$ bits between $pos_1$ and $pos_2$ should all be zero. As for the latter case of $w_p \nsubseteq w$, the output of $\text{SP}^2E$ decryption should be a completely indistinguishable value without any consecutive bits valued at zero visible. We validate this with more details and examples in Fig. 12 in Appendix D.

Example Illustration: We illustrate each algorithm of $\text{SP}^2E$ with the example of the Statue of Liberty. Consider a DGGs code $w = \text{“dr5r77”}$ and a queried prefix $w_p = \text{“dr5r7”}$. With all the required parameters of KeyGen, PreEnc computes the fixed-length hash value $m$ according to the inputs – a keyword $w$ and its sequence $seq$. A fixed-length string is obtained by XORing several hash substrings that correspond to the characters of $w$ (i.e., “dr5r77”), as depicted by a grey rectangular box in Fig. 5 (a). The encryption algorithm $\text{Enc}$ generates the ciphertext $ct$ by XORing $m$ with a mask $\delta$ which is generated uniformly at random and stored locally, as shown in Fig. 5 (b).

TKGen outputs an encrypted search token $k'$ for the query prefix $w_p$ (i.e., “dr5r7’”). As shown in Fig. 5 (c), $k'$ is computed by XORing the hash substrings with a pseudorandom string which is the concatenation of pseudo-random value $\eta$ and a local secret $\delta$. The hash substrings are associated sequentially with the characters of $w_p$. If SHA-256 is chosen for producing the hash substrings, for example, the output length $l(\lambda) = 256$.

With the input token $k'$ and a ciphertext $ct$, the prefix decryption algorithm PrefDec examines whether the $f$ bits in the output (i.e., $k' \oplus ct$) at positions $[pos_1, pos_2)$ are all zeros. All zeros mean a keyword $w$ matches a given prefix $w_p$. As shown in Fig. 5 (d), bits $\{4f, 5f\}$ of $k' \oplus ct$ are all zeros, which occurs due to the two same values $\delta.\text{sub}(pos_1, pos_2)$ presented in $k'$ and $ct$, respectively.

5420 33rd USENIX Security Symposium
USENIX Association
5.4 Security Analysis

In this part, we show the proposed SP²E of being proven IND-f-CPA secure in the random oracle model, under the security of cryptographic hash function G, PRF F and f-bit bounded symmetric encryption SE-f.

**Theorem 5.5 (Adaptive Security).** If \( G: \mathcal{K} \times X \rightarrow \mathcal{Y} \) is a secure PRF, \( H \) is a secure cryptographic hash function, \( H \) is modeled with a random oracle, and SE-f is an IND-CPA secure symmetric encryption, then the proposed SP²E is IND-f-CPA secure in the random oracle model.

**Proof Sketch:** We prove the security of our SP²E scheme through two parts, encryption security defined in Def. 5.2 and token security defined in Def. 5.3, for which we give concrete proofs in respective via a series of games. Based on both encryption and token security, the adaptive security of SP²E in Def. 5.4 holds. To complete the proof, we consider a sequence of hybrid games that differ from each other in the challenge pre-encryption ciphertext and challenge token. The first game in the sequence corresponds to the real IND-f-CPA game for SP²E while the last hybrid one is the standard IND-CPA game for the f-bit bounded symmetric encryption SE-f. The overall advantage of \( \mathcal{A} \) changes only by a negligible amount between each successive hybrid game, and is bounded by the advantage gained in IND-f-CPA game for token generation, which is negligible, thereby establishing Thm 5.5. Due to the limited space, the complete proof is provided in the extended version of this paper, which will be available on Arxiv.

6 GridSE: Dynamic pSSE for Secure Geographic Search from SP²E

In this section, we present GridSE, a dynamic prefix symmetric searchable encryption (pSSE) based on symmetric prefix predicate encryption (SP²E) for secure geographic search.

6.1 Construction Details

GridSE follows the standard APIs of a standard dynamic pSSE scheme as discussed in Sec 4.1. By adapting the definitions to a generic SQL or NoSQL database, we implement a practical dynamic pSSE system for fast secure geographic search over DGGS cells. Recall that the database is typically organized by blocks, i.e., files/records. To support forward and backward privacy, GridSE encrypts triplets of \((w, id, op)\) into two ciphertext components which are organized as key-value pairs in a dictionary. The key stored in the dictionary is the ciphertext of index-key \( w \), namely the address which indicates whether \( w \) is queried by a specific prefix during searches. The value in the dictionary is a list associated with \( w \), consisting of one or multiple encrypted blocks \((id, op)\).

The encryption of \((id, op)\) is based on a mask generated by PRF and encapsulates an update counter \( upd_{cnt} \) that stores the update times occurred on blocks under index-key \( w \) as widely adopted for dynamic SSE. Note that data operations are indicated using \( op = add/del \), and any deletion operation is logically recorded instead of executing a physical deletion action.

However, a ciphertext of \((id, op)\) generated only based on \( upd_{cnt} \) is not enough to achieve forward and backward privacy for dynamic pSSE. The blocks (i.e., \((id, op)\)) associated with different index-keys but with the same \( upd_{cnt} \) will produce the same masks. To solve this issue, we introduce an additional parameter, the sequence of index-keys \( seq \) that helps to “tag” those blocks. In this way, each block is encrypted based on a unique mask and every update operation accessing the encrypted blocks in GridSE appears randomly to the server. Hence, the server cannot distinguish between insertion and deletion, which achieves backward privacy. Since each block is a separate triplet, the server cannot distinguish a new keyword in a newly added block from a keyword contained in the previously searched block, which achieves forward privacy.

To complete a search, after receiving the index-keys for a prefix, the client can generate all masks for correctly decrypting each block. Notice that each index-key encrypted by SP²E corresponds to a DGGS cell code and can iden-
identify the queried cells for the given prefix by the function \( SP^2E.\text{PrefDec} \). GridSE hence enables efficient search while supporting forward and backward privacy. Detailed construction of GridSE is as follows:

**Setup** (1) \( \rightarrow K_E, \sigma, E.BD : \) On input a security parameter \( \lambda \), the client generates a local state \( \sigma \) and an empty encrypted database \( E.BD \). It obtains secret keys from sampling \( K \sim \{0, 1\}^\lambda \) and secret keys \( SK, MSK \) by invoking the Setup algorithm of \( SP^2E \). Next, the client initiates an empty list \( \delta \) and two maps \( (InxDict, \text{UpdtCnt}) \). \( \delta \) and \( \text{UpdtCnt} \) are locally maintained while \( InxDict \) is sent to the server. At last, client outputs \( K_E = (K, SK, MSK), \sigma = \{\text{UpdtCnt}, \delta\} \) and \( E.BD \).

**Update** \( K_E, \sigma, op = add/del, in = (w, id); E.BD \) \( \rightarrow K_E, \sigma, E.BD : \) In the update procedure, the client receives an index-key \( w \), an identifier \( id \) and operation \( op = add/del \). An input \( \text{(add, w, id)} \) means “add a block id in \( E.BD \) and an entry in \( InxDict \) for this block that has an index-key \( w \).” The client has access to the keys \( K_E \), the local state \( \text{UpdtCnt} \) and \( \delta \), where \( \delta \) keeps the parameters \( \delta \) that is produced once invoking \( SP^2E.\text{Enc} \), and \( \text{UpdtCnt} \) stores for each distinct index-key \( w \) a counter that denotes how many updates have taken place on blocks in relation to \( w \). First, the client checks whether \( \text{UpdtCnt}[w] \) has been initialized or not. If not, the client sets the counter value of \( w \) to 0 (lines 1-4). Here, \( \text{UpdtCnt} \) is the size of current sequence of index-keys which indicates how many index-keys are currently in the dictionary. It increases by 1 once a new distinct index-key is produced with updates. Next, the client produces a key-value pair \( (addr, \delta) \) using the algorithm \( \text{SP}^2E.\text{Enc} \) by encrypting a given index-key and its current sequence. \( \delta \) is kept local. Notice that if a new but repeated indexed-key \( w \) is associated with an update after it has been counted in \( \text{UpdtCnt} \), the newly produced address \( addr \) associated with this \( w \) will be the same as the previous one due to the same sequence and keys as the input (lines 5-7). The client then runs the PRF \( G \) with key \( K \) and computes \( G(K, (\text{seq}, \text{UpdtCnt}[w])) \) that is XORed with the message \( (\text{id} \mid op) \), and this result becomes an encrypted value \( val \) (line 8). The pair \( (addr, val) \) is sent to the server which stores them by appending \( val \) into \( \text{InxDict}[addr] \) or a newly produced \( \text{InxDict}[addr] \) (lines 9-13).

**Search** \( K_E, (\sigma, w_p); E.BD \) \( \rightarrow w_p : \) While searching for the index-keys with a certain prefix \( w_p \), the client first generates a list of tokens \( T.List \). This token generation is submitted to the respective \( \text{SP}^2E \) layer, namely \( \text{SP}^2E.\text{TKGen} \). \( T.List \) is then sent to the server (lines 1-6). Recall that the corresponding entries to this token are stored in \( \text{InxDict} \) on the server, which enables the server to evaluate these entries with the \( \text{SP}^2E \) function \( \text{SP}^2E.\text{PrefDec} \). If \( \text{SP}^2E.\text{PrefDec} \) indicates \( \text{"w_p is a prefix for the current index-key w"} \), the server obtains the associated key-value pairs of \( w \) from the index and puts the value and its sequence into \( R_{w_p} \). Specifically, the server retrieves the key-value pairs matching the token \( T.List \) in \( \text{InxDict} \) and then sends them back to the client (lines 7-17). Then, the client decrypts the returned result \( R_{w_p} \) which are composed of pseudorandom values. These values generated via PRF \( G \) during the previous updates for \( w \) are recovered to the original plaintext since \( G \) is a deterministic function. Concretely, the client decrypts the received dictionary \( R_{w_p} \) by calculating the PRF values \( G(K, (\text{seq}, i)) \) and XORing them with each element of \( \text{valList} \) in sequence for \( i = 1, \ldots, \text{valList.size} \), where \( \text{seq} \) is the sequence of the key-value pair in which the current \( \text{valList} \) locates in \( \text{InxDict} \) (lines 18-25). In the end, the client returns a search result \( I_{w_p} \) containing all requested block identifiers which correspond to the index-keys of a prefix \( w_p \).

**Theorem 6.1.** Assuming \( F \) is a secure PRF and \( \text{SP}^2E \) is IND-f-CPA secure, GridSE is an adaptively-secure dynamic pSSE scheme of forward and Type-II backward privacy with \( \mathcal{L}^{\text{Upr}}(op, w, id) = \bot \) and \( \mathcal{L}^{\text{Srch}}(w_p) = (\mathcal{DB}(w_p). \text{Update}(w_p)) \).

**Proof Sketch:** We prove the adaptive security of GridSE by constructing a sequence of games and show that the advantage of a PPT adversary against our protocol is negligible. Intuitively, the transcript \( (addr, val) \) sent to the server is indistinguishable from uniformly random samples, which is guaranteed by the construction. Notice that the value \( val \) is computed by XORing the block/operation tuple \( (id \mid op) \) with \( G(K, (\text{seq}, \text{UpdtCnt}[w])) \) generated by a pseudorandom function \( G \). In this way, the server cannot distinguish whether an

---

**Algorithm 1:** GridSE \( \Sigma : \) Setup\( (1^\lambda, DB) \)

1. \( \{EDB, \sigma\} \leftarrow \Sigma.\text{Setup}(1^\lambda) \)
2. \( K \sim \{0, 1\}^\lambda, SK, MSK \leftarrow \text{SP}^2E.\text{Setup}(1^\lambda) \)
3. \( \delta \leftarrow \text{empty linked list} \)
4. \( \text{UpdtCnt}.\text{InxDict} \leftarrow \text{empty linked map} \)
5. \( \sigma \leftarrow \text{UpdtCnt}.\delta \)
6. Return \( (K, SK, MSK), (\text{UpdtCnt}, \delta), E.BD \)

**Algorithm 2:** GridSE \( \Sigma : \) Update \( (K_E, \sigma, op = add/del, in = (w, id); E.BD) \)

**Client:**

1. if \( \text{UpdtCnt}[w] \) is NULL then
2. \( \text{UpdtCnt}[w] = 0 \)
3. end if
4. \( \text{UpdtCnt}[w]++ \)
5. \( \text{seq} = \text{UpdtCnt}.\text{getMapSize}() \)
6. \( \text{addr}, \delta = \text{SP}^2E.\text{Enc}(SK, \text{SP}^2E.\text{PreEnc}(MSK, \text{seq}, w)) \)
7. \( \sigma \leftarrow \delta \)
8. \( \text{val} = (\text{id} \mid \text{op}) \oplus G(K, (\text{seq}, \text{UpdtCnt}[w])) \)
9. Returns \( (\text{addr}, \text{val}) \)

**Server:**

10. if \( \text{InxDict}.\text{containsKey}(\text{addr}) \) then
11. \( \text{Set InxDict}[\text{addr}] \cup \text{val} \)
12. else
13. \( \text{Set InxDict}[\text{addr}] = \text{val} \)
13. end if
Algorithm 3: GridSE \( \Sigma : \text{Search} \)

\( (K_{c}, \sigma, w_{p}, EDB) \)

Client:
1. \( TList \leftarrow \{ \} \)
2. for \( i = 1 \) to \( \text{UpdtCnt.size} \) do
3. \( T_{i} = SP^{2}E.TKGen(SK, MSK, \delta_{i}, i, w_{p}) \)
4. \( TList = TList \cup \{ T_{i} \} \)
5. end for
6. Return \( TList \)

Server:
7. \( R_{w_{p}} \leftarrow \text{empty linked map} \)
8. if \( \text{InxDict.size} \neq TList.size \) then
9. Abort
10. end if
11. for \( i = 1 \) to \( \text{InxDict.size} \) do
12. \( (addr, \text{valList}) = \text{InxDict}.\text{getEntry}(i) \)
13. if \( SP^{2}E.\text{PrefDec}(TList[i], addr) == 0 \) then
14. Set \( R_{w_{p}}[i] = \text{valList} \)
15. end if
16. end for
17. Return \( R_{w_{p}} \)

Client:
18. \( I_{w_{p}} \leftarrow \{ \} \)
19. while \( R_{w_{p}}.\text{hasNext}() \) do
20. \( (seq, \text{valList}) = R_{w_{p}}.\text{getNext}() \)
21. for \( i = 1 \) to \( \text{valList.size} \) do
22. \( (id||op) = \text{valList}[i] \oplus G(K, (seq, i)) \)
23. end for
24. \( I_{w_{p}} = I_{w_{p}} \cup \{id||op\} \)
25. end while

A search comprises a sequence of operations such as generating tokens in the client, sweeping index-keys in the server and decrypting in the client. Particularly, it requires \( \delta_{w_{p}} \) evaluations of \( \text{InxDict} \) in the server, and \( \delta_{w_{p}} SP^{2}E.TKGen \) invocations and \( \delta_{w_{p}} \) PRF instances for decryption in the client. Recall that a \( SP^{2}E.TKGen \) invocation introduces a PRF and several hash instances while a \( SP^{2}E.\text{PrefDec} \) invocation requires a single XOR operation. Both of these invocations are all evaluated with constant. Thus, the overall complexity of a search operation is \( O(\delta_{w_{p}} + \delta_{w_{p}}) \). Regarding the communication cost of a search, it is \( O(\delta_{w_{p}}) \) as the returned result size in a list format is asymptotic \( \delta_{w_{p}} \). On the other hand, an update requires \( O(1) \) for the communication overhead due to that the client only submits a single entry to the server. After \( N \) updates have taken place, regarding the client keeps a local list \( \delta \) of size \( \delta_{w_{p}} \), the local storage is merely \( O(\delta_{w_{p}}) \). Finally, GridSE takes an extra roundtrip to fetch the block identifiers for \( w_{p} \) if the actual blocks are necessary to be retrieved.

Recall that an index-key is associated exclusively with a DGGS cell. The maximum value of \( \delta_{w_{p}} \) does not exceed the number of DGGS cells, which is a constant for a given geographical region. A constant-constrained \( \delta_{w_{p}} \) means an efficient search and tolerance of frequent updates in GridSE. This performance advantage is ensured by two properties, the fixed pattern of cell partitioning and the fixed resolution of a cell in DGGS. For instance, given a database located within the United States with a size up to \( 10^{9}, 10^{9}, \) or \( 10^{7} \), the number of DGGS cells is stabilized at around 1615 with each cell in the 0.01 km² scale.

Cleaning Deleted Items. Allowing the size of \( EDB \) to grow with each update including deletions is a common strategy in building dynamic SSE [14,35]. This simplifies the deletions while releasing forward and backward privacy. GridSE can mitigate this growth via a periodic “clean-up” operation that has been adopted in previous schemes [14]. The "clean-up" operation involves the client to remove the deleted identifiers in \( \text{valList} \) when he decrypts a search result \( R_{w_{p}} \). The remaining identifiers are then re-encrypted and sent back to the server. However, the deterministic encryption over identifiers with the repeated input to PRFs would lead to “identical” ciphertexts in \( \text{valList} \) which fail to protect privacy. "Identical" means the same ciphertexts as the previous ones. Hence, as a countermeasure, we add a new counter as input into the PRFs, which increments with every search cycle. To guarantee the correct token generation and decryption for subsequent searches, an additional counter map \( \text{IncrCnt} \) is required to be maintained locally. This design retains the same performance and backward privacy as GridSE. We omit the details of this part in this paper due to the space limit.

6.2 Towards Generic Dynamic pSSE

The key idea of GridSE to achieve forward/backward private dynamic pSSE is its utilization of \( SP^{2}E \) as a building block to support prefix search and its creation of a key-value dictionary based on the encrypted triplets \( (w, id, op) \). These techniques are also compatible with traditional dynamic SSE schemes that support only full keyword search. In this section, we provide a generic dynamic pSSE framework (Fig. 6) by leveraging the techniques in GridSE to enable prefix
search capabilities for forward/backward private dynamic keyword SSE. In this framework, one vital requirement is that the keywords with the same prefix need to be grouped and associated with the same index-key in the key-value dictionary. The framework consists of two main steps. First, when searching for a specific prefix, the server will identify the corresponding index-keys for this prefix by executing the prefix predicate function $SP^2E.PrefDec$ over the encrypted index-keys, namely $addr$. Next, the clients decrypt the blocks (i.e. files) returned from the server to obtain the result. Note that the encryption (highlighted in Fig. 6) in our framework is designated for forward/backward privacy and can be varied according to different designs, such as PRF used in ref. [12, 14], puncturable PRF in ref. [12, 34, 36] or encryption based on oblivious data structure in ref. [14, 15, 35]. While the framework allows different encryption, to enable prefix search, the message encrypted must include two parameters $seq$ and $UpdtCnt[w]$, where $UpdtCnt[w]$ stores the update operations on blocks in relation to an index-key $w$, and $seq$ records the sequence of $w$ in the index, hence making every block unique without compromising forward and backward privacy.

7 EXPERIMENTAL EVALUATION

In this section, we perform an in-depth analysis and report the performance of our scheme GridSE, comparing it with ST-ORAM [16], the state-of-the-art dynamic SSE for substring and plaintext search.

Table 1: Complexity of GridSE.

<table>
<thead>
<tr>
<th>Table 1: Complexity of GridSE.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation Time</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Encryption</td>
</tr>
<tr>
<td>$O(N + d_w)$</td>
</tr>
</tbody>
</table>

$d_w$: the number of (distinct) index-keys in the encrypted index dictionary $InxDict$ at server; $N$: the total number of updates; $a_{w_p}$: the number of updates in relation to the prefix $w_p$.

**Setup.** We implemented GridSE in Java using the JPBC [54] library for cryptographic operations, in particular, AES-256 as the PRF. All schemes were executed on a single machine while storing the database on RAM. We used an EC2 i3.xlarge AWS instance (Intel Xeon 2.30GHz CPU and 30.5GB RAM). The WAN experiment was run on two machines with a 30ms roundtrip time.

**Experiment Overview.** The experiment is conducted on the Gowalla location check-in dataset [55], a real-world dataset containing 6,442,890 records contributed by 196,591 worldwide users from which we picked 63,369 distinct users within California by Google API. We tested for various database sizes $|DB| = 10^3$-$10^7$ by randomly choosing and duplicating users. The cell codes produced from different DGGS methods are evaluated such as Niemeyer’s Geohash [2], Google S2 [3] and Uber H3 [4]. The letters of a cell code are from the textual alphabet. When building an index, we standardize the area of each cell to be 0.01km$^2$ which is equivalent to a neighboring range of 100m. Different DGGS methods yield various cell code lengths for a fixed cell area. Query ranges are tested spanning from 100m to 5km which associates the sizes from 100m$^2$ to 100km$^2$. We considered variable result between 10-10$^5$ blocks. Unless otherwise specified, after blocks were inserted, we delete at random 10% of the blocks matching the queried prefix, which is to emulate the impact of deletions on performance. All experimental values are the average obtained by 10 trials. To be precise, a location is kept with five digits such as (Long. = −97.66712, Lat. = 30.20155). This unit of digits corresponds to 1m difference in reality.

Two parameters $f$ and queried range are emphasized in our experiments. The queried range varies between 5km and 100m with the length of a queried prefix ranging from 5 to 10 characters. $f$ is a parameter (in $SP^2E$) indicating a queried result valued of 16-20 bits $w.r.t$ 256-bit AES ciphertexts.

7.1 Search Performance

We tested the computation time and communication size impacted from different $f$, queried ranges, deletions and WAN experiment for evaluating searches.

**Computation Time.** The computation time refers to the time needed for a single geographic search. As depicted in Fig. 7 (a)(b), GridSE and SOTA achieve nearly constant search
time as the database size varies while the search time for both schemes slightly increases with the result size, as more blocks need to be returned and decrypted.

Comparison with the State of the Art. It is noteworthy that Fig. 7 (a) and (b) show a 150 × −5000× speedup of our scheme over ST-ORAM. This improvement is as expected, given the fact that ST-ORAM needs to scan every possible prefix within repeated tree structures, organized by every possible suffix (including encryption) which results in a logarithmic cubic complexity. Instead, our solution leverages efficient symmetric encryption, maintaining almost constant complexity. Such improvement provides strong evidence for the feasibility and scalability of GridSE. For example, with a database of 1M blocks and a result size of 130, a geographic search over ciphertexts (including decryption) requires only 20ms. Notably, the majority of the time is spent on the decryption while the server only needs to perform the lookups. This is proved by the unaffected performance with various parameter $f$ as shown in Fig. 7, e.g., a 4-bit difference in $f$ results in a negligible time difference of approximately 2ms for an encrypted database size of $|DB| = 1M$. Moreover, it is crucial to highlight the efficiency of GridSE, especially when achieving both forward and backward privacy.

Deletion Impact. One interesting observation is the independence of search time from the volume of deletions. As shown in Fig. 7 (c), as the percentage of previous deletion increases from 0 to 50%, there is no notable variation in search time for a fixed database size of 1M and fixed result size of 10K. This is because the deleted entries are not physically removed from the index. Notice that “deletion” is recorded in the parameter $op$ stored within the index. Consequently, the deletion only influences the phase after decryption, i.e., to pick up the block identifiers that have been added but not yet deleted.

Experiment over WAN. To simulate the real-world scenario, we measure the end-to-end search time with separate server and client machines under the WAN setting. The latency between the server and the client is around 40ms. Fig. 7 (d) depicts the search time, breaking down to computation and communication, for database size $|DB| = 1M$. The data reported is for Niemeyer’s Geohash. Although communication is the dominant overhead for both schemes, ST-ORAM imposes much heavier communication overhead due to the numerous interactive rounds with poly-logarithmic sizes transferred to the server while in our scheme, only 1 round of communication is needed. From our experiments, GridSE outperforms ST-ORAM by a factor of 67× in terms of the result size.

Consider a database of size $10^6$ with returned locations size of $10^3$. Under such conditions, GridSE’s performance metrics would be 83ms for communication, 18ms for computation, and a client storage requirement of 30.7KB. This gives us a clearer grasp of the practical feasibility of our scheme.

Comparison Cost. Fig. 8 shows the communication cost for a search under varying database sizes (a) and different result sizes (b). The communication cost grows linearly with the result size but is independent of the database size. Compared with ST-ORAM, GridSE saves 99% communication cost. This is achieved because in our scheme, any search contains only one round of communication between the client and the server.
and server. The client only needs to send one “token” (TList) which is nearly constant in size and receive the result from the server. The communication cost can be further optimized by requiring the server to send the encrypted indexes back and then remove the deleted blocks, or asking the client to submit a cleanup process after each search.

7.2 Update Performance

Fig. 9 (a) and (b) show the update time and communication size for one update operation (add or deletion of a block) across varying database sizes, respectively. Notably, both the update time and communication size are constant (5µs for update time and 8 bytes for update size) and independent of the database size. This is attributed to the constant communication cost incurred by GridSE, where only a key-value pair \((addr, val)\) needs to be transmitted from client to server, therefore highlighting the scalability of GridSE in real-world GIS systems.

7.3 Comparison with Plaintext Search

To further evaluate the efficiency of GridSE, we compare it with the plaintext prefix search for a database of size \(10^6\) and a result of \(10^3\). We implement this baseline via B-tree, the default index in MySQL for common keyword searches [56]. As shown in Table 2, GridSE incurs only 1.4× additional computation cost and 0.9× additional communication overhead compared to the plaintext search. This is because GridSE is built atop lightweight operations, e.g., XOR and hash functions. Considering its privacy-enhancing features, GridSE is promising in terms of real-time response.

8 Conclusion and Future Work

In this work, we propose GridSE, the first dynamic prefix symmetric searchable encryption (pSSE) scheme that supports fast secure geographic search with data dynamics and achieves backward and forward privacy simultaneously. We first introduce a new cryptographic primitive \(SP^2E\) for identifying whether a keyword contains a given prefix based on a lightweight cryptographic hash function. By leveraging the lightweight cryptographic operations, GridSE built atop \(SP^2E\) introduces almost constant overhead within a given geographical grid and is thus highly scalable. Experimental results show that GridSE achieves 150× - 5000× speedup as compared to the state of the art, and its performance is close to plaintext search, proving its feasibility for large-scale GIS/LBS systems.

Due to the constrained resources, we leave the implementation of GridSE in a real-world GIS/LBS system as a future work. We also provide several potential research directions here. While GridSE achieves the attractive prefix SSE for secure geographic search, which can also be used as generic prefix search primitive in other applications, a more versatile version - substring SSE is of great interest to the community but has not been supported yet. We point out that the lightweightness of GridSE is made possible by leveraging secure bitwise operations. Combining such bit-level design with more complicated data structures, such as trees, might lead to a potential solution for substring SSE. In addition, extending GridSE to the multi-client setting is also a challenging but appealing direction.

References


Appendix A  Definition for SSE Forward/Backward Privacy

Forward Privacy. Forward privacy ensures that an update remains independent of past operations at the time this update occurs. Namely, previous searches will not leak any information about the blocks to be updated. A forward private scheme hides whether an addition is about a new block or one that might have been previously searched for.

Definition A.1 (Forward Privacy of Dynamic SSE). An L-adaptively secure dynamic SSE scheme is forward private if the leakage function \( L^{updt} \) can be written as:

\[
L^{updt}(op, w, id) = L^{updt}(op, id)
\]

where \( L' \) is a stateless function, \( w \) is an index key, \( op \) is insertion or deletion and \( id \) is a block identifier.

Backward Privacy. Backward privacy limits the information that the server can learn about blocks which are previously added but later deleted during searches related to them. In an ideal situation, the previously deleted blocks should not revealed to the adversary, at least the identifiers [14]. To capture backward privacy, we first follow the notation of [12] and then give the final definition.

Recall that an update is a tuple \((u, op, in)\) where input \( in = (w, id) \) with the modified block \( id \), operation \( op = add/del \), and \( u \) is the timestamp. For an index-key \( w \), TimeDB(w) is a function that returns all timestamp/block-identifier pairs (of matching index-key \( w \)) that have been added to \( DB \) but not deleted later.

\[
TimeDB(w) = \{(u, id) | (u, add, (w, id)) \in Q \text{ and } \forall' \in Q' \ (u', del, (w, id)) \notin Q' \}
\]  

where \( Q \) is a list of queries that have been executed. An element of \( Q \) indicates a query of the form \((u, w)\) where \( u \) is query timestamp, and \( w \) is the searched index-key.

The function \( Updates(w) \) returns the timestamp of all insertion and deletion operations on \( w \) in the previous queries \( Q \). That is

\[
Updates(w) = \{(u, id) | (u, add, (w, id)) \in Q \text{ or } (u, del, (w, id)) \notin Q \}
\]

DelHist(w) is a function of capturing previously deleted entries, which returns the timestamp of all insertion and deletion operations. The output of it reveals which deletion corresponds to which addition.

\[
DelHist(w) = \{(u^{add}, u^{del}) | (u, add, (w, id)) \in Q \text{ or } (u, del, (w, id)) \notin Q \}
\]

From the above functions, we give the formal definition of backward privacy with the three types of leakage, from Type-I which reveals information the least to Type-III which leaks the most. A parameter \( a_w \) is introduced here, which indicates the total number of updates associated with \( w \).

Definition A.2 (Backward Privacy of Dynamic SSE). An L-adaptively secure dynamic SSE scheme is backward private:

BP-Type-I: \( \forall w \in \mathbb{W}, \forall op \in \{add, del\}, \forall id \in \mathbb{I} \) such that \( \exists u \) with \( (u, add, (w, id)) \in Q \) and \( (u, del, (w, id)) \notin Q \),

\[
L^{updt}(op, w, id) = L^{updt}(op, id)
\]  

where \( L' \) and \( L'' \) are stateless functions.

Note that the above definition assumes the leakage of actually retrieving blocks will be allowed, namely the blocks that currently contain an index-key \( w \). Specifically, the function TimeDB reveals the identifiers of the blocks matching \( w \).

Appendix B  Security Experiments of SP²E

Fig. 10 presents the security experiments for encryption and token generation of SP²E in respective.

Appendix C  Correctness of PrefDec if \( w_p \sqsubseteq w \)

Fig. 11 shows the output formula of PrefDec, that correctly exposes the bits \( 0^J \) if \( w_p \sqsubseteq w \), for which we highlight these bits using the color blue. The correctness of \( w_p \sqsubseteq w \) is hold since there exists an indicator if the current keyword \( w \) satisfies the prefix query requirement, and \( 0^J \) is such an indicator.

Appendix D  Correctness of PrefDec if \( w_p \nsubseteq w \)

If \( w_p \nsubseteq w \), the correctness of PrefDec should guarantee that the output of PrefDec is an indistinguishable value without any indicator-bits, namely consecutive bits valued at zero. We consider two cases under this condition of \( w_p \nsubseteq w \), one is the first letters of \( w_p \) are the prefix of \( w \), but \( w_p \) is not a prefix of \( w \), e.g., \( w_p = "apps" \), \( w = "apple" \); and the other is any composed parts of the sequence in \( w_p \) is not a prefix of \( w \), e.g., \( w_p = "star" \), \( w = "apple" \). To formulate both cases, we introduce a set \( J \) for denoting each sequence of the character of the difference set \( w_J \), that is \( \{w - w_p | w_p \} \) in \( w_p \). For instance, \( w_J = 's' \) and \( J = 4 \) if \( w_p = "apps" \) and \( w = "apple" \), because 's' is the different letter in \( w_p \) compared to \( w \); and \( J \) indicates the sequence of 's' in \( w_p \). We have \( 1 \leq |J| \leq |w_p| \), and \( |J| \) is the size of \( J \). The output of PrefDec if \( w_p \nsubseteq w \) can be derived to the formula in Fig. 12. This formula shows that there exists no indicator-bits exposed in the decrypted result. Thus, the correctness of PrefDec for \( w_p \nsubseteq w \) is validated.
**Exp^{Ind-CPA}_{\Lambda,SP^2E.Enc}(\lambda):**

\[(SK, MSK) \leftarrow \text{KeyGen}(1^\lambda, t); \text{table } U, V \leftarrow \emptyset\]

\[(w_0, w_1, st) \leftarrow A^{\text{Enc}}_{SK,seq}(\cdot)(1^\lambda), \text{s.t. } |w_0|, |w_1| \leq t\]

\[b \overset{\$}{\leftarrow} \{0, 1\};\]

\[\hat{c} \leftarrow \text{Enc}(SK, \text{PreEnc}(MSK, seq, w_b))\]

\[b' \leftarrow A^{\text{Enc}}_{SK,seq}(\cdot)(st, \hat{c})\]

Return \((b' = b)\).

**Exp^{Ind-f-CPA}_{\Lambda,SP^2E.TKGen}(\lambda):**

\[(SK, MSK) \leftarrow \text{Setup}(1^\lambda, t); \text{table } W \leftarrow \emptyset\]

\[(w_{p_0}, w_{p_1}, st) \leftarrow A^{\text{TKGen}}_{SK,seq,sk}(\cdot)(1^\lambda), \text{s.t. } |w_{p_0}|, |w_{p_1}| \leq t, |w_{p_0}| = |w_{p_1}|\]

\[b \overset{\$}{\leftarrow} \{0, 1\};\]

\[\hat{k}' \leftarrow \text{TKGen}(SK, MSK, \delta, seq, w_{p_0})\]

\[b' \leftarrow A^{\text{TKGen}}_{SK,seq,sk}(\cdot)(st, \hat{k}')\]

Return \((b' = b)\).

**Figure 10:** Semantic Security of Encryption and Token Generation in SP^2E

\[
k' \oplus ct = 1: = \left\lfloor \frac{|w|}{|\lambda|} \right\rfloor \left( \bigoplus_{i=1}^{\left\lfloor \frac{|w|}{|l(\lambda)|} \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_{p_i}} \parallel seq \parallel i) \right) \right) \oplus \left( \bigoplus_{i=1}^{\left\lfloor \frac{|w|}{|l(\lambda)|} \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \right)
\]

Note that: we have \(sk_{w_i} = sk_{w_{p_i}}\) for \(1 \leq i \leq |w_{p_i}|\), if \(w_{p_i} \subseteq w\), \(w_{p_i} \supseteq w\).

\[
2: = \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \oplus \left( \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \right)
\]

Note that: for simplicity, we denote the part before \( \bigoplus \) as \( D_H = \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \).

\[
3: = (\eta \oplus \delta).\text{sub}(0, pos_1) \parallel 0^f \parallel (D_H \oplus (\eta \oplus \delta)).\text{sub}(pos_2, l(\lambda))
\]

**Figure 11:** Expanding the output \(k' \oplus ct\) of PrefDec if \(w_{p_i} \subseteq w\)

\[
k' \oplus ct = 1: = \text{Step 1 is as the same as the step 1 in Figure 11.}
\]

\[
2: = \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \oplus \left( \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \right)
\]

Note that: for simplicity, we denote \(S_H = \bigoplus_{i=\left\lfloor |w_{p_i}| + 1 \right\rfloor}^{\left\lfloor |w_{p_i}| + 1 \right\rfloor} \left( 0^{(i-1)f} \parallel H(sk_{w_i} \parallel seq \parallel i) \right) \) and \(J\) is the sequence of \(w_{p_i}\).

\[
3: = \left( S_H \oplus (\eta \oplus \delta) \right).\text{sub}(0, pos_1) \parallel S_H.\text{sub}(pos_1, pos_2) \parallel \left( (S_H \oplus D_H) \oplus (\eta \oplus \delta) \right).\text{sub}(pos_2, l(\lambda))
\]

**Figure 12:** Expanding the output \(k' \oplus ct\) of PrefDec if \(w_{p_i} \supseteq w\)