DISCO: Dynamic Searchable Encryption with Constant State

Xiangfu Song
National University of Singapore
songxf@comp.nus.edu.sg

Yu Zheng
Chinese University of Hong Kong
yuzheng404@link.cuhk.edu.hk

Jianli Bai
University of Auckland
jbai795@aucklanduni.ac.nz

Changyu Dong
Guangzhou University
changyu.dong@gmail.com

Zheli Liu
Nankai University
liuzheli@nankai.edu.cn

Ee-Chien Chang
National University of Singapore
changec@comp.nus.edu.sg

ABSTRACT

Dynamic searchable encryption (DSE) with forward and backward privacy reduces leakages in early-stage schemes. Security enhancement comes with a price – maintaining updatable keyword-wise state information. State information, if stored locally, incurs significant client-side storage overhead for keyword-rich datasets, potentially hindering real-world deployments.

We propose DISCO, a simple and efficient framework for designing DSE schemes using constant client state. DISCO combines range-constrained pseudorandom functions (RCPRFs) over a global counter and leverages nice properties from the underlying primitives and index structure to simultaneously achieve forward-and-backward privacy and constant client state. To configure DISCO concretely, we identify a set of RCPRF properties that are vital for the resulting DISCO instantiations. By configuring DISCO with different RCPRFs, we resolve efficiency and usability issues in existing schemes. We further optimize DISCO's concrete efficiency without downgrading security. We implement DISCO constructions and report performance, showing trade-offs from different DISCO constructions. Besides, we compare the practical efficiency of DISCO with existing non-constant-state DSE schemes, demonstrating DISCO's competitive efficiency.

CCS CONCEPTS

• **Security and privacy → Management and querying of encrypted data**.

KEYWORDS

Searchable encryption, Forward privacy, Backward privacy, Client storage

ACM Reference Format:
Xiangfu Song, Yu Zheng, Jianli Bai, Changyu Dong, Zheli Liu, and Ee-Chien Chang. 2024. DISCO: Dynamic Searchable Encryption with Constant State. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3634737.3637674

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3637674

1 INTRODUCTION

Searchable encryption [25] allows a client to search over its remote outsourced encrypted database held by a server while only revealing well-defined leakages. Dynamic searchable encryption (DSE) further supports updates (*i.e.*, adding/deleting files) to the encrypted database at the client's request. Earlier DSE schemes [4, 19, 20] reveal the linkage between a new update query with previous search/update queries, which is vulnerable to the adaptive file-injection attack [34]. Forward privacy ensures that the server cannot match an update with the prior search/update queries to the same keyword, and backward privacy prevents documental revealing afterward if a keyword has been deleted from this document. Recently, many forward/backward private DSE schemes [1, 2, 12, 14, 21, 22, 26, 32] were proposed. Most of them employing lightweight primitives can achieve competitive efficiency as early-stage efficient DSE schemes [4, 19, 20] without forward/backward privacy.

To achieve forward privacy, existing DSE schemes, despite slight differences, maintain some necessary updatable state for each keyword. For instance, in a counter-based DSE scheme, the client maintains an update counter a_w for a keyword w. The client simply stores encrypted indexes at positions of $F(\mathsf{k}, w||1)$, $F(\mathsf{k}, w||2)$, $\cdots$, $F(\mathsf{k}, w||a_w)$ on the server side, where F is a pseudorandom function (PRF) and k is a secret key owned by the client. To perform a new update to w, the client sends a new encrypted index with $F(\mathsf{k}, w||a_w + 1)$ to the server and updates $a_w \leftarrow a_w + 1$. Intuitively, $F(\mathsf{k}, w||a_w + 1)$ is unpredictable to the server from the security of PRF F, thus accomplishing forward privacy.

Storing the keyword-wise state locally demands extra client storage. Typically, client storage can be divided into two categories: *permanent* storage (*e.g.*, holding secret key) and *temporary* storage (*e.g.*, caching updates and search results). While temporary storage is acceptable since update pairs or search results can be discarded when no longer needed, local maintenance of the state incurs permanent storage linear to the size of the keyword set, raising challenges in both efficiency and usability.

1.1 Existing Solutions

Outsourcing encrypted states to the server is feasible, and the client accesses states when necessary. However, encryption is insufficient as the server can match queries from the storage access pattern, which motivates ORAM-based solutions of removal of access patterns [15, 28]. Specifically, Moneta [2] utilized Tworam [13] to design small-client-storage DSE through garbled circuits and ORAM. Alternative constructions [11, 14] with small client storage utilized Oblivious MAP (OMAP) [33], a more advanced oblivious primitive from ORAM. However, these solutions are mainly on theoretical

Table 1: Efficiency comparison with several DSE schemes. Let DB be an index database, N=#indexes in DB. Let $\mathrm{DB}(w)$ denote the documents containing w, then $n_w = |\mathrm{DB}(w)|$ denotes the number of documents containing w. a_w = #updates (insertions/deletions) for w. i_w=#insertion updates for w. d_w = # deletion updates for w. $\mathbb{W}$ denotes the keyword set. D=#documents, $|\mathbb{W}|$ = #distinct keywords. c is the global counter value. $\mathrm{RT_s}$ denotes #roundtrips for search (we count the round for the actual file retrieval), and $\mathrm{RT_u}$ denotes #roundtrips for an update. L is an upper bound for the number of batch updates, and $L \ll N$ in practice. All listed schemes support forward privacy. BP means backward privacy. Type-I BP reveals the least and type-III reveals the most.

Scheme	Computation		Communication				State	BP
	Search	Update	Search	Update	$\mathrm{RT_s}$	$\mathrm{RT_u}$		
Sophos [1]	$O(a_w)$	$O(a_w)$	$O(n_w)$	$O(1)$	1	1	$O(\lvert\mathbb{W}\rvert)$	-
FAST [26]	$O(a_w)$	$O(1)$	$O(n_w)$	$O(1)$	1	1	$O(\lvert\mathbb{W}\rvert)$	-
Mitra [14]	$O(a_w)$	$O(1)$	$O(a_w)$	$O(1)$	2	1	$O(\lvert\mathbb{W}\rvert)$	II
Moneta [2]	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(\log^3 N)$	2	2	$O(1)$	I
Orion [14]	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(n_w \log^2 \lvert\mathbb{W}\rvert)$	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(1)$	I
Horus [14]	$O(n_w \log d_w \log N + \log^2 \lvert\mathbb{W}\rvert)$	$O(\log^2 N)$	$O(n_w \log i_w + \log^2 \lvert\mathbb{W}\rvert)$	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(1)$	III
QOS [11]	$O(n_w \log i_w + \log^2 \lvert\mathbb{W}\rvert)$	$O(\log^2 N)$	$O(n_w \log i_w + \log^2 \lvert\mathbb{W}\rvert)$	$O(\log^2 N)$	$O(\log \lvert\mathbb{W}\rvert)$	$O(\log \lvert\mathbb{W}\rvert)$	$O(1)$	III
SD_a [11]	$O(\log N + a_w)$	$O(\log N)(am.)$	$O(\log N + a_w)$	$O(\log N)(am.)$	1	2	$O(1)$	II
SD_d [11]	$O(\log N + a_w)$	$O(\log^3 N)$	$O(\log N + a_w)$	$O(\log^3 N)$	2	$O(\log N)$	$O(1)$	II
Close-fb [17]	$O(L + a_w)$	$O(L - c)$	$O(a_w)$	$O(1)$	2	1	$O(1)$	II
$\mathrm{DISCO_h}$	$O(L + a_w)$	$O(L - c)$	$O(a_w)$	$O(1)$	2	1	$O(1)$	II
$\mathrm{DISCO_t}$	$O(c + a_w)$	$O(c)$	$O(a_w)$	$O(1)$	2	1	$O(1)$	II
$\mathrm{DISCO_g}$	$O(c + a_w)$	$O(\log L)$	$O(\log L + a_w)$	$O(1)$	2	1	$O(1)$	II

interest, hardly achieving practical efficiency compared with DSE schemes [1, 2, 12, 14, 22, 26] built on lightweight techniques.

CLOSE-FB [17] is a DSE scheme employing a fixed-length chain of secrets $s_0 \xleftarrow{H} s_1 \cdots \xleftarrow{H} s_L$ in reverse order during updates, where H is a cryptographic hash function, s_L is an initial (pseudo) random seed, and $s_i = H(s_{i+1})$ for $1 \le i < L$. The onewayness of the hash chain ensures that the server can recover nodes $\{s_1, s_2, \cdots s_c\}$ from s_c (thus enabling search), but not beyond the range (thus ensuring forward privacy). In CLOSE-FB, the client stores only a single global counter *shared by all keywords*. Yet, CLOSE-FB increases computational costs linear to the length of hash chains. When running out of available nodes in a hash chain, CLOSE-FB rebuilds the whole database from scratch, severely downgrading usability.

1.2 Our Solution

This paper proposes DISCO (**D**ynamic searchable encrypt**I**on with **S**mall **C**lient st**O**rage), a simple and efficient framework for designing constant-client-state DSE. We design DISCO inspired by CLOSE-FB and DSE schemes from *range constrained pseudorandom functions* (RCPRFs). We exploit nice properties of involved primitives and structure for designing DSE using constant state.

RCPRF abstraction for DSE. RCPRF is a PRF variation that allows an RCPRF master key holder to delegate PRF evaluation over a range by issuing a constrained key. The constrained key can only be used to evaluate PRF over a restricted range defined by the key. Previous works [1, 2] leveraged RCPRF to evolve secret tokens, yet requiring permanent client storage proportional to keywords set. We borrow the idea of using a global counter [17] and exploit properties from cryptographic primitive and structures, attaining *constant* client-side permanent state.

We formalize several RCPRF schemes and identify a set of RCPRF properties vital in the design of DISCO. Atop formalization, we propose a hybrid RCPRF abstraction for combining good properties given different RCPRF schemes. Since RCPRFs are useful in other areas [3] beyond DSE, our findings are of independent interest.

Our framework and practical instantiations. DISCO is not limited to one construction but provides a simple and generic framework for designing low-client-state DSE. DISCO captures several DISCO instantiations when parameterized with different RCPRFs. For example, CLOSE-FB can be regarded as a concrete DISCO instantiation, yet DISCO provides a simpler index structure and more optimizations. When we configure DISCO with different RCPRFs, DISCO resolves efficiency and usability issues in existing works, achieving improved efficiency with versatile trade-offs.

We provide three DISCO instantiations $\mathrm{DISCO_h}$, $\mathrm{DISCO_g}$, and $\mathrm{DISCO_t}$ with different efficiency tradeoffs as compared in Table 1. Overall, three constructions trade communication and client's state with a price of increased computation compared with [1, 14, 26], while ORAM-based approaches [2, 11, 14] require more communication and higher round complexity. Indeed, communication, rather than computation, is more likely to be a bottleneck, which is also confirmed in our performance evaluation, showing DISCO's superior efficiency during updates and searches. Moreover, we propose practical techniques to reduce computational overhead concretely. In particular, we show how to achieve parallel search, reduce RCPRF evaluation, and perform database merge to alleviate the side effects of using a global counter. These optimizations improve DISCO's concrete efficiency without downgrading security.

We implement DISCO and compare the performance trade-offs from different DISCO constructions, showing distinct efficiency properties of different DISCO constructions. We also compare DISCO with other DSE schemes. The evaluation results show that DISCO is much more efficient than state-of-the-art small-client-state DSE schemes. Compared with previous DSE schemes with a non-constant state, DISCO only introduces a small increased running time that is unnoticeable from human perception, while the communication remains almost the same.

Contribution. We summarize our contributions as follows:

- We propose DISCO, a simple framework for constructing forward-and-backward private DSE under the constraint of a constant client state.

- We identify a set of RCPRF properties that are vital in the design of DISCO. We also propose a hybrid RCPRF design by combining properties from different RCPRF schemes.
- We present candidate DSE constructions with small client storage under DISCO framework with configuable trade-offs. Our construction and the following optimizations resolve efficiency issues in previous low-client-storage DSE constructions.
- We implement three DISCO constructions and perform evaluations. The performance shows our constructions are practical and efficient. We report the efficiency and trade-off properties of different DISCO constructions.

2 PRELIMINARIES

2.1 Notations

$x \xleftarrow{\$} X$ denotes sampling an element x randomly from a set X and $|X|$ means the set cardinality. $\{0,1\}^\ell$ is a set of strings with ℓ bits and $\{0,1\}^*$ means a set of strings in the arbitrary length. $a||b$ denotes the concatenation of string a and string b. $|a|$ is a's bit length. For integers c, a, b, we use $[c]$ to denote the set $\{1, 2, \cdots, c\}$, and $[a, b]$ to denote $\{a, a+1, \cdots, b-1, b\}$. Let λ be the computational security parameter. A function $\nu : \mathbb{N} \to \mathbb{R}$ is negligible in λ if for every positive polynomial p, $\nu(\lambda) < 1/p(\lambda)$ for a sufficiently large λ. We use $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ to represent the unspecified polynomial and negligible functions in λ, respectively. Probabilistic polynomial time is abbreviated to PPT.

2.2 Cryptographic Primitives

Pseudorandom functions. Let $F : \mathcal{D} \times \mathcal{K} \to \mathcal{R}$ be a length-preserving, keyed function. F is a *pseudorandom function* (PRF) if for all PPT adversary $\mathcal{A}$, it holds that $|\Pr[\mathsf{Adv}^{F(k,\cdot)}(1^\lambda) = 1] - \Pr[\mathsf{Adv}^{f(\cdot)}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$ where $k \in \mathcal{K}$ is chosen uniformly and $f : \mathcal{D} \to \mathcal{R}$ is a truly random function.

Trapdoor permutations. A trapdoor permutation (TDP) $\pi : \mathcal{PK} \times \mathcal{D}_\mathsf{t} \to \mathcal{D}_\mathsf{t}$ is an asymmetric primitive. Using a public key $\mathsf{pk} \in \mathcal{PK}$, one can efficiently compute $y \leftarrow \pi_\mathsf{pk}(x)$ for $x \in \mathcal{D}_\mathsf{t}$. Only the secret key holder can efficiently compute its inverse $\pi^{-1} : \mathcal{SK} \times \mathcal{D}_\mathsf{t} \to \mathcal{D}_\mathsf{t}$, *i.e.*, $x \leftarrow \pi^{-1}_\mathsf{sk}(y)$. The security of TDP requires that those who only have the public key can compute $\pi(x)$ for any $x \in \mathcal{D}_t$ but not $\pi^{-1}(y)$ for $y \xleftarrow{\$} \mathcal{D}_t$. Refer to Def. A.1 in §A for a formal definition.

Goldreich-Goldwasser-Micali PRF. Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ be a length-doubling pseudo-random generator (PRG) and $G_0(\mathsf{s})$ and $G_1(\mathsf{s})$ be the first and second half of $G(\mathsf{s})$ for a seed $\mathsf{s} \in \{0,1\}^\lambda$, the GGM-PRF [16] over an n-bit input x is defined as $F_\mathsf{G}(\mathsf{s}, x) = G_{x_{n-1}}(\ldots G_{x_1}(G_{x_0}(\mathsf{s})))$, where $\mathsf{s} \in \{0,1\}^\lambda$ is a seed associated with the root and $x = x_0||\ldots||x_{n-1}$ is the binary representation of x.

The GGM PRF allows delegated evaluation over a range to an evaluator by issuing a constrained key ck_c. The idea is to compute the minimum nodes in the tree to cover the subset, for which we call the *Best Range Cover* (BRC) nodes. In this paper, we denote the algorithm of finding the best range cover as $\mathsf{ck}_c \leftarrow F_\mathsf{G}.\mathsf{BRC}(\mathsf{s}, [c])$. For example, in Fig. 1, to delegate the range between the first leaf and the seventh leaf, the key holder computes the green nodes, from which the evaluator can reconstruct all leaves corresponding to the range, but not the leaves out of the range.

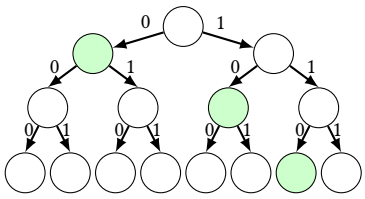


Figure 1: An example of GGM tree.

Symmeric Encryption. A symmetric encryption scheme with key space $\mathcal{K}$, message space $\mathcal{M}$ and ciphertext space C consists of three polynomial-time algorithms $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

- $\mathsf{Gen}(1^\lambda)$: It takes as input a computational security parameter λ, outputs a secret key $\mathsf{k} \in \mathcal{K}$.
- $\mathsf{Enc}(\mathsf{k}, m)$: It takes as input a secret key k and a message $m \in \mathcal{M}$, outputs a ciphertext $c \in C$.
- $\mathsf{Dec}(\mathsf{k}, c)$: It takes as input a secret key k and a ciphertext $c \in C$, output a decrypted message $m \in \mathcal{M}$ or a failure symbol $\perp$.

Security. We require a stronger security notion for symmetric encryption called *pseudo-randomness against chosen-plaintext attack* (PCPA) [10]. The security is defined in Def. A.2 from §A.

2.3 Dynamic Searchable Encryption

Let a database $\mathbf{DB} = \{(\mathsf{id}_i, \mathsf{W}_i)\}_{i=1}^D$ be a D-vector of identifier-keyword pairs, where id_i is a document identifier and W_i is the keyword set contained in document id_i. $|\mathsf{W}|$ and $|\mathbf{DB}|$ denote the number of keywords and the number of documents. The universe of keywords is $\mathsf{W} = \cup_{i=1}^D \mathsf{W}_i$, and $N = \sum_{i=1}^D |\mathsf{W}_i|$ denotes the number of document-keyword pairs. $\mathbf{DB}(w) = \{\mathsf{id}_i \mid w \in \mathsf{W}_i\}$ denotes the set of documents containing the keyword w. Given a set of updates of tuples $\mathsf{IN} = \{(\mathsf{op}_i, w_i, \mathsf{id}_i)\}$ with $|\mathsf{IN}|$ keyword/document pairs, we use $\mathsf{IN}(w) = \{(\mathsf{op}_i, w, \mathsf{id}_i) \in \mathsf{IN}\}$ to denote updates corresponding to the keyword w. We use $\mathsf{W}(\mathsf{IN})$ to denote the set of unique keywords contained in IN.

Definition 2.1. A DSE scheme Π is defined by a triple of protocols:

- $((K, \sigma); \mathbf{EDB}) \leftarrow \mathsf{Setup}(1^\lambda, \mathbf{DB}; \perp)$: On input security parameter λ and a database **DB**, the protocol outputs secret key and client's state (K, σ) to the client, and the encrypted database **EDB** to the server. We use notation $\mathsf{Setup}(1^\lambda; \perp)$ when **DB** is empty.
- $((\sigma', \mathbf{DB}(w)); \mathbf{EDB}') \leftarrow \mathsf{Search}(K, \sigma, w; \mathbf{EDB})$: On input K, σ, and a keyword w, the client outputs an updated state σ' and $\mathbf{DB}(w)$. On input **EDB**, the server outputs the updated encrypted database $\mathbf{EDB}'$.
- $(\sigma'; \mathbf{EDB}') \leftarrow \mathsf{Update}(K, \sigma, \mathsf{IN}; \mathbf{EDB})$: On input K, σ, and an update batch IN, the client outputs σ'. On input **EDB**, the server outputs updated $\mathbf{EDB}'$.

Security definition. The security definition follows Ideal and Real simulation paradigm [18, 20], which guarantees that no more information is leaked except for the explicit leakages $\mathcal{L} = \{\mathcal{L}_\mathsf{Setup}, \mathcal{L}_\mathsf{Search}, \mathcal{L}_\mathsf{Update}\}$. Two games $\mathbf{Real}^\Pi_\mathcal{A}$ and $\mathbf{Ideal}^\Pi_{\mathcal{A},\mathcal{S}}$ are defined to the real-world execution and ideal-world simulation, respectively. In $\mathbf{Real}^\Pi_\mathcal{A}$, DSE protocols are executed as proposed, where the adversary $\mathcal{A}$ observes the real transcripts of execution. In $\mathbf{Ideal}^\Pi_{\mathcal{A},\mathcal{S}}$, a simulator $\mathcal{S}$ simulate the transcript $\mathcal{S}(\mathcal{L})$ to the adversary $\mathcal{A}$ by

accessing to the leakage function $\mathcal{L}$. The adversary is required to output a bit b that represents the ability to distinguish two games.

Definition 2.2. A DSE scheme is an $\mathcal{L}$-adaptively-secure if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that $|\Pr(\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)=1) - \Pr(\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)=1)| \leq \mathsf{negl}(\lambda)$.

2.4 Leakage Functions

Let Q be the list of all queries performed in the history in which the search query is denoted as (t, w) and an update query is denoted as $(\mathsf{t}, \mathsf{op}, \mathsf{id}, w)$. The timestamp t identifies each query. Let $\mathsf{sp}(w)$ denote the search pattern defined as $\mathsf{sp}(w) = \{\mathsf{t} : (\mathsf{t}, w) \in Q\}$, which only matches search queries. We use $\mathsf{TimeDB}(w) = \{(\mathsf{t}, \mathsf{id}) \mid (\mathsf{t}, \mathsf{add}, \mathsf{id}, w) \in Q \wedge \forall \mathsf{t}', (\mathsf{t}', \mathsf{del}, \mathsf{id}, w) \notin Q\}$ to denote all timestamps of updates to keyword w excluding the deleted ones. $\mathsf{Updates}(w) = \{\mathsf{t} \mid (\mathsf{t}, \mathsf{add}, \mathsf{id}, w) \in Q \vee (\mathsf{t}, \mathsf{del}, \mathsf{id}, w) \in Q\}$ is a function that returns all timestamps of update queries (add/del) to w. $\mathsf{DelHist}(w) = \{(\mathsf{t}^{\mathsf{add}}, \mathsf{t}^{\mathsf{del}}) \mid \exists\, \mathsf{id} : (\mathsf{t}^{\mathsf{add}}, \mathsf{add}, (w, \mathsf{id})) \in Q \wedge (\mathsf{t}^{\mathsf{del}}, \mathsf{del}, (w, \mathsf{id})) \in Q\}$ is the function that returns for all deletion operations to w, together with the timestamp of the inserted entry it removes. We have the following formal privacy definitions.

Forward and backward privacy. Forward privacy ensures an update is not linked with any previous queries. Backward privacy limits the leakage from a search for w for which some identifiers have been previously deleted. Backward privacy is formally defined by Bost *et al.* [2] for three types with different leakage patterns. Type-I BP reveals the least information and type-III reveals the most.

Definition 2.3 (Forward privacy [2]). An $\mathcal{L}$-adaptively-secure DSE scheme is forward private if the update leakage satisfies $\mathcal{L}_{\mathsf{Update}}((\mathsf{op}, w, \mathsf{id})) = \mathcal{L}'(\mathsf{op}, \mathsf{id})$. $\mathcal{L}'$ is a stateless function and leaks no keyword information.

For batch update $\mathsf{IN} = \{\mathsf{op}_i, w_i, \mathsf{id}_i\}$ containing many update tuples, the update leakage function satisfying forward privacy can be written as $\mathcal{L}_{\mathsf{Update}}(\mathsf{IN}) = \mathcal{L}'(|\mathsf{IN}|, \{\mathsf{op}_i, \mathsf{id}_i\}_{i\in|\mathsf{IN}|})$.

Definition 2.4 (Backward privacy [2]). An $\mathcal{L}$-adaptively-secure DSE scheme achieves backward privacy:

- BP-I (BP with insertion pattern): iff $\mathcal{L}_{\mathsf{Update}}(\mathsf{op}, w, \mathsf{id}) = \mathcal{L}'(\mathsf{op})$ and $\mathcal{L}_{\mathsf{Search}}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), a_w)$.
- BP-II (BP with update pattern): iff $\mathcal{L}_{\mathsf{Update}}(\mathsf{op}, w, \mathsf{id}) = \mathcal{L}'(\mathsf{op}, w)$ and $\mathcal{L}_{\mathsf{Search}}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{Updates}(w))$.
- BP-III (weak BP): iff $\mathcal{L}_{\mathsf{Update}}(\mathsf{op}, w, \mathsf{id}) = \mathcal{L}'(\mathsf{op}, w)$ and $\mathcal{L}_{\mathsf{Search}}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{DelHist}(w))$.
  where $\mathcal{L}'$, $\mathcal{L}''$ are stateless functions.

All three types of backward privacy leak the documents currently matching w and when they were inserted. For other leakages: BP-I only allows the leakage of "the total number of updates on w"; BP-II further allows the leakage of "when all the updates on w happened"; and BP-III further allows the leakage of "which deletion update canceled which insertion update".

3 RANGE CONSTRAINED PSEUDORANDOM FUNCTION: REVISITING AND FINDINGS

In this section, we revisit range-constrained pseudorandom functions (RCPRF) and several concrete instantiations from different cryptographic primitives. We identify properties and new findings that are vital for designing DISCO.

3.1 RCPRF Definition

An RCPRF enables a master key holder to delegate PRF evaluations over a constrained range to an evaluator by revealing a constrained key. An RCPRF $F_{\mathrm{rc}} : \mathcal{D}_{\mathrm{r}} \times \mathcal{K}_{\mathrm{r}} \to \mathcal{R}_{\mathrm{r}}$ with an additional constrained key space $\widetilde{K}_{\mathrm{r}}$ contains the following algorithms (GenKey, Constrain, Eval):

- $F_{\mathrm{rc}}.\mathsf{KeyGen}(1^\lambda)$: It takes as input a security parameter λ, outputs a master key $\mathsf{k} \xleftarrow{\$} \mathcal{K}_{\mathrm{r}}$.
- $F_{\mathrm{rc}}.\mathsf{Constrain}(\mathsf{k}, [c])$: It takes as input a master key k and a range $[c]$, outputs a constrained key $\mathsf{ck}_c \in \widetilde{K}_{\mathrm{r}}$.
- $F_{\mathrm{rc}}.\mathsf{Eval}(\mathsf{ck}_c, x)$: It takes as input a constrained key ck_c and an element $x \in \mathcal{D}_{\mathrm{r}}$, outputs $y = F_{\mathrm{rc}}(\mathsf{k}, x)$ iff $x \in [c]$; $\perp$ otherwise.

We use $F_{\mathrm{rc}}.\mathsf{BEval}(\mathsf{ck}_c, [c])$ to denote a batch evaluation of F_{rc} over the range $[c]$, which outputs c PRF outputs. The correctness of RCPRF requires that $\mathsf{Eval}(\mathsf{ck}_c, x) = F_{\mathrm{rc}}(\mathsf{k}, x)$ iff $x \in [c]$. The security of RCPRF is defined in Def. 3.1.

Definition 3.1 (Security of RCPRFs). An RCPRF F_{rc} is secure if every PPT adversary $\mathcal{A}$ has negligible advantage in λ, where the advantage is defined as $\mathsf{Adv}^{\Pi}_{\mathcal{A},F_{\mathrm{rc}}}(\lambda) = |\Pr[\mathbf{Exp}^{\mathrm{rcprf}}_{\mathcal{A},F_{\mathrm{rc}}}(\lambda)=1] - 1/2|$, and $\mathbf{Exp}^{\mathrm{rcprf}}_{\mathcal{A},F_{\mathrm{rc}}}(\lambda)$ is defined in Fig. 17 from §A.

3.2 Concrete RCPRF Instantiations

We can construct RCPRFs from hash chains, GGM trees, and trapdoor permutations. We formalize these constructions under the same algorithm interface and show three RCPRF schemes $F^{(\mathrm{h})}_{\mathrm{rc}}$, $F^{(\mathrm{g})}_{\mathrm{rc}}$, $F^{(\mathrm{t})}_{\mathrm{rc}}$, respectively. Looking ahead, these RCPRF instantiations feature different properties, bringing different properties to DISCO.

$\underline{\mathsf{KeyGen}(1^\lambda)}$	$\underline{\mathsf{Constrain}(\mathsf{k}, [c])}$	$\underline{\mathsf{Eval}(\mathsf{ck}_c, i)}$
1: $s_L \xleftarrow{\$} \{0,1\}^{2\lambda}$	1: parse $\mathsf{k} = (L, s_L)$	1: $\mathsf{ck}_c = (L, s_c, c)$
2: $\mathsf{k} \leftarrow \{L, s_L\}$	2: $s_c \leftarrow H^{L-c}(s_L)$	2: $s_i \leftarrow \perp$, $\mathsf{st}_i \leftarrow \perp$
3: **Return** k	3: $\mathsf{ck}_c \leftarrow \{L, s_c, c\}$	3: **if** $i \in [c]$ **then**
	4: **Return** ck_c	4: $s_i \leftarrow H^{c-i}(s_c)$
		5: $\mathsf{st}_i \leftarrow H(s_i \| i)$
		6: **Return** st_i

Figure 2: RCPRF $F^{(\mathrm{h})}_{\mathrm{rc}}$ from hash chains.

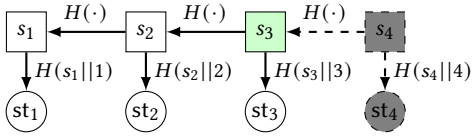


Figure 3: RCPRF $F^{(\mathrm{h})}_{\mathrm{rc}}$ from hash chains. H is a hash function. The green node corresponds to a constrained key for $[1, 3]$.

RCPRF from hash chains. We review RCPRF using hash chains in Fig. 2. Given a hash function $H : \{0,1\}^* \to \{0,1\}^{2\lambda}$, a hash chain contains L nodes derived from one initial seed $s_L \in \{0,1\}^{2\lambda}$, where the i-th node is computed as $s_i \leftarrow H(s_{i+1})$ for $i \in [1, L-1]$. We use

$y = H^n(x) = H(\cdots H(H(x)))$ to denote n-th fold iteration of H. The master key is (s_L, L). To generate a constrained key ck_c, the Constrain algorithm computes the c-th hash node from s_L, which invokes H for $L - c$ times. The evaluation algorithm Eval taking as the input a constrained key ck_c can recover any hash node s_i for $i \in [c]$, but not the nodes s_i for $i > c$. Fig. 3 depicts an example.

$\underline{\mathsf{KeyGen}(1^\lambda)}$	$\underline{\mathsf{Constrain}(\mathsf{k}, [c])}$	$\underline{\mathsf{Eval}(\mathsf{ck}_c, i)}$
1: $\mathsf{s} \xleftarrow{\$} \{0,1\}^\lambda$ 2: $\mathsf{k} \leftarrow \{L, \mathsf{s}\}$ 3: **Return** k	1: $\mathsf{ck}_c \leftarrow F_{\mathsf{G}}.\mathsf{BRC}(\mathsf{s}, [c])$ 2: **Return** ck_c	1: $\mathsf{st}_i \leftarrow \perp$ 2: **if** $i \in [c]$ **then** 3: $\mathsf{st}_i \leftarrow F_{\mathsf{G}}(\mathsf{ck}_c, i-1)$ 4: **Return** st_i

Figure 4: RCPRF $F_{\mathrm{rc}}^{(\mathrm{g})}$ from GGM PRF.

RCPRF from GGM PRF. Fig. 4 shows how to convert GGM PRF to an RCPRF. We define $(L, \mathsf{k}_{\mathsf{G}})$ as the master key. To generate a constrained key ck_c for a range $[c]$, one can compute a *best range cover* (BRC) of the GGM tree corresponding to a range $[c]$. The BRC corresponding to a range $[c]$ is sufficient to recover all leaf nodes st_i for $i \in [c]$, but not for those beyond the range. As a toy example, Fig. 5 shows a GGM tree with $L = 4$. A constrained key ck_3 contains two green nodes in this tree.

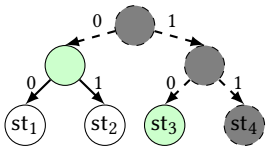


Figure 5: RCPRF $F_{\mathrm{rc}}^{(\mathrm{g})}$ from GGM trees. The green nodes correspond to a constrained key for the range $[1, 3]$.

RCPRF from trapdoor permutations. The hash-based and GGM-based RCPRFs only require symmetric primitives, but both approaches only support a bounded domain (*i.e.*, L). The RCPRF from trapdoor permutations avoids this limitation.

$\underline{\mathsf{KeyGen}(1^\lambda)}$	$\underline{\mathsf{Constrain}(\mathsf{k}, [c])}$	$\underline{\mathsf{Eval}(\mathsf{ck}_c, i)}$
1: $s_1 \xleftarrow{\$} \{0,1\}^\lambda$ 2: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \pi.\mathsf{Gen}(1^\lambda)$ 3: $\mathsf{k} \leftarrow \{\mathsf{pk}, \mathsf{sk}, s_1\}$ 4: **Return** k	1: $s_c \leftarrow \pi_{\mathsf{sk}}^{-(c-1)}(s_1)$ 2: $\mathsf{ck}_c \leftarrow \{\mathsf{pk}, s_c, c\}$ 3: **Return** ck_c	1: $\mathsf{ck}_c = \{\mathsf{pk}, s_c, c\}$ 2: $s_i \leftarrow \perp, \mathsf{st}_i \leftarrow \perp$ 3: **if** $i \in [c]$ **then** 4: $s_i \leftarrow \pi_{\mathsf{pk}}^{c-i}(s_c)$ 5: $\mathsf{st}_i \leftarrow H(s_i \| i)$ 6: **Return** st_i

Figure 6: RCPRF $F_{\mathrm{rc}}^{(\mathrm{t})}$ from trapdoor permutation.

Fig. 6 shows an RCPRF scheme from trapdoor permutation π. Fig. 7 depicts an example. Like hash-based RCPRF, the TDP-based RCPRF maintains a linked-list structure where π is used to evolve the list for *both* directions. In particular, KeyGen samples a random $s_1 \xleftarrow{\$} \mathcal{D}_{\mathsf{t}}$ and initializes the public and secret key pair $(\mathsf{pk}, \mathsf{sk})$ for trapdoor permutation π, using the TDP key generation algorithm $\pi.\mathsf{Gen}(1^\lambda)$. The master key of the resulting RCPRF is

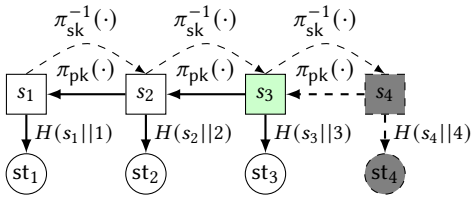


Figure 7: RCPRF $F_{\mathrm{rc}}^{(\mathrm{t})}$ from a trapdoor permutation π, and H is a hash function. The green node corresponds to the constrained key for the range $[1, 3]$.

$\mathsf{k} = (\mathsf{pk}, \mathsf{sk}, s_1)$. To generate a constrained key ck_c for a range $[c]$, the master key holder only needs to compute $s_c \leftarrow \pi_{\mathsf{sk}}^{-(c-1)}(s_1)$, where $\pi_{\mathsf{sk}}^{-(c-1)}(s_1) = \pi_{\mathsf{sk}}^{-1}(\cdots \pi_{\mathsf{sk}}^{-1}(\pi_{\mathsf{sk}}^{-1}(s_1)))$ denotes $(c-1)$-th fold iteration of π^{-1}. Anyone who has pk can only recover $\{s_i\}_{i \in [c]}$ given ck_c and pk. In particular, $s_i \leftarrow \pi_{\mathsf{pk}}^{c-i}(s_c)$, where $\pi_{\mathsf{pk}}^{c-i}(s_c) = \pi_{\mathsf{pk}}(\cdots \pi_{\mathsf{pk}}(\pi_{\mathsf{pk}}(s_{c-1})))$, *i.e.*, π_{pk} is invoked for $c - i$ times.

A note on security. One may wonder why we define RCPRF output st_i by applying H over s_i in the hash-based and TDP-based RCPRFs. This is necessary, otherwise, we cannot achieve the claimed RCPRF security: assume the adversary has queried $O_{\mathsf{k}}^{\mathsf{Constrain}}([c])$, which means the adversary learns s_c. Now the adversary issues a challenge query over $c+1$ and obtains a challenge output y. The adversary can simply check whether $s_c = H(y)$ (resp. $s_c = \pi_{\mathsf{pk}}(y)$) for $F_{\mathrm{rc}}^{(\mathrm{h})}$ (resp. $F_{\mathrm{rc}}^{(\mathrm{h})}$). This means the adversary can distinguish y from a random value, which is not allowed by the RCPRF security definition. Using H cancels the relation.

Master key derivation from a seed. For all the above RCPRFs, a master key contains two parts: the *public part* containing non-secret parameters (*e.g.*, L, pk in the above RCPRFs) and the *secret part* containing secret states and keys (*e.g.*, s_L, s, and (sk, s_1)). In the above specification, KeyGen samples the secret parts using uniform randomness. Indeed, it's possible to determine the secret part from a λ-bits pseudorandom seed. Looking ahead, we will set up an RCPRF instance for each keyword on-the-fly in DISCO, where a pseudorandom seed (derived from another pseudorandom function) is used to configure the secret part for each RCPRF instance, and all instances share the common public parameters (*e.g.*, L, pk). We defer the details to §4.2.

Trade-off discussions. Different RCPRFs enjoy distinct properties. We identify several properties in terms of efficiency and usability that will be vital in the design of DISCO.

- *Key size*. Key size refers to the size of the master/constrained key). The master key size is constant for three RCPRFS. In terms of constrained keys, both the hash-based and TDP-based RCPRFs require constant size, while the GGM-based RCPRF requires $O(\log L)$ elements of λ bits (*i.e.*, the BRC nodes).
- *Constrained key generation*. The hash-based RCPRF requires $L - c$ hash invocations for constrained key generation, the TDP-based requires c TDP inverse computation, and the GGM-based requires $O(\log L)$ PRG invocations.
- *Batch evaluation*. The batch computation overhead corresponds to the overhead for evaluating a batch of RCPRF evaluation (*i.e.*, RCPRF evaluation over a range $[c]$). Concretely, both hash-based and GGM-based RCPRFs require $O(c)$ invocations, though the

hash-based RCPRF performs hash computation and the TDP-based RCPRF invokes TDP inverse. The GGM-based RCPRF performs $O(c)$ PRG invocations.

- *Domain size*. This refers to whether an RCPRF requires explicitly defining an upper bound L for the RCPRF evaluation domain. We note that both the hash-based and GGM-based RCPRFs only support a pre-defined evaluation domain, while the TDP-based RCPRF supports an unbounded (polynomial) domain.

A summary is shown in Table 2. Looking ahead, when utilizing RCPRFs for designing DSE, the master key is stored by the client, which contributes to the client-side local storage. Constrained key generation contributes to the client-side computational overhead. The constrained key is sent to the server to perform constrained evaluation, contributing to client-to-server communication. Batch RCPRF evaluation contributes to server-side computation during a search. We leave more details in the next section.

Table 2: Main efficiency properties of RCPRF schemes.

Scheme	Con. Key Size	Key Gen. Comp.	Bat. Eval. Comp.	Unb. rang.
$F_{\text{rc}}^{(\text{h})}$	$O(\lambda)$	$O(L-c)$	$O(c)$	No
$F_{\text{rc}}^{(\text{g})}$	$O(\lambda \log L)$	$O(c)$	$O(c)$	No
$F_{\text{rc}}^{(\text{t})}$	$O(\lambda)$	$O(c)$	$O(c)$	Yes

A hybrid RCPRF design. Beyond DSE, RCPRFs are useful in other privacy-preserving applications [3]. A pre-defined upper bound of evaluation may restrict usability for update-frequent applications. We also propose a hybrid unbounded RCPRF by combining good properties from different RCPRFs, which is of independent interest. Due to the page limit, we move the details to §C.

4 DSE WITH CONSTANT CLIENT STATE

This section details the DISCO framework, concrete instantiations, and optimizations.

4.1 Overview

Update fashion. DISCO works in the batch update setting, where the client periodically updates a batch of indexes to the server-side encrypted database. A batch update setting is reasonable in many scenarios, *e.g.*, when data updates arrive in a batch manner or the client catches a small number of updates locally before pushing them to the remote encrypted database. For example, the client may perform a daily batch update, *i.e.*, pushing updates (if they exist) to the server-side encrypted database at the end of each day.

Database organization. Following the batch update fashion, we can logically parse the server-side encrypted database $\mathbf{EDB} = \mathbf{EDB}^{(1)} \cup \mathbf{EDB}^{(2)} \cup \cdots \cup \mathbf{EDB}^{(c)}$ as a union of the sub batched encrypted databases, where $\mathbf{EDB}^{(i)}$ is the encrypted indexes from the i-th batch. In DISCO, the encrypted database contains a collection of encrypted index pairs with the form (u, e), where u is the reference/address, e is the encrypted content, and e can be efficiently fetched using u.

Index structure. DISCO aims to achieve forward-and-backward privacy efficiently only using a constant client-side state. Before introducing the high-level idea of DISCO, we first overview the high-level index structure in previous DSE works.

Index structure in previous works. We will use a toy example to elaborate on the index structure. Without loss of generality, suppose we have two keywords w_1 and w_2 in the system, and the client issues three batches of updates:

- $\mathsf{IN}^{(1)}$: $\{[w_1 : \{(\mathsf{add}, \mathsf{id}_1), (\mathsf{add}, \mathsf{id}_2)\}]; [w_2 : \{(\mathsf{add}, \mathsf{id}_1)\}]\}$;
- $\mathsf{IN}^{(2)}$: $\{[w_2 : \{(\mathsf{del}, \mathsf{id}_1), (\mathsf{add}, \mathsf{id}_3), (\mathsf{add}, \mathsf{id}_4)\}]\}$;
- $\mathsf{IN}^{(3)}$: $\{[w_1 : \{(\mathsf{add}, \mathsf{id}_5)\}]; [w_2 : \{(\mathsf{add}, \mathsf{id}_5)\}]\}$

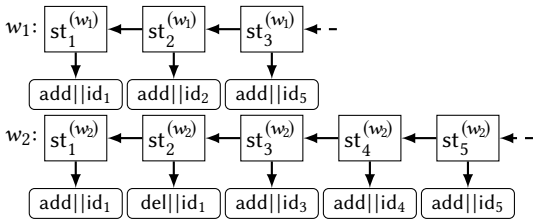


Figure 8: The index data structure using $O(|\mathsf{W}|)$ state.

The index structure of previous works is sketched in Fig. 8. Despite slight differences, the key idea is how to evolve secret tokens during updates and how to reveal tokens to the server to enable search under the constraint of forward privacy. This is done by the client locally maintaining an updatable state (*e.g.*, a counter) *per* keyword. The client generates a fresh secret token for each new update and uses the new token to encrypt the update. Only the client can evolve the token list for a new token (using client-known secrets and states) such that the server receiving the new encrypted index cannot link it with previous queries. To enable search, the client only reveals the secret tokens confined to the update history of the keyword to be searched, but not the tokens used for future updates; this ensures forward privacy.

Index structure in DISCO. Our goal is to design a different index structure only using a constant state. The index structure is sketched in Fig. 9. Now we overview its high-level design.

First, we remove the $O(|\mathsf{W}|)$ state using a global counter as considered in [17]. Instead of using $O(|\mathsf{W}|)$ counters, the client uses one single global counter c shared among all keywords, and the client increases c *once* per update batch.

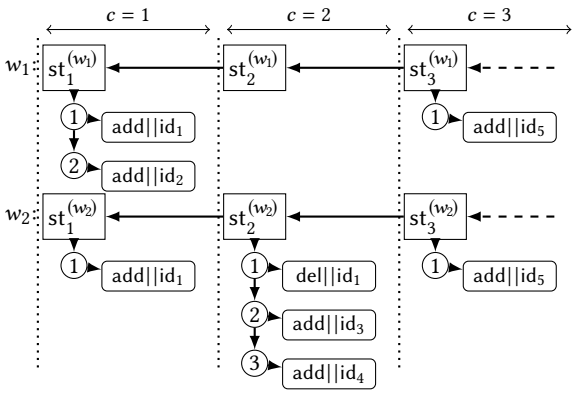


Figure 9: Index structure in DISCO using $O(1)$ state.

Second, DISCO re-uses a secret token for a keyword w to encrypt all updated indexes associated with w within a batch. Note that a secret token in Fig. 8 is only for one single update. Differently, DISCO takes a secret token for a keyword w with a sub-counter to

encrypt indexes corresponding to w. For example, the client uses $\mathsf{st}_1^{(w_1)}$ to handle both $(\mathsf{add}, \mathsf{id}_1)$ and $(\mathsf{add}, \mathsf{id}_2)$ for w_1 when $c = 1$.

Another issue is how to efficiently evolve secret tokens during an update and perform restricted token revealing during a search. We resort to RCPRF for evolving and controlling ranged secrets. We apply RCPRFs to the global counter setting to get rid of $O(|\mathsf{W}|)$ client state. As a promising property, DISCO permits different RCPRF configurations. Indeed, one can regard CLOSE-FB as a concrete instantiation within the DISCO framework, yet DISCO can provide flexible trade-offs when configured with different RCPRFs, opening the possibility of efficiency improvement.

An issue that comes with using a global counter is that the server must search through each sub-database to find all possible matched indexes, which increases computational overhead, particularly for search queries with a small number of matched indexes. We propose practical optimizations to alleviate the issue without downgrading security. We refer to §4.3 for the details.

Remark. The above index structure is only for conceptual understanding. We will encrypt the index securely and properly in DISCO.

4.2 The DISCO Framework

Following the overview, this section details the DISCO framework and its concrete instantiations. We also propose optimizations to improve DISCO's efficiency concretely.

Generic constant-state DSE framework. Fig. 10 presents the pseudocode of DISCO. We show details of update and search protocols as follows.

Update. Give an update batch IN containing possibly many updates, DISCO encrypts the indexes only using a constant number of keys. To achieve forward privacy, DISCO implicitly maintains an RCPRF instance for each keyword, and the client computes the RCPRF master key $\mathsf{k}_\mathsf{r}^{(w)} \leftarrow F_1(\mathsf{k}_1, w)$ for w on the fly using a normal PRF F_1.[1] Using the master RCPRF key $\mathsf{k}_\mathsf{r}^{(w)}$, the client derives a fresh secret token $\mathsf{st}_c^{(w)}$ corresponding to c, and uses $\mathsf{st}_c^{(w)}$ to perform index encryption. Within an update batch, it's possible to have multiple document-keyword pairs for the same keyword. DISCO employs a simple counter-based index encryption trick to encrypt these indexes corresponding to the same keyword w (line. 8-11, Update). The client uploads $\mathbf{EDB}^{(c)}$, completing the update.

Search. To search a keyword w, the client only needs to release the constrained key $\mathsf{ck}_c^{(w)}$ (for the RCPRF instance associated with the keyword w being searched) for range $[c]$. Using $\mathsf{ck}_c^{(w)}$, the server can recover all PRF values $\{\mathsf{st}_i\}_{i\in[c]}$. Note that the server uses each $\mathsf{st}_i^{(w)}$ to perform a search over $\mathbf{EDB}^{(i)}$ to find all the matched indexes. The server performs the encrypted search and returns a collection of encrypted items to the client. The client simply decrypts the encrypted values and removes deleted ids to obtain $\mathbf{DB}(w)$; here we rely on *result-hiding* to achieve type-II backward privacy, which is a standard technique from [2].

Concrete DISCO **instantiations**. While some ideas from DISCO are already discovered by previous DSE works [1, 2, 17], DISCO combines these techniques together as a new and simple framework for designing constant-state FB-DSE. Moreover, we can configure DISCO with different RCPRFs to obtain configurable trade-offs.

DISCO from hash-based RCPRF. To concretely configure DISCO with $F_{\mathrm{rc}}^{(\mathrm{h})}$, the client simply uses $\mathsf{k}_\mathsf{r}^{(w)}$ as an initial seed to setup a hash chain for w. In particular, the client uses $\mathsf{k}_\mathsf{r}^{(w)}$ as the seed to determine the secret part (*i.e.*, $\mathsf{st}_L^{(w)}$) and computes the hash chain on the fly whenever needed. The public parameter contains L and the hash function H for the hash chain.

We can interpret CLOSE-FB as a concrete instantiation of DISCO using the hash-based RCPRF. However, both $\mathsf{DISCO}_\mathsf{h}$ and CLOSE-FB suffer from efficiency/usability issues due to using hash chains. In particular, the client must specify an upper bound L for the size of the hash chains. After running up the hash chains (*i.e.*, $c = L$), CLOSE-FB requires the client to download all the encrypted indexes, decrypt them, and perform database rebuilding using new hash chains. To avoid database rebuilding too frequently, it's better to use hash chains with large L. However, we stress that the client must perform $L - c$ invocations of hash evaluation for updating or searching one single keyword, thus a large L will significantly increase client-side computation overhead; it seems that the hash-based construction is only desired for DSE with low update frequency.

DISCO *from GGM-based RCPRF*. We can instantiate the RCPRF with the GGM-based RCPRF, and we call the resulting DISCO construction $\mathsf{DISCO}_\mathsf{g}$. To concretely configure DISCO with $F_{\mathrm{rc}}^{(\mathrm{g})}$, the client simply uses $\mathsf{k}_\mathsf{r}^{(w)}$ as an initial seed of the GGM tree for w. In particular, the client uses $\mathsf{k}_\mathsf{r}^{(w)}$ as the seed to derive a GGM tree on the fly whenever needed. The public parameter contains tree size L and the PRG G for the GGM trees.

As a benefit of using GGM-based PPRF, the client-side computation is $O(\log L)$ during search/update, which is sublinear in L and is much more efficient than $\mathsf{DISCO}_\mathsf{h}$ for large L. Though $\mathsf{DISCO}_\mathsf{g}$ still requires a pre-specified upper bound L like $\mathsf{DISCO}_\mathsf{h}$, the client-side computation overhead is reduced significantly for large L. Note that the constrained key for GGM-based RCPRF is $O(\log L)$ BRC nodes, whereas the constrained key size of hash-based RCPRF is one single value. Overall, $\mathsf{DISCO}_\mathsf{g}$ requires much less computation overhead than $\mathsf{DISCO}_\mathsf{h}$ for large L, with a slightly increased client-to-server communication from the constrained key.

DISCO *from TDP-based RCPRF*. We can configure DISCO with $F_{\mathrm{rc}}^{(t)}$, and we call the resulting DISCO construction $\mathsf{DISCO}_\mathsf{t}$. The client simply uses $\mathsf{k}_\mathsf{r}^{(w)}$ to determine the initial seed $s_1^{(w)}$. The client sets up a pair of public/secret key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \pi.\mathsf{Gen}(1^\lambda)$. We note that the client can use this key pair across all RCPRF instances.

While the TDP-based RCPRF is generally computationally more expensive than prior symmetric-based alternatives, the efficiency gap is not as significant as expected in practice because in most cases the bottleneck of SSE schemes is not computation. Also, the TDP-based RCPRF has a distinguishing property: it supports an unbounded (polynomial-size) domain without any restriction. We leverage this property in the design of the hybrid RCPRF constructions in §C. Using this hybrid RCPRF, we can achieve both low computational overhead and unbounded evaluation domain; resolving issues from running out hash chains in CLOSE-FB [17].

[1] We abuse the notation without explicitly specifying the public parameters of the master key, and the secret part is determined by a λ-bits pseudorandom value generated by F_1. We note all RCPRFs formalized in this paper support this seed-setup property and will show in detail when configuring DISCO with a concrete RCPRF scheme.

$\underline{\mathsf{Setup}(1^\lambda; \perp)}$

Client:

1: $\mathsf{k}_1 \leftarrow \{0,1\}^\lambda$, $\mathsf{k}_2 \leftarrow \mathsf{SE.Gen}(1^\lambda)$, $c \leftarrow 0$
2: Store $\sigma \leftarrow (\mathsf{k}_1, \mathsf{k}_2, c)$

Server:

3: $\mathbf{EDB} \leftarrow \emptyset$

$\underline{\mathsf{Update}(\sigma, \mathsf{IN}; \mathbf{EDB})}$

Client:

1: Parse σ as $(\mathsf{k}_1, \mathsf{k}_2, c)$
2: Update $c \leftarrow c + 1$.
3: $\mathbf{EDB}^{(c)} \leftarrow \emptyset$
4: **for** $w \in \mathsf{W}(\mathsf{IN})$ **do**
5: $\quad \mathsf{k}_\mathrm{r}^{(w)} \leftarrow F_1(\mathsf{k}_1, w)$, $\mathsf{st}_c^{(w)} \leftarrow F_\mathrm{rc}(\mathsf{k}_\mathrm{r}^{(w)}, c)$
6: $\quad j \leftarrow 1$
7: $\quad$ **for** $(\mathsf{op}, w, \mathsf{id}) \in \mathsf{IN}(w)$ **do**
8: $\quad\quad u \leftarrow H_1(\mathsf{st}_c^{(w)} || j)$
9: $\quad\quad e \leftarrow \mathsf{SE.Enc}(\mathsf{k}_2, \mathsf{op}||\mathsf{id})$
10: $\quad\quad \mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u, e)\}$
11: $\quad\quad j \leftarrow j + 1$
12: Send $\mathbf{EDB}^{(c)}$ to the server

Server:

16: $\mathbf{EDB} \leftarrow \mathbf{EDB} \cup \mathbf{EDB}^{(c)}$

$\underline{\mathsf{Search}(\sigma; \mathbf{EDB})}$

Client:

1: Parse σ as $(\mathsf{k}_1, \mathsf{k}_2, c)$
2: $\mathsf{k}_\mathrm{r}^{(w)} \leftarrow F_1(\mathsf{k}_1, w)$
3: $\mathsf{ck}_c^{(w)} \leftarrow F_\mathrm{rc}.\mathsf{Constrain}(\mathsf{k}_\mathrm{r}^{(w)}, [c])$
4: Send $(\mathsf{ck}_c^{(w)}, c)$ to the server

Server:

5: $\mathbf{E} \leftarrow \emptyset$
6: $\{\mathsf{st}_i^{(w)}\}_{i \in [c]} \leftarrow F_\mathrm{rc}.\mathsf{BEval}(\mathsf{ck}_c^{(w)}, [c])$
7: **for** $i \in [c]$ **do**
8: $\quad j \leftarrow 1$, $u \leftarrow H_1(\mathsf{st}_c^{(w)} || j)$
9: $\quad$ **while** $\mathbf{EDB}^{(i)}[u] \neq \perp$ **do**
10: $\quad\quad e \leftarrow \mathbf{EDB}^{(i)}[u]$, $\mathbf{E} \leftarrow \mathbf{E} \cup \{e\}$
11: $\quad\quad j \leftarrow j + 1$, $u \leftarrow H_1(\mathsf{st}_c^{(w)} || j)$
12: Send $\mathbf{E}$ to the client

Client:

13: Decrypt $\mathbf{E}$ using k_2 and remove deleted identifiers to obtain $\mathbf{DB}(w)$.

Figure 10: The pseudocode of DISCO.

Security. DISCO achieves forward privacy and type-II backward privacy. Intuitively, DISCO achieves forward privacy from the security of RCPRF since the server cannot learn future secret tokens from a constrained key. The search result is encrypted using a PCPA-secure symmetric-key encryption and only the client can decrypt it to learn plaintexts; this result-hiding property is a standard technique to obtain type-II BP. We refer to §D for the proof.

Summary. The main idea in DISCO is to trade a low client state with the price of increased computation, and previous work [11] trades a low client state with the price of increasing communication. The trade-off from DISCO is more desired because communication, especially round complexity, is more likely to be a bottleneck for cloud-based applications. Also, DISCO mainly relies on lightweight cryptographic operations. The increased computational overhead is mainly on the server side, and the client-side computation is almost the same as the previous lightweight DSE with $O(|\mathsf{W}|)$ state. Our following experiment results show that the increased computational overhead is almost unnoticeable from human perception. Moreover, DISCO is conceptually simple, and simplicity is also one important principle when designing new protocols and applications.

4.3 DISCO$^+$: Optimized DISCO

Using a global counter achieves a constant client state but incurs more computational overhead. To alleviate this issue, we propose optimizations to improve DISCO's efficiency concretely; we denote the optimized version as DISCO$^+$. Fig. 12 shows the formal description, and Fig. 11 sketches optimizations on the index perspective.

Enable parallel search. DISCO$^+$ additionally supports parallel search. We modify the index encryption method in DISCO to support parallel search. Roughly speaking, the idea is to somehow let the server learn the number of matched indexes before performing an actual search, then the server can allocate multi-thread to perform a parallel search if necessary (*e.g.*, for a search query with many matched indexes). To support this, we modify the update protocol by additionally encrypting the counter $j = |\mathsf{IN}(w)|$ as a part of the encrypted indexes (line. 12-15, Update), which can only be decrypted when the server learns $\mathsf{st}_c^{(w)}$ during a search. By knowing $j = |\mathbf{DB}_c(w)|$, the server can perform a parallel search to find out all matched encrypted indexes.

Reduce RCPRF evaluation. DISCO incurs additional overhead from redundant RCPRF invocations as the server must search over all sub-databases to find matched indexes. For keywords with few matchings, the extra RCPRF evaluation may dominate computation.

To reduce the overhead, the server can perform a post-search re-keying strategy to re-organize the encrypted indexes after a search, reducing RCPRF evaluation to the same keyword in a future search query; similar tricks can be found from previous works [1, 12, 21, 26]. In particular, the server can re-use the latest ephemeral token $\mathsf{st}_c^{(w)}$ to re-generate new (pseudorandom) addresses for the encrypted indexes $\mathbf{E}$ after each search and update the re-encrypted items back to $\mathbf{EDB}^{(c)}$ (line. 19-26, Search); the server also removes old indexes during the search to save storage. As a result, the server has no need to perform RCPRF evaluation over $i < c$ for the next search to the same keyword.

Following the intuition, DISCO$^+$ additionally stores one bit of information indicating whether the server should stop during a search. Specifically, the client encrypts a bit $b \in \{0, 1\}$ along with the counter $j = |\mathsf{IN}(w)|$, where $b = 0$ tells the server to stop. Initially, the client sets $b = 1$ for all the keywords. Once a keyword is searched, the server updates $b \leftarrow 0$ during the post-search re-keying phase. By doing this, the server can know when to stop searching in a future search query to the same keyword (line. 17, Search), reducing RCPRF evaluation overhead. Fig. 11 shows an example from keyword w_1.

Enable database merging. For applications with small update batches, the client can merge small encrypted sub-databases together periodically to reduce the overhead from RCPRF evaluation. When necessary, the client can periodically download several

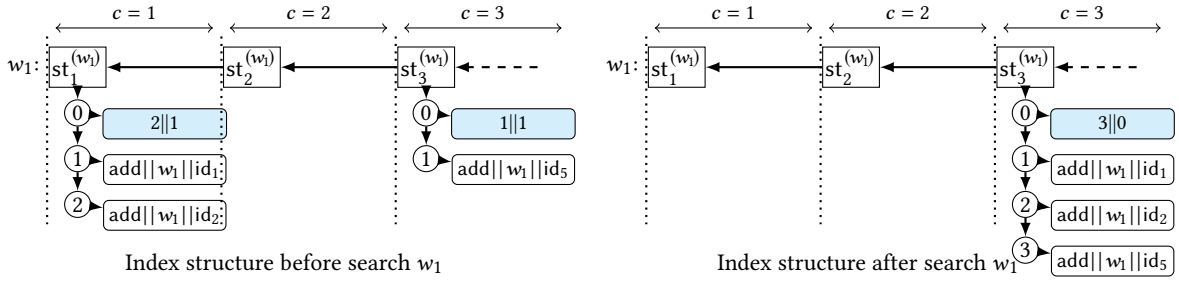
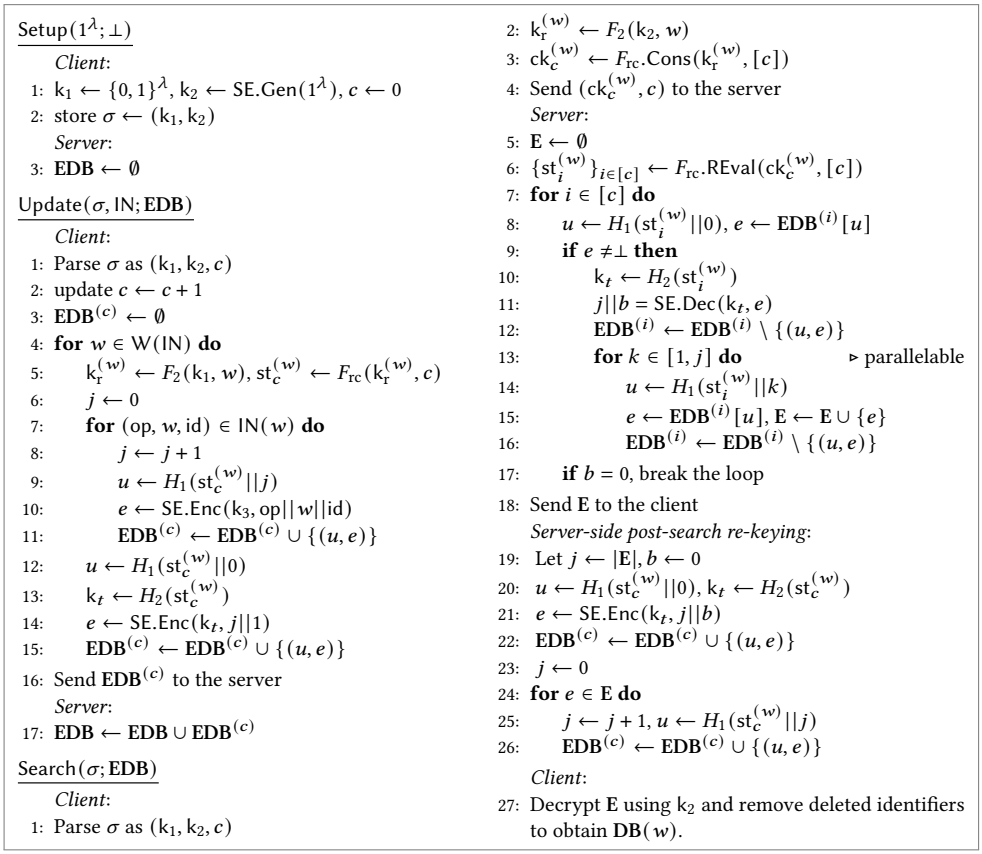


Figure 11: The (sketched) index data structure in DISCO$^+$. The structure and information are reorganized after searching w_1.



$\underline{\text{Setup}(1^\lambda; \perp)}$

Client:

1: $\mathsf{k}_1 \leftarrow \{0,1\}^\lambda$, $\mathsf{k}_2 \leftarrow \mathsf{SE.Gen}(1^\lambda)$, $c \leftarrow 0$
2: store $\sigma \leftarrow (\mathsf{k}_1, \mathsf{k}_2)$

Server:

3: $\mathbf{EDB} \leftarrow \emptyset$

$\underline{\text{Update}(\sigma, \mathsf{IN}; \mathbf{EDB})}$

Client:

1: Parse σ as $(\mathsf{k}_1, \mathsf{k}_2, c)$
2: update $c \leftarrow c + 1$
3: $\mathbf{EDB}^{(c)} \leftarrow \emptyset$
4: **for** $w \in \mathsf{W}(\mathsf{IN})$ **do**
5: $\quad \mathsf{k}_\mathrm{r}^{(w)} \leftarrow F_2(\mathsf{k}_1, w)$, $\mathsf{st}_c^{(w)} \leftarrow F_{\mathrm{rc}}(\mathsf{k}_\mathrm{r}^{(w)}, c)$
6: $\quad j \leftarrow 0$
7: $\quad$ **for** $(\mathsf{op}, w, \mathsf{id}) \in \mathsf{IN}(w)$ **do**
8: $\quad\quad j \leftarrow j + 1$
9: $\quad\quad u \leftarrow H_1(\mathsf{st}_c^{(w)} || j)$
10: $\quad\quad e \leftarrow \mathsf{SE.Enc}(\mathsf{k}_3, \mathsf{op}||w||\mathsf{id})$
11: $\quad\quad \mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u, e)\}$
12: $\quad u \leftarrow H_1(\mathsf{st}_c^{(w)} || 0)$
13: $\quad \mathsf{k}_t \leftarrow H_2(\mathsf{st}_c^{(w)})$
14: $\quad e \leftarrow \mathsf{SE.Enc}(\mathsf{k}_t, j||1)$
15: $\quad \mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u, e)\}$
16: Send $\mathbf{EDB}^{(c)}$ to the server

Server:

17: $\mathbf{EDB} \leftarrow \mathbf{EDB} \cup \mathbf{EDB}^{(c)}$

$\underline{\text{Search}(\sigma; \mathbf{EDB})}$

Client:

1: Parse σ as $(\mathsf{k}_1, \mathsf{k}_2, c)$
2: $\mathsf{k}_\mathrm{r}^{(w)} \leftarrow F_2(\mathsf{k}_2, w)$
3: $\mathsf{ck}_c^{(w)} \leftarrow F_{\mathrm{rc}}.\mathsf{Cons}(\mathsf{k}_\mathrm{r}^{(w)}, [c])$
4: Send $(\mathsf{ck}_c^{(w)}, c)$ to the server

Server:

5: $\mathbf{E} \leftarrow \emptyset$
6: $\{\mathsf{st}_i^{(w)}\}_{i \in [c]} \leftarrow F_{\mathrm{rc}}.\mathsf{REval}(\mathsf{ck}_c^{(w)}, [c])$
7: **for** $i \in [c]$ **do**
8: $\quad u \leftarrow H_1(\mathsf{st}_i^{(w)} || 0)$, $e \leftarrow \mathbf{EDB}^{(i)}[u]$
9: $\quad$ **if** $e \neq \perp$ **then**
10: $\quad\quad \mathsf{k}_t \leftarrow H_2(\mathsf{st}_i^{(w)})$
11: $\quad\quad j||b = \mathsf{SE.Dec}(\mathsf{k}_t, e)$
12: $\quad\quad \mathbf{EDB}^{(i)} \leftarrow \mathbf{EDB}^{(i)} \setminus \{(u, e)\}$
13: $\quad\quad$ **for** $k \in [1, j]$ **do** ▷ parallelable
14: $\quad\quad\quad u \leftarrow H_1(\mathsf{st}_i^{(w)} || k)$
15: $\quad\quad\quad e \leftarrow \mathbf{EDB}^{(i)}[u]$, $\mathbf{E} \leftarrow \mathbf{E} \cup \{e\}$
16: $\quad\quad\quad \mathbf{EDB}^{(i)} \leftarrow \mathbf{EDB}^{(i)} \setminus \{(u, e)\}$
17: $\quad$ **if** $b = 0$, break the loop
18: Send $\mathbf{E}$ to the client

Server-side post-search re-keying:

19: Let $j \leftarrow |\mathbf{E}|$, $b \leftarrow 0$
20: $u \leftarrow H_1(\mathsf{st}_c^{(w)} || 0)$, $\mathsf{k}_t \leftarrow H_2(\mathsf{st}_c^{(w)})$
21: $e \leftarrow \mathsf{SE.Enc}(\mathsf{k}_t, j||b)$
22: $\mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u, e)\}$
23: $j \leftarrow 0$
24: **for** $e \in \mathbf{E}$ **do**
25: $\quad j \leftarrow j + 1$, $u \leftarrow H_1(\mathsf{st}_c^{(w)} || j)$
26: $\quad \mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u, e)\}$

Client:

27: Decrypt $\mathbf{E}$ using k_2 and remove deleted identifiers to obtain $\mathbf{DB}(w)$.

Figure 12: The pseudocode of DISCO$^+$. Here H_2 is for generating a key k_t used for encrypting counter information. To make the encrypted counter ciphertext indistinguishable from normal indexes from size, padding is done whenever necessary.

batches of encrypted databases, decrypt them, and encrypt them as a whole new batch. The client uploads the encrypted indexes (using $\mathsf{st}_c^{(w)}$ associated with a new fresh c) to the server, and the server can delete the old ones. To enable rebuilding, we modify the index encryption method by adding the keyword within the encrypted indexes (line. 10, Update); this allows the client to fully recover all the $(\mathsf{op}, w, \mathsf{id})$ tuples after local decryption. The client removes the deleted identifiers during rebuilding to save server-side storage.

Security. Similiar as previous works [12, 21, 22, 26, 32], optimizations in DISCO$^+$ don't introduce more leakage than DISCO. Therefore, we only provide security proof for DISCO in this paper.

5 EXPERIMENTAL EVALUATION

5.1 Implementation and Setting

We implemented DISCO (https://github.com/BintaSong/openssse-schemes) in C++. We use OpenSSL for cryptographic operations, Rocksdb (http://rocksdb.org) for database storage, and gRPC (http://www.grpc.io) for network communication. The identifier length is 64-bit. We run evaluation on a PC with four cores Intel(R) Core(TM) i7-10850 CPU, 2.7GHz, 16GB RAM, and 256G SSD disk, running Ubuntu 16.04. We use Linux tc tool to simulate local-area network (LAN, RTT: 0.1 ms, 1 Gbps), wide-area network (WAN, RTT: 6 ms, 100 Mbps).

We use DB(n, m, N) to denote a database with n documents, m keywords, and N document/keyword pairs. We follow a DB generation method from OpenSSE(https://github.com/OpenSSE/opensse-schemes/blob/master/lib/utils/db_generator.cpp) used by previous works [1, 2], and generate three databases DB1, DB2, DB3 in Table 3, and the Enron dataset is between DB2 and DB3. We set the upper bound L for DISCO_h (CLOSE-FB) and DISCO_g in our experiment. For example, setting $L = 2^{14}$ supports more than 44 years if the client performs a daily update fashion.

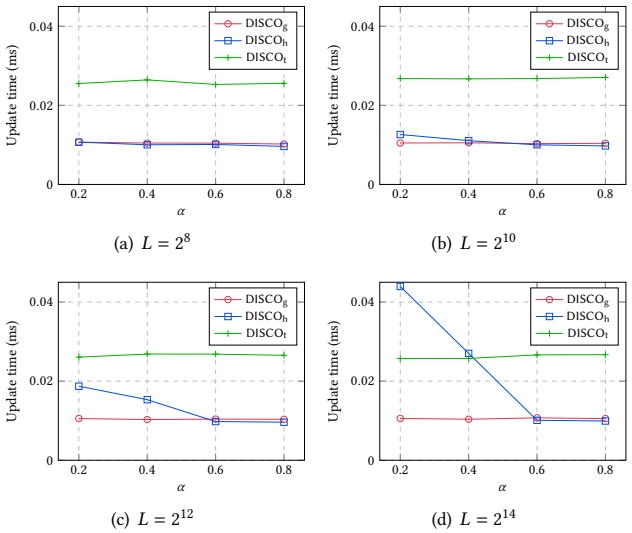
Table 3: Database size

Database	n	m	N
DB1	100, 000	23, 050	1, 737, 895
DB2	1, 000, 000	229, 835	18, 212, 888
DB3	10, 000, 000	2, 315, 889	189, 516, 363
Enron	517, 080	390, 423	22, 900, 317

Storage. We measure the client-side and server-side Rocksdb storage. DB3, Sophos, FAST, and Mitra require about 500MB of client storage for storing counter information, while the encrypted database on the server side is about 6GB. DISCO requires $\leq$ 2KB permanent client storage without blowing up server-side storage.

5.2 Update Performance

Comparison between different DISCO **constructions**. To show the trade-offs between DISCO constructions under different c and L, we report update efficiency under different $c = \alpha \cdot L$, where $\alpha \in [0, 1]$ is a configurable parameter called *update rate*.



(a) $L = 2^8$ (b) $L = 2^{10}$ (c) $L = 2^{12}$ (d) $L = 2^{14}$

Figure 13: Update time for DISCO **constructions in the LAN setting.**

We first report update efficiency and discuss trade-offs from different DISCO constructions. In these settings, we vary $L \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$ and $c = \alpha \cdot L$ where $\alpha \in \{0.2, 0.4, 0.6, 0.8\}$. We update DB1 into the server-side database and measure the atomized update time for each document/keyword pair. Fig. 13 shows the update efficiency for DISCO under different L and α. The performance evaluation is consistent with the complexity properties of different DISCO constructions. In particular, we have the following findings.

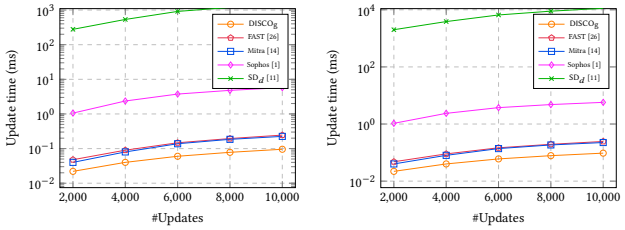
DISCO_h *ranks the best for large enough* α. DISCO_h is the least efficient scheme for sufficiently large L, consistent with complexity analysis, i.e., performing $L - c$ hash invocations for secret token generation. DISCO_h is computationally bounded for large L and small α. When α approaches 1, the computational overhead is not the bottleneck anymore; in this case, DISCO_h ranks at the top.

DISCO_t *outbeats* DISCO_h *for large* L *and small* α. Note DISCO_t requires relatively inefficient asymmetric computation. Though DISCO_t generally has the worst update efficiency, it still beats DISCO_h for large L and small α. As shown in Fig. 13(d), DISCO_t is about 1.7× faster than DISCO_h when $L = 2^{14}$ and $\alpha = 0.2$. This is because DISCO_h performs worse if $L - c$ is large.

DISCO_g *has the most stable update efficiency*. We can see that DISCO_g owns the most stable update efficiency among the three constructions, achieving about 0.01ms/pair update efficiency for any L and α. Even for large α, DISCO_g is only 0.1× slower than DISCO_h. DISCO_h owns the most unstable update efficiency.

Comparison with previous DSE constructions. We compare update efficiency with previous DSE schemes. We choose DISCO_g (we set $L = 2^{14}$ and $\alpha = 0.5$) for comparison since DISCO_g is the most stable version among DISCO constructions. We compare with Sophos, FAST, Mitra, and SD_d.[2] We do not compare with Moneta, Orion, and Horus that use oblivious primitive like SD_d; since SD_d is more efficient than them according to [11].

Running time. We first compare update time in Fig. 14. For each construction, we update $u \in \{2000, 4000, 6000, 8000, 10000\}$ indexes into the server-side encrypted database and measure the total time to complete u updates in the LAN and WAN setting, respectively.



(a) LAN setting (b) WAN setting

Figure 14: Update time for different DSE schemes.

As we can see, DISCO_g requires the least running time, or equivalently, the best update throughput, among all schemes. The reason is that DISCO_g batches all updates to the same keyword and re-uses the secret token derived from the RCPRF to encrypt these indexes. In previous DSE schemes, the client computes a fresh secret token for each update index regardless of whether the same keyword is updated in the same batch. As a result, DISCO_g is about 2× faster

[2]The original implementation of SD_d holds encrypted indexes in RAM without network communication, which cannot show the impact of network condition. To make the comparison fair, we use the version with network from https://github.com/MonashCybersecurityLab/SDd. We note this version still has an I/O advantage over other DSE schemes since it still uses RAM to hold encrypted indexes.

than the previous fully symmetric DSE scheme FAST and Mitra.[3] SD_d has the worst update throughput among all reported DSEs. There is no wonder since SD_d is based on OMAP, which requires $O(\log N)$ rounds and $O(\log^3 N)$ communication per update. Overall, SD_d is about three orders of magnitude slower even compared with Sophos, which owns the lowest update efficiency among the remaining DSE schemes. Another finding is that the update efficiency of SD_d is enlarged in the WAN setting. In particular, SD_d's update throughput is about 7.5× slower in WAN compared with the LAN setting. This is because SD_d's update protocol is not constant round; thus, network latency is more influential to its update efficiency. Like previous DSE schemes Sophos, FAST, and Mitra, $DISCO_g$ enjoys constant round update protocol; thus, its update efficiency only downgrades slightly in the WAN setting.

Communication. We report the total communication for each scheme in Table 4, where the number of updated indexes is from {2000, 4000, 6000, 8000, 10000} and we measure the communication using Linux command IPTraf-ng. We note that DISCO enjoys the same update communications as previous DSE constructions Sophos, FAST, and Mitra; all DISCO constructions own the same update communication complexity. SD_d requires two orders of magnitude more communication than DISCO. The reason is that SD_d entails a "download-merge-upload" update fashion which requires significantly more communication.

Table 4: Update communication (MB) for different schemes.

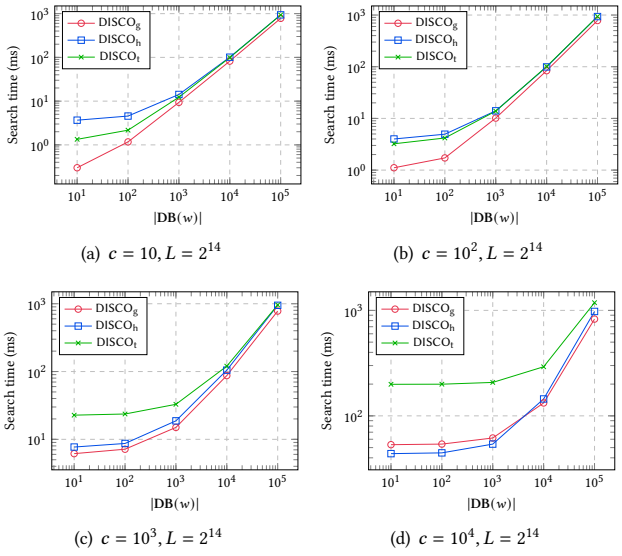
#Updates	DISCO	Sophos	FAST	Mitra	SD_d
2000	0.49	0.49	0.59	0.49	47.94
4000	0.97	0.97	1.18	0.97	103.87
6000	1.46	1.46	1.77	1.46	163.45
8000	1.95	1.95	2.36	1.95	223.91
10000	2.43	2.43	2.94	2.43	287.66

Summary of update efficiency. All DISCO constructions enjoy optimal update communication. Computation-wise, $DISCO_h$ owns the worst efficiency for large L and small α, but the best efficiency occurs when α is large. $DISCO_g$ enjoys the most balanced efficiency and $DISCO_t$ provides unbounded evaluation domain.

5.3 Search Performance

Performance comparison of DISCO **constructions**. We report the search performance (running time and communication) of three DISCO constructions. Due to the page limit, we move the detailed performance to §B, elaborating the trade-offs between different DISCO constructions. Here, we plot the running time in Fig. 15, where each sub-figure reports the search running time for three DISCO constructions under different values of global counter c.

$DISCO_h$ owns the worst search efficiency for small c. As we discussed, $DISCO_h$ requires the client to compute $L - c$ hash invocations to compute and reveal a search token $st_c^{(w)}$ to the server, and the server receiving the token must perform c invocations of the hash function to perform a search. This means $DISCO_h$ performs L hash invocations independent of c. One may wonder why $DISCO_h$ performs well when c is large since the total hash invocations are always L. The reason is that we use two threads for the server-side search process in $DISCO_h$'s implementation: the first is to recover all search tokens from the hash chain using the constrained key, and the second is to perform I/O to fetch desired indexes using the tokens from the first thread. Since I/O contributes to the main overhead during the search (also observed from [1, 22, 26]), the computation overhead of server-side RCPRF evaluation is covered by I/O. Therefore, when c approaches L, the total search running time is reduced. $DISCO_h$'s search running time is the worst among all DISCO's constructions for small cs, even compared with $DISCO_t$.



(a) $c = 10, L = 2^{14}$ (b) $c = 10^2, L = 2^{14}$

(c) $c = 10^3, L = 2^{14}$ (d) $c = 10^4, L = 2^{14}$

Figure 15: Search running time for different DISCO **instantiations over DB3 (LAN setting).**

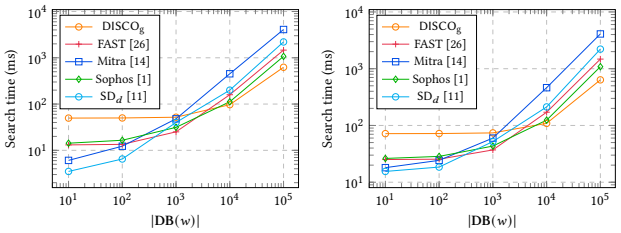
$DISCO_g$ has the most stable search efficiency. $DISCO_g$ owns the most stable search efficiency among three DISCO constructions. In particular, $DISCO_g$ is 10× and 4.5× faster than $DISCO_h$ and $DISCO_t$, respectively, when $|\mathbf{DB}(w)| = 10$ and $c = 10$. For large c, $DISCO_h$ is only about 1.2× faster than $DISCO_g$ when $|\mathbf{DB}(w)| = 10$ and $c = 10$; note that $DISCO_h$ has the best advantage in this case.

DISCO enjoy good search communication. $DISCO_h$ and $DISCO_t$ enjoy $O(a_w)$ communications, which is the same as previous DSE works [1, 2, 12, 26], while $DISCO_g$ only slightly increases the communication due to its non-constant constrained key. In practice, different DISCO constructions have very closed communication (less than 1KB difference when $a_w = 10^5$).

Performance comparison with previous DSE works. We compare DISCO with previous works on search efficiency. As we know, the general idea of DISCO is to trade constant client state and low communication with the price of increasing running time (due to redundant RCPRF evaluation). We show that this increased computational overhead is small and acceptable.

Fig. 16 shows the search time for different DSE schemes in the LAN and WAN settings, respectively. As we can see, $DISCO_g$ does require more running time for search queries with a small result set (*e.g.*, $|\mathbf{DB}(w)| = 10$ and 100), which is about 10× slower than SD_d.

[3] This shows that secret-reuse does improve efficiency. Therefore, the same batch update trick can be used to optimize previous schemes.



(a) LAN setting (b) WAN setting

Figure 16: Search running time of different DSE schemes.

However, the running time is unnoticeable from a human perspective: DISCO$_g$ requires less than 80ms (resp. 100ms) to complete a search query in the LAN (resp. WAN) setting when $|\mathbf{DB}(w)| = 10$ and 100. As the number of matched identifiers increases, the running time of DISCO$_g$ performs even better than other schemes. The reason is that DISCO$_g$ entails a parallel search optimization. In the implementation, the server can allocate one thread for RCPRF evaluation and another thread to perform the search. Because the search efficiency is I/O bound, the running time from RCPRF evaluation is totally covered by the I/O overhead. Also, the server can allocate multiple threads to perform I/O since the indexes organization of DISCO$_g$ totally supports it. All these optimizations result in that DISCO$_g$ achieves the best search efficiency when searching keywords with a large number of matched identifiers.

Summary of search efficiency. Communication-wise, all DISCO constructions enjoy (near) the same communication (*i.e.*, $O(a_w)$) as previous works [1, 14, 26]; only DISCO$_g$ requries slightly more communication (*i.e.*, $O(a_w + \log L)$). Computation-wise, DISCO$_h$ owns the worst efficiency for large L and small α, but the best efficiency when α is large. DISCO$_g$ enjoys the most balanced efficiency.

6 RELATED WORK

Dynamic searchable encryption. Searchable encryption (SE) was first proposed by the seminal work [25]. Curtmola *et al.* [10] formalized the most widely accepted security model for SE. Dynamic searchable encryption (DSE) allows the client to update the remote encrypted database after setup. Rather than sequentially scanning the whole database [25], Kamara *et al.* [20] proposed a DSE scheme atop the work [10], achieving sub-linear search time. The following works [4–6, 19, 23] further improved the efficiency and functionality of SSE schemes. Early DSE schemes cannot achieve forward privacy. Forward privacy was first formalized by [29]. Bost [1] proposed an elegant work to achieve forward privacy using trapdoor permutation. A serial of the following works [1, 2, 12, 14, 21, 26, 31] showed that forward privacy can be achieved efficiently using symmetric primitives. Backward privacy was first considered by Stefanov *et al.* [29] and formally defined by Bost *et al.* [2] with three types of backward privacy. Recent works [8, 30, 31] proposed new efficient (symmetric) primitives to achieve backward privacy with improved efficiency. Some DSE works [9, 24, 27] leveraging a multi-server paradigm achieve better efficiency with less leakage. There is another line of works [7, 11, 14, 24] design DSE atop oblivious primitives. These works achieve advanced security properties and/or asymptotically efficient DSE schemes, but their practical efficiency can be worse than prior DSE schemes based on lightweight primitives.

DSE with small client storage. Reducing client storage is vital in real-world applications. Existing lightweight DSE schemes require a client to maintain the state for each keyword to achieve forward privacy efficiently, thwarting feature-rich usage (e.g., video retrieval [35]). The ORAM-based DSE scheme Moneta [2], Orion [14], and Horus [14] do not require client-side storage, the price is increased round and/or communication complexity. Without using ORAM, Demertzis *et al.* [11] proposed a construction using *static-to-dynamic* transform that transforms any static data structure to be dynamic. The idea is that the client downloads and merges several databases together when necessary. The *download-merge-upload* process causes a cascade effect, saying $O(N)$ communication for one update in the worst case. The authors also proposed a lazy rebuilding strategy based on Oblivious Map to resolve the issue by reducing the worst $O(N)$ communication, requiring $O(\log^3 N)$ communication overhead per update pair. He *et al.* [17] use hash-chains to design small-client-storage DSE with linear computation in the length of the hash chain. Their works inspired our framework using RCPRF over a global counter. Since DISCO supports different RCPRF configurations, it achieves flexible trade-offs than [17].

7 CONCLUSION

We propose an efficient DSE framework DISCO using a constant client state. DISCO employs RCPRF to generate and release the search tokens. By instantiating RCPRF with different techniques, we design three DISCO constructions. All three constructions require a constant client state with different trade-offs. We evaluate the performance of DISCO, showing their improved efficiency.

Future work. While DISCO only requires constant client-side permanent storage, it still requires temporary storage to catch updates. For a client with little storage (*e.g.*, IoT sensors), the client may increase the global counter frequently; this can increase server-side computational overhead during a search query. Reducing the increment of the global counter in such extreme cases is still an open problem. Another direction is exploring DISCO in the multi-device setting, where a user owns multiple devices, and each may perform updates. How to leverage DISCO into such a scenario efficiently and securely in a user-friendly way requires further investigation.

ACKNOWLEDGEMENT

This research is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative and the National Natural Science Foundation of China under Grant 62302118, Grant 62072132, and Grant 62261160651. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

REFERENCES

[1] Raphael Bost. 2016. Σοφος: Forward secure searchable encryption. In *ACM SIGSAC Conference on Computer and Communications Security*. 1143–1154.

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

[3] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. 2020. {TimeCrypt}: Encrypted data stream processing at scale with cryptographic access control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 835–850.

[4] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. *Network and Distributed System Security Symposium* (2014).

[5] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cuatualin Rocsu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual Cryptology Conference*. 353–373.

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

[7] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. 2022. Dynamic searchable encryption with optimal search in the presence of deletions. In *USENIX Security*. 2425–2442.

[8] Tianyang Chen, Peng Xu, Wei Wang, Yubo Zheng, Willy Susilo, and Hai Jin. 2021. Bestie: Very practical searchable encryption with forward and backward security. In *European Symposium on Research in Computer Security, Part II 26*. 3–23.

[9] Shujie Cui, Xiangfu Song, Muhammad Rizwan Asghar, Steven D Galbraith, and Giovanni Russello. 2021. Privacy-preserving dynamic symmetric searchable encryption with controllable leakage. *ACM Transactions on Privacy and Security (TOPS)* 24, 3 (2021), 1–35.

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

[11] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2020. Dynamic Searchable Encryption with Small Client Storage. In *Annual Network and Distributed System Security Symposium*.

[12] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient Dynamic Searchable Encryption with Forward Privacy. *Privacy Enhancing Technologies* 1 (2018), 5–20.

[13] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*. 563–592.

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

[15] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM symposium on Theory of computing*. 182–194.

[16] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *Journal of the ACM (JACM)* 33, 4 (1986), 792–807.

[17] Kun He, Jing Chen, Qinxi Zhou, Ruiying Du, and Yang Xiang. 2020. Secure dynamic searchable symmetric encryption with constant client storage cost. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1538–1549.

[18] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. 2018. Structured encryption and leakage suppression. In *Annual International Cryptology Conference, Part I 38*. 339–370.

[19] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *FC*.

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

[21] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward secure dynamic searchable symmetric encryption with efficient updates. In *ACM SIGSAC Conference on Computer and Communications Security*. 1449–1463.

[22] Russell WF Lai and Sherman SM Chow. 2017. Forward-secure searchable encryption on labeled bipartite graphs. In *International Conference on Applied Cryptography and Network Security*. 478–497.

[23] Feng Li, Jianfeng Ma, Yinbin Miao, Pengfei Wu, and Xiangfu Song. 2023. Beyond Volume Pattern: Storage-Efficient Boolean Searchable Symmetric Encryption with Suppressed Leakage. In *European Symposium on Research in Computer Security*. Springer, 126–146.

[24] Zheli Liu, Yanyu Huang, Xiangfu Song, Bo Li, Jin Li, Yali Yuan, and Changyu Dong. 2020. Eurus: towards an efficient searchable symmetric encryption with size pattern protection. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 2023–2037.

[25] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *IEEE symposium on security and privacy. S&P 2000*. 44–55.

[26] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. 2018. Forward private searchable symmetric encryption with optimized I/O efficiency. *IEEE Transactions on Dependable and Secure Computing* 17, 5 (2018), 912–927.

[27] Xiangfu Song, Dong Yin, Han Jiang, and Qiuliang Xu. 2020. Searchable Symmetric Encryption with Tunable Leakage Using Multiple Servers. In *International Conference on Database Systems for Advanced Applications*. Springer, 157–177.

[28] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[29] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*.

[30] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph K Liu, Surya Nepal, and Dawu Gu. 2021. Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy. In *Network and Distributed System Security Symposium*.

[31] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *ACM SIGSAC Conference on Computer and Communications Security*. 763–780.

[32] Jiafan Wang and Sherman SM Chow. 2022. Forward and Backward-Secure Range-Searchable Symmetric Encryption. *Privacy Enhancing Technologies* 1 (2022), 28–48.

[33] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *ACM SIGSAC Conference on Computer and Communications Security*. 215–226.

[34] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: the power of {File-Injection} attacks on searchable encryption. In *USENIX Security Symposium (USENIX Security)*. 707–720.

[35] Yu Zheng, Heng Tian, Minxin Du, and Chong Fu. 2022. Encrypted video search: Scalable, modular, and content-similar. In *ACM Multimedia systems Conference*. 177–190.

A ADDITIONAL DEFINITIONS

Definition A.1. A TDP π is secure if for all PPT adversary $\mathcal{A}$, $\mathcal{A}$ successfully computes an inverse of $y = \pi_{\mathsf{pk}}(x)$ for a uniformly sampled $x \xleftarrow{\$} \mathcal{D}_{\mathrm{t}}$ without the secret key is negligible in λ, *i.e.*, $\mathsf{Adv}^{\mathsf{TDP}}_{\mathcal{A},\pi}(\lambda) = |\Pr[x' = x : x \xleftarrow{\$} \mathcal{D}_{\mathrm{t}}, y \leftarrow \pi_{\mathsf{pk}}(x), x' \leftarrow \mathcal{A}(\mathsf{pk}, y)]| \leq \mathsf{negl}(\lambda)$.

Definition A.2. A symmetric encryption scheme $\mathsf{SE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is secure against pseudo-randomness chosen-plaintext attack (PCPA) if for all PPT adversary $\mathcal{A}$, $\mathcal{A}$ has a negligible advantage in λ, where the advantage is defined as $\mathsf{Adv}^{\mathsf{PCPA}}_{\mathcal{A},\mathsf{SE}}(\lambda) = |\Pr[\mathbf{Exp}^{\mathsf{PCPA}}_{\mathcal{A},\mathsf{SE}}(\lambda) = 1] - 1/2|$ and $\mathbf{Exp}^{\mathsf{PCPA}}_{\mathcal{A},\mathsf{SE}}(\lambda)$ is an experiment between an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and a challenger in Fig. 18.

B SUPPLEMENTAL EXPERIMENTS

Table 5 details running time and communication under the LAN.

C HYBRID RCPRF CONSTRUCTION

We can combine good properties from different RCPRFs using a hybrid construction method. In a nutshell, a hybrid RCPRF is composed of two (or more) instances of RCPRF: the upper layer and the lower layer. The upper layer is for deriving a group of master keys, and each key parameterizes a low-level RCPRF. We exploit a *re-keying* property of all existing RCPRFs. In particular, we can choose the master key pseudorandomly, and define new RCPRF instances from the chosen key. Following the intuition, our framework just uses an upper-level RCPRF to derive the master keys for the lower level and initiates a lower-level RCPRF using the derived master keys. In this manner, we can obtain good properties that cannot be achieved from one single kind of RCPRF.

As an example, Fig. 19 shows a two-layer RCPRF from TDP-based RCPRF and the GGM-based RCPRF, where the TDP-based RCPRF serves as the upper layer and the GGM-based RCPRF as the

Table 5: Running time and communication for three DISCO's instantiations.

		Running time (ms)				Communication (MB)			
		$c = 10$	$c = 10^2$	$c = 10^3$	$c = 10^4$	$c = 10$	$c = 10^2$	$c = 10^3$	$c = 10^4$
DISCO_g	$a_w = 10$	0.30	1.11	6.15	53.29	0.003	0.003	0.003	0.003
	$a_w = 10^2$	1.17	1.72	7.15	53.98	0.008	0.008	0.008	0.008
	$a_w = 10^3$	9.37	10.15	14.99	61.67	0.056	0.056	0.056	0.056
	$a_w = 10^4$	82.23	83.88	87.20	132.52	0.530	0.527	0.527	0.532
	$a_w = 10^5$	780.88	783.20	780.17	827.03	5.122	5.310	5.305	5.303
DISCO_h	$a_w = 10$	3.67	4.00	7.70	43.82	0.003	0.003	0.003	0.003
	$a_w = 10^2$	4.56	4.95	8.69	44.57	0.008	0.008	0.008	0.008
	$a_w = 10^3$	14.21	14.02	18.74	53.96	0.056	0.056	0.056	0.056
	$a_w = 10^4$	101.02	99.16	104.83	144.61	0.533	0.533	0.533	0.533
	$a_w = 10^5$	936.87	933.61	948.24	977.31	5.298	5.298	5.298	5.298
DISCO_t	$a_w = 10$	1.34	3.23	22.73	199.16	0.004	0.004	0.004	0.004
	$a_w = 10^2$	2.17	4.18	23.64	199.83	0.009	0.009	0.009	0.009
	$a_w = 10^3$	12.30	13.82	32.85	207.48	0.056	0.056	0.056	0.056
	$a_w = 10^4$	99.24	101.06	120.02	292.11	0.532	0.532	0.532	0.532
	$a_w = 10^5$	933.89	936.86	959.16	1187.99	5.307	5.307	5.307	5.307

$\textbf{Exp}^{\text{rcprf}}_{\mathcal{A},F}(\lambda)$

1: $b \xleftarrow{\$} \{0,1\}; E, Q, Z \leftarrow \emptyset$
2: $\mathsf{k} \leftarrow F_{\text{rc}}.\text{KeyGen}(1^\lambda)$
3: $(\{x_i\}_{i\in[n]}, st) \leftarrow \mathcal{A}_1^{O_\mathsf{k}^{\text{Eval}}(\cdot), O_\mathsf{k}^{\text{Constrain}}(\cdot)}(1^\lambda)$
4: $Z \leftarrow Z \cup \{x_i\}_{i\in[n]}$
5: For $i \in [0]$, $y_i^{(0)} \xleftarrow{\$} \mathcal{R}_\text{r}$; $y_i^{(1)} \leftarrow F(\mathsf{k}, x_i)$
6: $b' \leftarrow \mathcal{A}_2(st, \{y_i^{(b)}\}_{i\in[n]})$
7: **return** $(b = b' \wedge Z \cap E = \emptyset \wedge Z \cap Q = \emptyset)$

Oracle $O_\mathsf{k}^{\text{Eval}}(x)$
1: $y \leftarrow F_{\text{rc}}(\mathsf{k}, x)$
2: $E \leftarrow E \cup \{x\}$
3: **return** y

Oracle $O_\mathsf{k}^{\text{Constrain}}(c)$
1: $\mathsf{ck}_c \leftarrow F_{\text{rc}}.\text{Constrain}(\mathsf{k}, [c])$
2: $Q \leftarrow Q \cup [c]$
3: **return** ck_c

Figure 17: RCPRF security game. Here $n = \text{poly}(\lambda)$ and st contains all constrained keys and RCPRF evaluation result from querying $O_\mathsf{k}^{\text{Constrain}}(\cdot)$ and $O_\mathsf{k}^{\text{Eval}}(\cdot)$, respectively.

$\textbf{Exp}^{\text{PCPA}}_{\mathcal{A},\text{SE}}(\lambda)$

1: $b \xleftarrow{\$} \{0,1\}; \mathsf{k} \leftarrow \text{SE.Gen}(1^\lambda)$
2: $(m^*, st) \leftarrow \mathcal{A}_1^{O_\mathsf{k}^{\text{Enc}}(\cdot)}(1^\lambda)$
3: If $b = 0$, $ct^* \leftarrow \text{SE.Enc}(\mathsf{k}, m^*)$;
4: Else $ct^* \xleftarrow{\$} C$
5: $b' \leftarrow \mathcal{A}_2^{O_\mathsf{k}^{\text{Enc}}(\cdot)}(st, ct^*)$
6: **return** $(b = b')$

Oracle $O_\mathsf{k}^{\text{Enc}}(x)$
1: $y \leftarrow \text{SE.Enc}(\mathsf{k}, x)$
2: **return** y

Figure 18: PCPA security of symmetric encryption.

lower layer. The resulting RCPRF enjoys good properties of two involved RCPRFs: supporting unbounded RCPRF evaluation (from the TDP-based RCPRF) while most of the evaluation only involves symmetric evaluation (from the GGM-based RCPRF). This hybrid design enables trade-offs between computation, communication,

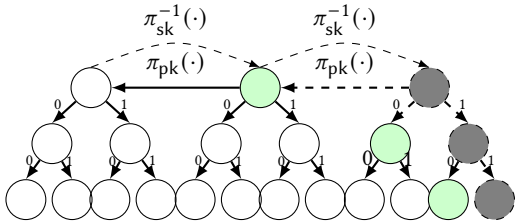


Figure 19: RCPRF from TDP-based and GGM-based RCPRF. Green nodes correspond to a constrained key range [11].

and usability. In particular, we still mainly use the GGM-based RCPRF for key derivation and only use the TDP-based RCPRF over the root to derive a fresh root when updates run out of the previous GGM tree. In terms of the constrained key, its size corresponds to the constrained key from the third GGM tree and the second node in the TDP-based RCPRF (*i.e.*, the green nodes in Fig. 19). Overall, the key size is still $O(\log L)$.

One can generally extend the above idea to other RCPRFs, *e.g.*, combining hash-based RCPRF with the TDP-based one.

D SECURITY PROOF FOR DISCO

Theorem D.1. *Let F be a pseudorandom function, F_{rc} be a range-constrained pseudorandom function, H_1 be a hash function modeled as random oracles, and* SE *be a semantic secure symmetric encryption. Define leakage $\mathcal{L} = (\mathcal{L}_{Setup}, \mathcal{L}_{Search}, \mathcal{L}_{Update})$ as:*

$$\mathcal{L}_{Setup}(\perp) = (\perp)$$
$$\mathcal{L}_{Search}(w) = (sp(w), TimeDB(w), Updates(w))$$
$$\mathcal{L}_{Update}(\text{IN}) = (|\text{IN}|, \{\text{op}_i, \text{id}_i\}),$$

then DISCO is an $\mathcal{L}$-adaptively-secure DSE with forward privacy and backward privacy (BP-II).

Proof. We define a sequence of games and prove Theorem D.1 by showing the distinguishability between $\textbf{Real}^\Pi_\mathcal{A}(\lambda)$ and $\textbf{Ideal}^\Pi_{\mathcal{A},\mathcal{S}}(\lambda)$ is negligible in λ.

Hybrid G_1: G_1 is the same as $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ except that the experiment maintains a map $\mathbf{K}$ to store $k_r^{(w)}$ for each keyword w instead of generating $k_r^{(w)}$ using F_1. Whenever $k_r^{(w)}$ is needed, the experiment first checks whether $\mathbf{K}$ contains $(w, k_r^{(w)})$, if so returns the entry $\mathbf{K}$; otherwise the experiment randomly picks a $k_r^{(w)} \xleftarrow{\$} \{0,1\}^\lambda$ and stores the $(w, k_r^{(w)})$ pair in $\mathbf{K}$. It's easy to see that G_0 and $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ are indistinguishable, otherwise we can build an adversary $\mathcal{B}_1$ to distinguish F from a truly random function, thus we have $\Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)=1] - \Pr[G_1=1] \le \mathsf{Adv}^{\mathrm{prf}}_{F,\mathcal{B}_1}(\lambda)$.

Hybrid G_2: G_2 is the same as G_1 except that instead of computing $u \leftarrow H_1(st_c^{(w)}||j)$, the client executes: $u \xleftarrow{\$} \{0,1\}^\ell, \mathbf{U}[w||c||j] \leftarrow u$, where $\mathbf{U}$ is a map maintained by the experiment. Correspondingly, the next change is to the search protocol. Instead of performing $u \leftarrow H_1(st_c^{(w)}||j)$, G_2 programs the random oracle H_1 *before* search:

$j \leftarrow 0, u \leftarrow \mathbf{U}[w||c||j]$
while $u \neq \perp$ **do**
  $\mathbf{H}_1[st_c^{(w)}||j] \leftarrow u$
  $j \leftarrow j+1, u \leftarrow \mathbf{U}[w||c||j]$

where $\mathbf{H}_1$ is the table for random oracle H_1.

G_2 and G_1 behave exactly the same except that in G_2, with a probability of inconsistency in querying random oracle that is observed by the distinguisher. In G_1, $\mathbf{H}_1$ is not updated immediately but programmed before the search. Therefore, if the adversary $\mathcal{A}$ queries H_1 with $st_c^{(w)}||j$ for some j before the next searching, $u' \neq u$ is returned with an overwhelming probability. This is from $\mathbf{H}_1[st_c^{(w)}||j]$ is not updated and a random string u' is chosen by the oracle in this case. For the following search, the adversary $\mathcal{A}$ queries $H_1(st_c^{(w)}||j)$. $H_1(st_c^{(w)}||j)$ is then updated to u, and u is returned as the query result of random oracle. The only difference between G_1 and G_2 is the above inconsistency (denoting this event as Bad). Thus, we have $\Pr[G_1=1] - \Pr[G_2=1] \le \Pr[\mathrm{Bad}]$.

We show that if $\mathcal{A}$ can make such inconsistent queries with non-negligible probability, then we can build adversary $\mathcal{B}_2$ to break the security of RCPRF with the same probability. In particular, $\mathcal{B}_2$ guess (w^*, c^*) for which Bad is set to be true for the first time when querying $H_1(st_{c^*}^{(w^*)}||j)$ for some j. $\mathcal{B}_2$ works as follows: When update to w^* for $i < c^*$ and any j is issued, $\mathcal{B}_2$ query $O^{\mathrm{Eval}}_{k_{w^*}}(i)$ to obtain $st_i^{(w^*)}$ and use $st_i^{(w^*)}$ to perform updates related to w^* in the i-th batch. For all updates corresponding to w^* in the i-th batch for $i > c^*$, $\mathcal{B}_2$ sends i as the challenge queries to $\mathbf{Exp}^{\mathrm{rcprf}}_{\mathcal{A},F}(\lambda)$ and receives back $st_i^{(w^*)}$, By definition of $\mathbf{Exp}^{\mathrm{rcprf}}_{\mathcal{A},F}(\lambda)$, $st_{i^*}^{(w^*)}$ is either 1) a random value or 2) the real RCPRF evaluation output.

Note that Bad is set to be true only if H_1 is queried on $(st_c^{(w^*)}, j)$ such that c is not queried over $O^{\mathrm{Eval}}_{k_{w^*}}(c)$, and no query $O^{\mathrm{Constrain}}_{k_{w^*}}(c)$ for $c > c^*$. This means all the challenge queries for $i > c^*$ are valid. The values $st_i^{(w^*)}$ for $i > c^*$ raising Bad to be true is indistinguishable from random by the security of RCPRF. Now consider $st_i^{(w^*)}$ for $i > c^*$ are uniformally randomly generated. If $\mathcal{A}$ queries random oracle for at most q times, then the probability H_1 was called on $st_c^{(w^*)}||j$ for some j is bounded by $q/2^\lambda$. Therefore, we have $\Pr[\mathrm{Bad}$ is set to true by quering $st_c^{(w^*)}||j$ for some $j] \le \mathsf{Adv}^{\mathrm{rcprf}}_{F_{\mathrm{CR}},\mathcal{B}_2}(\lambda) + q/2^\lambda$. Since guessing (w^*, c) implies a *at most* N loss in security reduction, then we have $\Pr[G_1=1] - \Pr[G_2=1] \le \Pr[\mathrm{Bad}] \le N \cdot \mathsf{Adv}^{\mathrm{rcprf}}_{F_{\mathrm{CR}},\mathcal{B}_2}(\lambda) + Nq/2^\lambda$.

Hybrid G_3: G_3 is the same as G_2 except that it replaces all ciphertexts generated by SE.Enc with random strings from the ciphertext domain $\mathcal{C}$. If G_2 is distinguishable from G_3, then we can build an adversary $\mathcal{B}_3$ to break the PCPA security of SE. Thus, we have $\Pr[G_2=1] - \Pr[G_3=1] \le \mathsf{Adv}^{\mathrm{pcpa}}_{\mathsf{SE},\mathcal{B}_3}(\lambda)$.

$\mathcal{S}.\mathsf{Setup}(\mathcal{L}_{\mathsf{Setup}}(\perp))$
1: $c \leftarrow 0$
2: $\mathbf{U}, \mathbf{K} \leftarrow$ empty map

$\mathcal{S}.\mathsf{Update}(\mathcal{L}_{\mathsf{Update}}(\mathsf{IN}))$
1: $\mathbf{EDB}^{(c)} \leftarrow \emptyset, c \leftarrow c+1$
2: **for** $0 \le j < |\mathsf{IN}|$ **do**
3: $u_j \xleftarrow{\$} \{0,1\}^\ell$
4: $\mathbf{U}[c,j] \leftarrow u_j$
5: $e_j \xleftarrow{\$} \mathcal{C}$
6: $\mathbf{EDB}^{(c)} \leftarrow \mathbf{EDB}^{(c)} \cup \{(u_j, e_j)\}$
7: send $\mathbf{EDB}^{(c)}$ to Server

$\mathcal{S}.\mathsf{Search}(\mathsf{sp}(w), \mathsf{Update}(w))$
1: $\underline{w} \leftarrow \min \mathsf{sp}(w)$
2: $k_r^{(\underline{w})} \leftarrow \mathbf{K}[\underline{w}]$
3: $ck_c^{(\underline{w})} \leftarrow F_{\mathrm{rc}}.\mathsf{Constrain}(k_r^{(\underline{w})}, [c])$
4: send $ck_c^{(\underline{w})}$ to Server
$\mathcal{S}$ *programs* H_1 *before searching:*
5: $\gamma \leftarrow |\mathsf{Updates}(w)|$
6: $\{(c_j, i_j)\}_{j\in[\gamma]} \leftarrow \mathsf{Updates}(w)$
7: **for** $1 \le j \le \gamma$ **do**
8: $st_{c_j}^{(\underline{w})} \leftarrow F_{\mathrm{rc}}.\mathsf{Eval}(ck_c^{(\underline{w})}, j)$
9: $\mathbf{H}_1[st_{c_j}^{(\underline{w})}||j] \leftarrow \mathbf{U}[c_j, i_j]$

Figure 20: Pseudocode of simulator

Hybrid G_4: G_4 is the same as G_3 except that it indexes $\mathbf{U}$ by $c||j, 0 \le j < |\mathsf{IN}|$ instead of indexing $\mathbf{U}$ by $w||c||j$. Specifically, we identify each update by (c, j), where c is the batch number and j denotes the update number in the c-th batch. It is easy to see G_5 and G_4 are indistinguishable. Hence, we have $\Pr[G_3=1] = \Pr[G_4=1]$.

$\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}$: The simulator accessing the allowed leakage functions needs to generate a viewing that is indistinguishable from G_4. Here, we consider the view of generating the index rather than the whole viewing such as file operation. We use $\underline{w} = \min \mathsf{sp}(w)$ to denote the first index that w appeared in search pattern. $\underline{w}$ is used as the identifier to identify a keyword without knowing the keyword information. Before performing search on the server, the simulator programs $\mathbf{H}_1$ by learning all updates corresponding to $\underline{w}$ of $\mathsf{Updates}(w)$. As we can see, $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}$ is identical to G_4 except that all needed information for simulation is from the leakage functions. Then, we get $\Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}=1] = \Pr[G_4=1]$. All sum up, we have:

$$\Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)=1] - \Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)=1] \le \mathsf{Adv}^{\mathrm{prf}}_{F,\mathcal{B}_1}(\lambda) + N \cdot \mathsf{Adv}^{\mathrm{rcprf}}_{F_{\mathrm{CR}},\mathcal{B}_2}(\lambda) + Nq/2^\lambda + \mathsf{Adv}^{\mathrm{pcpa}}_{\mathsf{SE},\mathcal{B}_3}(\lambda)$$

□