Corruption is Futile: Hardware-Assisted Multi-User Searchable Encryption

Feng Li, Chengyan Ma, Xiaoguo Li, Pengfei Wu, Lisha Yao, Jianting Ning, Guomin Yang, Jianfeng Ma, and Robert H. Deng, *Fellow, IEEE,*

***Abstract*—Searchable encryption enables writers to upload encrypted data to an untrusted cloud server, while allowing authorized readers to perform keyword searches over the encrypted content without compromising data privacy. The multi-writer/multi-reader (M/M) setting better reflects real-world deployment requirements because it aligns with the collaborative nature of many modern applications, but introduces substantial challenges in both efficiency and security. The state-of-the-art scheme, delegatable searchable encryption (DSE), achieves optimal search time and forward privacy in the M/M setting. Nevertheless, DSE provides only limited security guarantees: search tokens generated by readers reveal the update count of queried keywords, and the scheme remains susceptible to collusion attacks between users and the server.**

In this work, we present Archon, a novel hardware-assisted multi-user searchable encryption scheme that addresses the aforementioned security limitations while maintaining sublinear search complexity. By leveraging trusted execution environments and an updatable keyword private information retrieval protocol, Archon achieves strong collusion resistance and supports flexible user revocation in multi-user settings. We implement Archon and evaluate its performance on real-world datasets to demonstrate its practical efficiency.

***Index Terms*—Searchable encryption, multi-writer, multi-reader, collusion attack, user revocation.**

I. Introduction

In the era of Industry 4.0 [1, 2], the Industrial Internet of Things (IIoT) is transforming data generation, sharing, and analysis. Industrial devices such as sensors, cameras, and controllers are widely deployed across production environments to collect data and transmit it to cloud servers for storage and analysis. Engineers and technicians rely on this data to monitor system performance, optimize production processes, and make decisions. However, as sensitive industrial data is outsourced to untrusted cloud servers, data confidentiality becomes critical. Unauthorized access or data leakage not only threatens operational privacy but can also lead to severe financial and reputational losses. Therefore, secure data-sharing mechanisms are needed to allow authorized personnel to access and search encrypted data without compromising privacy.

Searchable encryption (SE) [3, 4] enables data owners (i.e., writers) to outsource encrypted data to an untrusted cloud server while allowing authorized users (i.e., readers) to search over the encrypted content. Existing SE techniques are developed under two cryptographic paradigms: searchable symmetric encryption (SSE) [5] and public key encryption with keyword search (PEKS) [6]. SSE provides efficient search in settings where the data owner and the user share a secret key, while PEKS allows writers to generate searchable ciphertexts under a reader's public key. From a system perspective, SE can further be categorized into single-writer/single-reader (S/S) [7], multi-writer/single-reader (M/S) [8], single-writer/multi-reader (S/M) [9], and multi-writer/multi-reader (M/M) [10] architectures. This work focuses on the M/M architecture, which is required by collaborative applications such as industrial data sharing, where many devices continuously upload data and multiple operators issue search queries.

However, neither a direct SSE-based nor a direct PEKS-based extension fully addresses the requirements of M/M searchable encryption. An SSE-based extension may require users to share secret keys or keyword states, which simplifies search but weakens access control and makes revocation costly. A PEKS-based extension naturally supports multiple writers, but each writer may need to generate reader-specific searchable ciphertexts, or the system may need to maintain reader-specific searchable structures. Such designs increase storage overhead and may require searches over multiple ciphertext collections or indices, leading to high retrieval latency and complicating user revocation. Furthermore, the M/M setting introduces additional collusion threats, including collusion between corrupted writers and the server, as well as collusion between corrupted readers and the server, which must be explicitly addressed in the security model.

Numerous searchable encryption schemes have been proposed for multi-user settings. Curtmola *et al.* [5] first proposed a searchable encryption scheme supporting multiple readers. Their construction relies on broadcast encryption and requires readers to share a secret key for generating search tokens. While revocation is straightforward in the non-collusive setting, collusion-aware revocation requires refreshing and redistributing the shared key material and reconstructing the

Feng Li, Chengyan Ma, Lisha Yao, Guomin Yang, Robert H. Deng are with the School of Computing and Information Systems, Singapore Management University, Singapore 188065. (e-mail: lifengvbg@gmail.com, chengyanma@smu.edu.sg, lishayao@smu.edu.sg, gmyang@smu.edu.sg, robertdeng@smu.edu.sg). Xiaoguo Li is with the College of Computer Science, Chongqing University, Chongqing 400044, China. (e-mail: csxgli@cqu.edu.cn). Pengfei Wu is with National Engineering Research Center for Software Engineering, Peking University, Beijing 100870, China, with School of Software and Microelectronics, Peking University, Beijing 102600, China, and with Beijing Key Laboratory of Data Intelligence and Security, Peking University, Beijing 102600, China. (e-mail: wpf9808@gmail.com). Jianting Ning is with the Zhejiang Key Laboratory of Digital Fashion and Data Governance, Zhejiang Sci-Tech University, Hangzhou 310018, China, with the Laboratory of Cryptography and Blockchain Technology, Wuhan University, Wuhan 430072, China, and with the Faculty of Data Science, City University of Macau, Macau 999078, China. (e-mail: jtning88@gmail.com). Jianfeng Ma is with the School of Cyber Engineering, Xidian University, Xi'an 710071, China. (e-mail: jfma@mail.xidian.edu.cn).

affected index entries, thereby introducing additional key-management and system-maintenance overhead. Subsequent schemes [11, 12] assign distinct keys to different users. However, such designs may incur high search overhead and can expose honest readers' query privacy when corrupted readers collude with the server. Chamani *et al.* [13] and Wang *et al.* [9] considered collusion resistance by generating a separate index for each reader. While this approach effectively mitigates collusion between readers and the server, it increases server-side storage overhead and search latency, and does not naturally scale to the M/M setting.

Motivated by these limitations and the practical requirements of the M/M architecture, this paper aims to design a multi-writer/multi-reader searchable encryption scheme that simultaneously supports flexible user enrollment and revocation, efficient search, forward privacy, and corruption-resistant privacy against user-server collusion. Table I compares representative multi-user SE schemes, and the following subsection details our design goals and challenges.

A. Design Goals and Challenges

As discussed, the M/M setting arises in a wide range of practical application scenarios, necessitating careful consideration of both search efficiency and data privacy in system design. We outline the main design goals and key challenges addressed in this work below.

Flexible Enrollment and Revocation. A fundamental yet often overlooked requirement in the M/M setting is flexible user management, including both enrollment and revocation. Specifically, user enrollment and revocation. Despite its practical significance, this aspect remains underexplored in prior works [10, 14], largely due to the trade-off between computational efficiency and privacy preservation. From a key-management perspective, existing approaches can be broadly categorized into shared-key and distinct-key schemes. The shared-key setting enables efficient operations and directly inherits desirable features from SSE schemes. However, it exposes severe privacy risks: users may learn the search and update keywords of others, and revoking a user necessitates costly index reconstruction and key redistribution.

At first glance, assigning distinct keys to individual users appears to be a natural solution to these problems. Yet, this approach introduces another challenge: each piece of data must be encrypted separately for each reader, degrading the system into multiple parallel instances of PEKS and incurring significant storage and computational overhead.

Efficient Search. A straightforward way to extend S/M or M/S schemes to the M/M setting is to let each writer or reader maintain a separate index. However, this incurs substantial computational and storage overheads: a query may need to search multiple indices, or the same document may need to be encrypted and inserted into multiple distinct indices.

A more practical goal is to let all users share a global index. This requires supporting updates and queries from users with different cryptographic keys, while preserving the privacy of inserted data and searched keywords. Enabling secure and efficient cross-user interaction over a shared index is therefore fundamental to scalable M/M SE.

Forward Privacy. Dynamic SE enables writers to update the encrypted database by generating update tokens. While this flexibility is crucial for real-world deployments, it also introduces a new class of attacks, injection attacks [16, 17]. In such attacks, an adversary injects crafted documents into the database and observes whether subsequent search tokens match the newly inserted content, thereby inferring the queried keywords. This vulnerability has made forward privacy a *de facto* standard for dynamic SE schemes [18, 19]. Forward privacy ensures that past search queries do not compromise the privacy of future updates.

To achieve forward privacy, most dynamic SE constructions require users to maintain a secret state, which evolves over time and is used to generate fresh update tokens. However, in the M/M architecture, maintaining and synchronizing this state across multiple independent writers poses a significant challenge. The situation becomes even more complex when considering malicious or compromised writers. If an adversary corrupts one writer and obtains its internal state, the privacy of updates made by other honest writers is at risk. Thus, achieving forward privacy in a multi-writer setting requires careful coordination and robust protection mechanisms.

Corruption Resistant Privacy. We consider adversarial settings in which users may be corrupted. In practical deployments, IIoT devices acting as writers are often vulnerable to external attacks, while human operators acting as readers may be coerced or incentivized to collude with adversaries. In such cases, corrupted users can exploit their privileged knowledge to infer the private data of honest users. The threat becomes even more severe when corrupted users collude with the server.

The server inherently learns the locations of data uploaded by all writers, as well as the query access patterns of all readers, i.e., the exact locations accessed during search operations. If a colluding reader issues a query overlapping with the access patterns of honest users, the server can correlate these accesses and infer sensitive associations between keywords and documents, thereby violating user privacy. This highlights the need for strong protection against collusion between corrupted users and the server, especially in systems designed for multi-user environments with untrusted infrastructure.

B. Architectural Rationale for Hardware Assistance

Achieving the above design goals using purely cryptographic techniques alone is inherently challenging and often leads to complex and inefficient constructions. Dynamic revocation requires that different users hold distinct secret keys, while efficient search over a shared dataset necessitates that all users operate on a single global index. This, in turn, requires different users to generate semantically consistent search and update tokens for the same keyword. These requirements are fundamentally in tension: enforcing per-user key separation conflicts with the need for globally consistent token generation. Furthermore, forward privacy requires maintaining per-keyword update state across multiple users. Synchronizing such state in a multi-writer/multi-reader setting introduces substantial communication overhead. More critically, if this state is directly accessible to users, corrupted users may leverage it to facilitate collusion attacks with the server.

TABLE I: Comparison of Multi-User Searchable Encryption Schemes

Scheme	Architecture	Sym./Asym. Scenario	Collusion Resistance	Revocation	Dynamic/ Static	Forward Privacy	Search Round	Search Efficiency	Access/Search Pattern
CGK+06 [5]	S/M	Sym.	✗	✓	Static	N/A	2	$\mathcal{O}(\lvert D(w)\rvert)$	✗
YLW11 [12]	S/M	Asym.	✗	✓	Dynamic	✗	1	$\mathcal{O}(n)$	✗
WP21 [9]	S/M	Sym.	✓	✗	Static	N/A	2	$\mathcal{O}(\lvert D(w)\rvert + \log^2 n)^\dagger$	✗
CWP+23[13]	S/M	Sym.	✓	✗	Dynamic	✓	2	$\mathcal{O}(\lvert D(w)\rvert + \log^2 n)^\dagger$	✗
WC22 [14]	M/S	Asym.	✗	✗	Dynamic	Epoch-based	1	$\mathcal{O}(\lvert D(w)\rvert \cdot \lvert\mathcal{W}\rvert \cdot \lvert S\rvert)^\ddagger$	✗
LH25 [15]	M/S	Asym.	✗	✗	Dynamic	Epoch-based	1	$\mathcal{O}(\lvert D(w)\rvert \cdot \varsigma\sqrt{\lvert\mathcal{W}\rvert} \cdot \lvert S\rvert)^\ddagger$	✗
BDD+08 [11]	M/M	Asym.	✗	✓	Dynamic	✗	1	$\mathcal{O}(n)$	✗
WC24 [10]	M/M	Asym.	✗	✗	Dynamic	✓	1	$\mathcal{O}(\lvert D(w)\rvert)$	✗
Ours	M/M	Asym.	✓	✓	Dynamic	✓	1	$\mathcal{O}(\lvert\mathcal{W}\rvert + \lvert D(w)\rvert \cdot \xi)$	✓

Sym./Asym. denotes the symmetric/asymmetric search setting, classified according to whether writers and readers use the same key to generate searchable ciphertexts and search tokens, respectively. $|D(w)|$ is the number of documents containing the keyword w. n is the database size. † indicates that each reader has an independent index. ‡ indicates that each writer has an independent index. *Epoch-based* means the scheme provides forward privacy across different epochs, but not within the same epoch. In scheme [14], each writer rebuilds the encrypted token set at the start of a new epoch, but this cost is excluded from our complexity analysis. $\mathcal{W}$ denotes the set of universal keywords. S denotes the set of writers to be searched. ς indicates the maximum number of supported epochs is $2^\varsigma - 1$. Let $\xi = (M + Q + \log c + c)$ denote the cost of a single Eureka query operation, including client-side hint generation and server-side parity reconstruction.

To resolve these tensions, we introduce hardware assistance. Our work leverages a trusted execution environment (TEE) as a stateful trusted mediator. First, to reconcile flexible revocation with efficient search, the TEE acts as a token-transformation mediator that converts user-generated randomized tokens into consistent effective tokens for the shared global index. Specifically, we split the master secret key into two complementary shares: one assigned to the user and the other securely stored inside the TEE. A user can generate only a randomized token using its own key share, while the TEE transforms this token into an effective token applicable to the global index. Upon revocation, the TEE removes the corresponding user entry or key share, thereby preventing the revoked user from generating valid effective tokens without affecting other users or reconstructing the index. Second, to resolve the tension between multi-user state synchronization and privacy under corruption, the TEE maintains and updates the sensitive keyword states internally. Therefore, all search and update operations must be processed through the TEE before interacting with the encrypted data. This design isolates the global state from users, ensuring that even if a user is corrupted, the adversary cannot obtain global keyword state information or complete key material.

In principle, a traditional trusted third party (TTP) could perform a similar role. However, a TTP-based solution introduces additional trust assumptions and deployment burdens. A remote TTP incurs extra network round trips for every operation, requires institutional trust, and may become a bottleneck or single point of failure. In contrast, a co-located TEE operates within the server's hardware boundary, provides hardware-enforced isolation with remote attestation, and confines the trusted computing base to a minimal execution environment.

C. Our Contributions

In this work, we propose Archon[1], a hardware-assisted searchable encryption construction designed for the M/M architecture. Archon addresses critical challenges in user scalability, security, and efficiency, achieving strong privacy guarantees without imposing heavy user-side burdens. Our main contributions are as follows:

- Archon enables flexible user enrollment and revocation by assigning user-specific cryptographic keys. The index structure remains unaffected by membership changes. Leveraging a TEE, the system securely authenticates users and manages index generation over a shared global index. Revoked users are cryptographically prevented from performing valid update or search operations, even with access to the encrypted database.
- Archon ensures forward privacy while eliminating the need for users to maintain or synchronize secret state. Search pattern privacy is preserved even against corrupted readers, who gain no advantage in inferring honest readers' queries from their own search histories.
- We introduce a new keyword private information retrieval protocol that supports dynamic updates while hiding access patterns. This scheme enables privacy-preserving searches over a mutable database and forms the foundation for Archon's robust server-side privacy, even under collusion between the cloud server and corrupted users.
- We formalize the syntax and security definitions for hardware-assisted multi-user searchable encryption, and prove that Archon achieves adaptive security and forward privacy under user corruption. We implement a full prototype, evaluate its performance on real-world datasets.

II. Related Work

Hybrid SE (HSE), introduced by Wang *et al.* [14], seeks to combine the advantages of SSE and PEKS within a M/M architecture. In HSE, each writer maintains a separate index and uses identity-coupling key-aggregate encryption to enable a reader to access multiple indices with a compact token size. Furthermore, HSE introduces an epoch-based forward privacy notion, ensuring that updates cannot be identified by search tokens generated in past epochs, while updates within the same epoch remain searchable. However, HSE is vulnerable to keyword guessing attacks (KGA), and requires each writer to re-encrypt all search tokens at the start of a new epoch. Le *et al.* [15] improved HSE by introducing hidden-identity coupling key-aggregate encryption to mitigate KGA

[1]Archon stands for hardware-assisted searchable encryption scheme, symbolizing secure and coordinated multi-party management.

and partitioning search tokens to enhance efficiency. They also encode epochs so that the current keys can decrypt prior tokens, thereby eliminating the need for mass re-encryption. Nonetheless, these constructions do not address user revocation and provide limited security in adversarial settings where users may collude with the cloud server.

Collusion between users and the cloud server poses a significant threat to data privacy. In particular, the server can compromise the query privacy of honest readers by correlating their access patterns with those of colluding users. Chamani *et al.* [13] and Wang *et al.* [9] studied such attacks in the S/M architecture, proposing mitigation strategies that require the writer to replicate the index for each reader. Although these approaches improve resistance to collusion, they incur substantial storage overhead and fail to support user revocation.

Wang *et al.* [10] proposed delegatable searchable encryption for the M/M architecture, which enforces multi-user access control via delegated keywords. While the design supports delegation, it remains susceptible to inference attacks. In particular, all users can observe the number of updates associated with delegated keywords and the search frequency of all keywords. A corrupted user may leverage this global visibility to infer the private activities of honest users by tracking usage patterns. Furthermore, the cloud server can exploit update count leakage to deduce information about keyword updates, as the search token generated by a reader reveals the current update count. As with many existing constructions, the scheme does not address collusion between users and the cloud server, leaving it vulnerable to coordinated privacy breaches.

A TEE [20] provides a hardware-protected execution environment within modern processors, enabling secure processing of sensitive data even with an untrusted operating system. Recently, a growing body of work has explored TEEs for secure ciphertext retrieval while mitigating information leakage during search. Amjad *et al.* [21] first proposed SGX-supported dynamic SSE achieving forward and backward privacy by offloading token generation and document filtering to the TEE. Li *et al.* [22] proposed SEDCPT to mitigate count attacks while preserving forward and backward privacy. Subsequent works further extended TEE-assisted searchable encryption. Wu *et al.* [23] supported Boolean and range queries with forward and backward privacy. Jiang *et al.* [24] designed a scheme supporting conjunctive queries while hiding the result pattern. Yang *et al.* [25] proposed a volume-hiding range query scheme enabled by TEEs. Zheng *et al.* [26] introduced an efficient TEE-assisted protocol for secure similarity search.

However, these works primarily focus on single-writer single-reader settings and are not designed for multi-writer multi-reader environments. In such settings, additional challenges arise, including flexible user revocation, efficient search, and resistance to user–server collusion. Addressing these challenges remains an important problem for practical multi-user searchable encryption systems.

III. Preliminaries

A. Notations

Let λ denote the computational security parameter, and let $\mathsf{negl}(\cdot)$ denote a negligible function. We write $a \leftarrow A(x; y)$ to denote that a is the output of a probabilistic algorithm A on inputs x and y, where x is provided by one party and y by another. We use $[m]$ to denote the set of integers $\{0, 1, \ldots, m-1\}$, and $x||y$ to denote the concatenation of strings x and y.

B. Invertible Pseudorandom Function

A pseudorandom function (PRF) [27] is a keyed function $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ such that, for a uniformly random key $k \in \mathcal{K}$, the function $F(k, \cdot)$ is computationally indistinguishable from a uniformly random function f sampled from the set of all functions mapping $\mathcal{D}$ to $\mathcal{R}$. An invertible pseudorandom function (iPRF) [28] extends the standard notion of PRF by requiring the existence of an efficient inversion algorithm without sacrificing security.

Definition 1 (iPRF). *A keyed invertible pseudorandom function* $\mathsf{iF} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ *is specified by a triple of efficiently computable functions: a randomized key generation function* $\mathsf{Gen} : \{0,1\}^* \to \mathcal{K}$*, a deterministic evaluation function* $\mathsf{F} : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$*, and a deterministic inversion function* $\mathsf{F}^{-1} : \mathcal{K} \times \mathcal{R} \to 2^{\mathcal{D}}$*. The correctness property requires that for all* $y \in \mathcal{R}$*, it holds that*

$$\Pr_{k \xleftarrow{\$} \mathsf{Gen}(1^\lambda)} [\mathsf{F}^{-1}(k, y) \neq \{x \in \mathcal{D} | \mathsf{F}(k, x) = y\}] \leq \mathsf{negl}(\lambda).$$

For an adversary $\mathcal{A}$ *and iPRF* iF*, the adversarial advantage is defined as*

$$\mathbf{Adv}_{\mathcal{A},\mathsf{iF}}(1^\lambda) = \left| \Pr_{k \xleftarrow{\$} \mathsf{Gen}(1^\lambda)} [\mathcal{A}^{\mathsf{F}(k,\cdot),\mathsf{F}^{-1}(k,\cdot)}(1^\lambda) = 1] - \Pr_{\mathsf{R} \xleftarrow{\$} \mathsf{Func}[\mathcal{D},\mathcal{R}]} \mathcal{A}^{\mathsf{R}(\cdot),\mathsf{R}^{-1}(\cdot)}(1^\lambda) = 1 \right|.$$

The function iF *is said to be secure if for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that* $\mathbf{Adv}_{\mathcal{A},\mathsf{iF}}(1^\lambda) \leq \mathsf{negl}(\lambda)$.

C. Bilinear Map

Let $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ be cyclic groups of prime order p. A bilinear map is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ that satisfies the following properties:

- Bilinear: For all $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ and all $a, b \in \mathbb{Z}_p^*$, it holds that $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$.
- Non-degenerate: There exist elements $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ such that $e(g_1, g_2) \neq 1_{\mathbb{G}_t}$, where $1_{\mathbb{G}_t}$ is the identity element of $\mathbb{G}_t$.
- Computable: The map $e(g_1, g_2)$ can be computed efficiently for all $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$.

Computational Assumptions. Our proposed scheme relies on the hardness of the following standard computational problems.

- Decisional Bilinear Diffie-Hellman (DBDH): Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ be cyclic groups of prime order p, with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Given random elements $g_1^a, g_1^b, g_1^c \in \mathbb{G}_1$, plus a few additional terms, the quantity $e(g_1, g_2)^{abc} \in \mathbb{G}_t$ is indistinguishable from a random element in $\mathbb{G}_t$.

- Decisional Diffie-Hellman (DDH): Let $\mathbb{G}$ be a cyclic group of prime order p. Given random elements $g^a, g^b \in \mathbb{G}$, the element g^{ab} is indistinguishable from a random element in $\mathbb{G}$.

In this paper, we work in Type-3 [29] bilinear groups of prime order, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are asymmetric and there exists no efficiently computable homomorphism between them.

D. Proxy Re-Encryption

Proxy re-encryption (PRE) [30] enables transforming a ciphertext encrypted under one key into one decryptable under another, without revealing the underlying plaintext. In a PRE scheme, a proxy is provided with a re-encryption key that allows it to perform this transformation without learning anything about the original message.

A proxy re-encryption scheme consists of six algorithms $\mathsf{PRE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{ReKeyGen}, \mathsf{Enc}, \mathsf{ReEnc}, \mathsf{Dec})$ defined as follows:

- $(param) \leftarrow \mathsf{Setup}(1^\lambda)$: On input a security parameter 1^λ, output a cyclic group $\mathbb{G}$ of prime order p with generator g, i.e., $param = (\mathbb{G}, p, g)$.
- $(a, X) \leftarrow \mathsf{KeyGen}(param)$: Given the public parameters, choose a secret key $a \in \mathbb{Z}_p^*$ uniformly at random, and compute the corresponding public key $X = g^a$.
- $rk_{a\to b} \leftarrow \mathsf{ReKeyGen}(a, b, param)$: On input the secret keys a and b and the public parameters, output the re-encryption key $rk_{a\to b} = \frac{b}{a}$.
- $C \leftarrow \mathsf{Enc}(m, X, param)$: On input a message $m \in \mathbb{G}$, a public key X, and the public parameters, sample $r \leftarrow \mathbb{Z}_p^*$ and output the ciphertext $C = (C_1, C_2) = (X^r, m \cdot g^r)$.
- $C' \leftarrow \mathsf{ReEnc}(C, rk_{a\to b}, param)$: Given a ciphertext C encrypted under the secret key a, a re-encryption key $rk_{a\to b} = \frac{b}{a}$, and the public parameters, output the re-encrypted ciphertext $C' = (C_1^{\frac{b}{a}}, C_2)$.
- $m \leftarrow \mathsf{Dec}(C, a, param)$: On input a ciphertext $C = (C_1, C_2)$, a secret key a, and the public parameters, recover the message as $m = \frac{C_2}{C_1^{1/a}} = \frac{m \cdot g^r}{g^r}$.

Theorem 1 (**Semantic Security**). *The proxy re-encryption scheme [31] is semantically secure under the DDH assumption in a cyclic group* $\mathbb{G}$.

E. Trusted Execution Environment

A Trusted Execution Environment (TEE) [32] is a hardware-isolated area within a main processor that protects sensitive data with high confidentiality and integrity. Various TEE hardware solutions have been developed for different scenarios, such as ARM TrustZone [33] for mobile and embedded devices, and Intel Software Guard Extensions (SGX) [34] for server-side confidential computing. In this work, we leverage server-side Intel SGX for our cloud-based scheme.

Intel SGX introduces a specialized set of instructions and memory access mechanisms to the Intel architecture. These extensions allow an application to instantiate a protected container known as an enclave. An enclave provides a secure execution context that ensures the confidentiality and integrity of its internal code and data, even when the host system is compromised by privileged malware (e.g., an untrusted OS or Hypervisor). All enclaves reside in the Enclave Page Cache (EPC), a dedicated portion of physical memory encrypted by hardware. During enclave instantiation, SGX performs a measurement process to verify the enclave's integrity, ensuring the code has not been tampered with. The hardware-enforced memory protection and address mapping ensure that the enclave's memory remains inaccessible to any software outside the enclave boundary.

Despite these robust hardware protections, the untrusted OS remains responsible for managing enclave resources, which gives rise to potential side-channel attacks, such as cache timing attacks [35, 36] and page-fault attacks [37]. These attacks can leak enclave memory access patterns at cache-line or page-level granularity. To mitigate these risks, the execution logic within the enclave should ideally be data-oblivious. While advanced techniques like Oblivious RAM (ORAM) [38] can effectively hide access patterns, they often introduce significant computational and communication overhead. In our construction, since the sensitive metadata stored within the enclave is relatively small, we employ linear scanning as a straightforward yet effective strategy to ensure data-obliviousness. This approach effectively resists side-channel leakage while maintaining the system's overall efficiency. We do not consider side-channel attacks [39, 40] arising from platform-specific hardware vulnerabilities, which can be avoided by adopting more secure TEEs.

F. Keyword Private Information Retrieval

Private information retrieval (PIR) [41, 42] enables a client to retrieve a specific entry from a server-hosted indexed database while ensuring that the server learns no information about the identity of the retrieved entry. Keyword PIR (KPIR) [43] extends this notion by allowing the client to retrieve entries based on keywords, where the database is structured as a collection of keyword-value mappings, i.e., $D = \{(w_1, val_1), \ldots, (w_n, val_n)\}$. A KPIR scheme must satisfy the following correctness and privacy guarantees.

Correctness. For any database D and any keyword w contained in D, if the client and the server honestly follow the protocol, then the client obtains the corresponding value val such that $(w, val) \in D$, except with negligible probability.

Privacy. For any two keywords w_0 and w_1 in the database D, no PPT adversary can distinguish whether the client queried w_0 or w_1, except with negligible probability.

IV. Efficient and Updatable Keyword PIR

While KPIR enables private retrieval of entries by keywords, most existing constructions assume a static database. However, real-world databases are often dynamic, undergoing frequent insertions and deletions. In this section, we present Eureka[2], an efficient and updatable KPIR scheme that improves upon schemes [28, 41], providing strong privacy guarantees while supporting lightweight database updates with minimal cost.

[2]Eureka stands for efficient and updatable keyword private information retrieval, symbolizing the moment of discovery–"I have found it."

A. Definitions

Definition 2 (KPIR). *A client pre-processing KPIR scheme, which supports up to Q queries before refresh, consists of a tuple of efficient algorithms, defined as follows:*

- $(\mathsf{st}, D') \leftarrow \mathsf{Setup}(1^\lambda, D)$*, given a security parameter* $\lambda \in \mathbb{N}$ *and a keyword-value database* D*, the algorithm uses hashing-to-bins techniques to transform* D *into an indexable database. It outputs an indexed database* D' *for the server and a secret state* st *for the client.*
- $val \leftarrow \mathsf{Query}(w, \mathsf{st}; D')$*, given a query keyword* w*, the client's secret state* st*, and the database* D'*, the algorithm outputs the value* val *associated with keyword* w*.*
- $\mathsf{st}' \leftarrow \mathsf{Replenish}(\mathsf{st}, w, \xi)$*, given the secret state* st *of the client, the query keyword* w*, and the entry* ξ*, which holds the contents of the bin in* D' *to which keyword* w *is mapped, the algorithm outputs the updated* st'*.*
- $(\mathsf{st}'; D') \leftarrow \mathsf{Update}((w, val), \mathsf{st}; D')$*, given a keyword-value pair* (w, val) *to be inserted (or deleted*[3]*), the client secret state* st *and the database* D'*, the algorithm outputs the updated state* st' *and the modified database* D'*.*

Definition 3 (Correctness). *We say a KPIR scheme is correct if for all security parameters* $\lambda \in \mathbb{N}$*, all keyword-value databases* D *with at most* n *valid pairs, and all sequences of at most* Q *keyword queries* $\{w_1, \ldots, w_Q\}$ *such that each* w_i *is in the support of* D*, the following holds with probability at least* $1 - \mathsf{negl}(\lambda)$ *over the randomness of all algorithms:*

$$\begin{aligned}&\Pr[\mathsf{Query}(w, \mathsf{st}; D') = val : \\ &\quad (\mathsf{st}, D') \leftarrow \mathsf{Setup}(1^\lambda, D), \\ &\quad \mathsf{st}' \leftarrow \mathsf{Replenish}(\mathsf{st}, w, \xi), \\ &\quad (\mathsf{st}', D') \leftarrow \mathsf{Update}((w, val), \mathsf{st}; D')] \geq 1 - \mathsf{negl}(\lambda).\end{aligned}$$

Definition 4 (Query Privacy). *We define the IND-based security of a KPIR scheme via the following game* $\mathsf{IND}_{\mathcal{A}}^{KPIR,b}(\lambda)$ *shown in Figure 1, where the PPT adversary* $\mathcal{A}$ *has oracle access to* $\mathsf{Query}\mathcal{O}$ *and* $\mathsf{Update}\mathcal{O}$*.*

We say a KPIR is secure if for all PPT adversaries $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that:*

$$\left|\Pr\left[\mathsf{IND}_{\mathcal{A}}^{\mathrm{KPIR},0}(\lambda) = 1\right] - \Pr\left[\mathsf{IND}_{\mathcal{A}}^{\mathrm{KPIR},1}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

$\mathsf{IND}_{\mathcal{A}}^{\mathrm{KPIR},b}(1^\lambda)$
$D \leftarrow \mathcal{A}(1^\lambda)$
$(\mathsf{st}, D') \leftarrow \mathsf{Setup}(1^\lambda, D)$
$\mathcal{O} = (\mathsf{Query}\mathcal{O}, \mathsf{Update}\mathcal{O})$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}(D')$
return b'

$\mathsf{Query}\mathcal{O}^b(w_0, w_1)$
$val_b \leftarrow \mathsf{Query}(w_b, \mathsf{st}; D')$
$\mathsf{st}' \leftarrow \mathsf{Replenish}(\mathsf{st}, w_b, \xi_b)$
return val_b

$\mathsf{Update}\mathcal{O}^b(w_b, val_b)_{b\in\{0,1\}}$
$(\mathsf{st}'; D') \leftarrow \mathsf{Update}((w_b, val_b), \mathsf{st}; D')$
return st and D'

Fig. 1: Indistinguishability-based Game for KPIR

[3]To simplify the presentation, we refer to update operations as insertions. Deletions are treated in a similar manner by inserting an empty string as the val associated with the keyword.

B. Technical Overview

Eureka encodes the keyword-value database into an indexable database using hashing-to-bins techniques, enabling efficient KPIR by applying standard PIR protocols over the resulting indexed database. The use of an iPRF eliminates the need for the client to maintain extensive partition indices for each hint and avoids linear scans during updates. When updates are required, the client only needs to modify the affected bin and associated hints, without regenerating the entire database or all hints. This design ensures both scalability and practicality in dynamic settings.

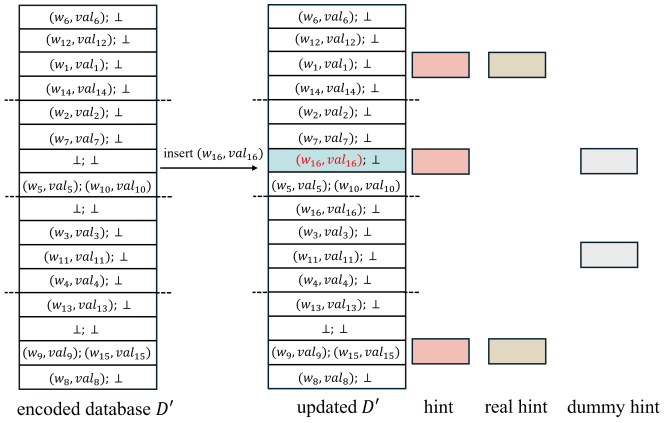


Fig. 2: Illustration of the Update and Search Operations

Concretely, the encoded database is divided into c partitions of size r. Each hint is generated by applying the iPRF to select $c/2 + 1$ partitions, and then choosing one bin from each of the selected partitions. Thus, each hint covers $c/2+1$ random bins, one from each selected partition. To search for a keyword in the i-th bin, the client locates a hint that covers this bin and sends the indices of the remaining $c/2$ bins in the hint to the server. The server returns the parity of these $c/2$ bins, from which the client recovers the content of the target bin. For insertions, the client updates the i-th bin and all hints referencing it.

Figure 2 illustrates the process. The encoded database D' is partitioned into 4 partitions of 4 bins (16 bins total, with 2 slots each, $\perp$ indicating empty slots). To insert (w_{16}, val_{16}), the client maps w_{16} to its bin, updates the bin, and refreshes all corresponding hints. To search for w_{16}, the client retrieves a hint covering its bin, constructs a real hint by excluding the bin containing w_{16}, and generates a dummy hint by selecting 2 random bins from the remaining partitions.

C. Our Concrete Eureka Scheme

a) Setup Phase: Eureka begins by encoding the input keyword-value database into an indexable format. Specifically, the keyword-value pairs are hashed into m bins using a collision-resistant hash function h. Each bin can hold up to s pairs, and the pairs of each bin are treated as a single atomic entry. The resulting database D' is divided into $c = m/r$ partitions, each consisting of r bins. For convenience, we assume c is an even integer. To construct hints, the client selects $c/2 + 1$ partitions uniformly at random from the c partitions.

Within each selected partition, one entry is randomly chosen to contribute to a parity computation.

Concretely, for each hint indexed by j, the client uses an iPRF to derive $c/2+1$ distinct partition indices: $v_{j,t} = \mathsf{iF.F}(k_1, j||t)$, where t is incremented from 0 until $c/2+1$ unique partitions are obtained. For each selected partition v, the client computes an offset $r_{j,v} = \mathsf{iF.F}(k_2, j||v)$, which determines the specific entry within that partition. The parity value is then computed as $p_j = \bigoplus_{v \in S_j} D'[vr + r_{j,v}]$, where S_j denotes the set of selected partitions. Each hint is stored as a tuple (j, p_j, t_j), where t_j denotes the maximum counter value used to obtain the required partitions. For correctness, the client must maintain M such hints (called main hints). Additionally, to support up to Q queries, it generates Q backup hints (organized in pairs) to replenish consumed ones, since each hint is single-use. Unlike main hints, each backup hint is composed of two parities, where each parity includes $c/2$ entries, each selected from a different partition. Algorithm 1 provides the pseudocode.

Algorithm 1: $(\mathsf{st}, D') \leftarrow \mathsf{Setup}(1^\lambda, D)$

Initialize m bins $B_1, \cdots, B_m$, each filled with 0;
For each $(w, val) \in D$, compute $i = h(w) \in [m]$ and append $w||val$ into B_i, resulting in an indexable database $D' = \{B_1, \cdots, B_m\}$;
Divide D' into $c = m/r$ partitions, each of size r;
$k_1, k_2 \leftarrow \mathsf{iF.Gen}(1^\lambda)$;
for $j \in \{0, 1, \cdots, M+Q-1\}$ **do**
  Initialize parity $p_j = 0$, and additionally initialize $p'_j = 0$ if $j \geq M$;
  Compute $v_{j,t} = \mathsf{iF.F}(k_1, j||t)$ to obtain $c/2$ unique partitions, where t is incremented from 0;
  if $j < M$ **then**
    Compute one more partition;
for $v \in \{0, 1, \cdots, c-1\}$ **do**
  Download $D'[vr : (v+1)r - 1]$ from the server;
  Compute preimage set $P_v = \mathsf{iF.F}^{-1}(k_1, v)$, remove preimages $j_i||t_i$, if $j_i \geq M+Q$ or $t_i > t_j$, and keep only hint ID j_i;
  for $j \in P_v$ **do**
    $x = D'[r_{j,v} + vr]$ where $r_{j,v} = \mathsf{iF.F}(k_2, j||v)$;
    $p_j = p_j \oplus x$;
  for $j \in \{M, M+1, \cdots, M+Q-1\} \setminus P_v$ **do**
    $x = D'[r_{j,v} + vr]$ where $r_{j,v} = \mathsf{iF.F}(k_2, j||v)$;
    $p'_j = p'_j \oplus x$;
return st =(hints, keys), D'.

b) Query Phase: To retrieve the value associated with a keyword w, the client first determines the index i of the bin to which w was assigned, and identifies the corresponding hint j that includes this bin. The client then computes all partitions involved in hint j, excluding the one containing bin i, to form a set S_j. From S_j, the client derives the set of bin indices $\mathcal{S}_j$ used in the parity computation, which is submitted as the query. Upon receiving $\mathcal{S}_j$, the server computes the parity $p = \bigoplus_{i \in \mathcal{S}_j} D'[i]$ and returns it to the client. The client then reconstructs the target bin entry as $p \oplus p_j$ and obtains the value associated with keyword w.

Algorithm 2: $val \leftarrow \mathsf{Query}(w, \mathsf{st}; D')$

Compute $(\alpha, \beta) = (\lfloor h(w)/r \rfloor, h(w) \bmod r)$;
Compute preimage set $P_\alpha = \mathsf{iF.F}^{-1}(k_1, \alpha)$, remove preimages $j_i||t_i$, if $j_i \geq M$ or $t_i > t_j$, and keep only hint ID j_i;
Compute preimage set $O_\beta = \mathsf{iF.F}^{-1}(k_2, \beta)$, remove preimages $j_i||v_i$, if $j_i \geq M$ or $v_i \neq \alpha$, and keep only hint ID j_i;
Initialize empty sets S_j, S'_j, $\mathcal{S}_j$, and $\mathcal{S}'_j$;
if $P_\alpha \cap O_\beta \neq \emptyset$ *and* $j = \min P_\alpha \cap O_\beta < M$ **then**
  $S_j = \{\mathsf{iF.F}(k_1, j||t)\}_{t=0}^{t_j} \setminus \{\alpha\}$;
  $S'_j = [c] \setminus S_j$;
  $\mathcal{S}_j = \{vr + r_{j,v} \mid r_{j,v} = \mathsf{iF.F}(k_2, j||v)\}_{v \in S_j}$;
  $\mathcal{S}'_j = \{vr + \mathsf{rand}()\}_{v \in S'_j}$;
else
  for $j = \{M, M+1, \cdots, M+Q-1\}$ **do**
    if $\eta = 0$ **then**
      $S_j = \{\mathsf{iF.F}(k_1, j||t)\}_{t=0}^{t_j}$;
      $S'_j = [c] \setminus S_j$;
    else if $\eta = 1$ **then**
      $S'_j = \{\mathsf{iF.F}(k_1, j||t)\}_{t=0}^{t_j}$;
      $S_j = [c] \setminus S'_j$;
    if $\alpha \in S_j$ *and* $\beta = \mathsf{iF.F}(k_2, j||\alpha)$ **then**
      $\mathcal{S}_j = \{e_j\} \cup \{vr + r_{j,v} \mid r_{j,v} = \mathsf{iF.F}(k_2, j||v)\}_{v \in S_j \setminus \{\alpha\}}$;
      $S_j = S_j \setminus \{\alpha\} \cup \{\lfloor e_j/r \rfloor\}$;
      $S'_j = S'_j \cup \{\alpha\} \setminus \{\lfloor e_j/r \rfloor\}$;
      $\mathcal{S}'_j = \{vr + \mathsf{rand}()\}_{v \in S'_j}$;
      break;
    else if $h(w) = e_j$ **then**
      $\mathcal{S}_j = \{vr + r_{j,v} \mid r_{j,v} = \mathsf{iF.F}(k_2, j||v)\}_{v \in S_j}$;
      $\mathcal{S}'_j = \{vr + \mathsf{rand}()\}_{v \in S'_j}$;
      break;
Send $(\mathcal{S}_j, \mathcal{S}'_j)$ or $(\mathcal{S}'_j, \mathcal{S}_j)$ to the server with equal probability;
Server computes the parities over $\mathcal{S}_j$ and $\mathcal{S}'_j$, and returns them to the client;
Compute $D'[h(w)] \leftarrow p \oplus p_j$, and extract the value val corresponding to keyword w;
return val.

The exclusion of the query index i from the set $\mathcal{S}_j$ implicitly reveals that i is not contained within $\mathcal{S}_j$. To prevent this leakage, the client generates a dummy set $\mathcal{S}'_j$, which consists of $c/2$ random indices, each selected from a distinct partition not involved in $\mathcal{S}_j$. Subsequently, the client randomly permutes the order of $\mathcal{S}_j$ and $\mathcal{S}'_j$ before sending both to the server. The server computes and returns the parities for both received sets, but cannot distinguish which one corresponds to the real set. To obtain the hint containing the query index with over-

whelming probability, we typically set $M = \lambda r$. Algorithm 2 provides the pseudocode for the query procedure.

Since each hint is single-use, a replenishment hint must be selected from the backup hints after each query. To maintain the randomness of the partitions selected by the main hints, the queried index i must be incorporated into the chosen backup hint. Specifically, each backup hint comprises two parities, each consisting of $c/2$ distinct partitions. The client selects one parity whose associated partition set does not include the partition containing index i. Let $\eta \in \{0, 1\}$ indicate how the parity in the replenishment hint is derived, where $\eta = 1$ indicates that the parity is constructed from the complementary partition set. The client then appends index i to the replenishment hint, and entry $D'[i]$ is added to the selected parity. The resulting replenishment hint is represented as (j, i, η, p_j, t_j). Algorithm 3 provides the pseudocode for the hint replenishment. Following the replenishment of main hints, two formats of hints, namely (j, p_j, t_j) and (J, i, η, p_J, t_j), may coexist. Accordingly, the query process must be modified slightly to correctly handle both formats. Lines 10-27 of Algorithm 2 present the modified steps of the query procedure.

Recall that the Query algorithm supports up to Q queries. After completing Q queries, the client must regenerate both the main and backup hints. This can be achieved using one of two strategies. The first, and simpler, approach is to rerun the Setup algorithm after every Q queries. The second approach involves initializing an additional set of main and backup hints in advance. During each query, the client downloads one partition and uses it to incrementally update the additional hint set. After Q queries, this maintained hint set is promoted to serve as the new active hint set.

Algorithm 3: $\mathsf{st}' \leftarrow \mathsf{Replenish}(\mathsf{st}, w, \xi)$

Compute $i = h(w)$ and $(\alpha, \beta) = (\lfloor i/r \rfloor, i \bmod r)$;
Let J be the ID of the next unused backup hint;
Compute $S_J = \{\mathsf{iF.F}(k_1, J||t)\}_{t=0}^{t_J}$;
if $\alpha \in S_J$ **then**
  $p_J = p'_J, \eta = 1$;
$e_J = i, p_J = p_J \oplus \xi$;
Replace hint j with backup hint (J, e_J, η, p_J, t_J).

c) Update Phase: An appealing feature of the Eureka construction is its support for dynamic updates to the database, which is enabled by the design of the hint mechanism and the use of iPRF. Specifically, to update (insert or delete) a keyword-value pair (w, val), the server first sends the entry from the bin to which keyword w is assigned to the client. The client then identifies all hints that include this entry in their parity computations and updates them accordingly. Simultaneously, the corresponding entry in the database D' is modified to reflect the update. Algorithm 4 provides the pseudocode for the update procedure.

Note. The database cannot accommodate an unbounded number of valid keyword-value pairs and supports arbitrary updates only within a fixed size n determined in advance. In practice, this limitation can be addressed by deleting outdated pairs or by carefully provisioning the database capacity. Notably, this design is well-suited for the multi-client settings considered in this work (e.g., IIoT), where readers typically care only about recent data, and older pairs (e.g., from a month ago) can be safely discarded.

Algorithm 4: $(\mathsf{st}'; D') \leftarrow \mathsf{Update}((w, val), \mathsf{st}; D')$

Compute $(\alpha, \beta) = (\lfloor h(w)/r \rfloor, h(w) \bmod r)$;
Server returns entry $\xi = D'[h(w)]$;
Update the entry ξ to ξ' according to (w, val);
Update the database $D'[h(w)] = \xi'$;
Compute preimage set $P_\alpha = \mathsf{iF.F}^{-1}(k_1, \alpha)$, remove preimages $j_i||t_i$, if $j_i \geq M + Q$ or $t_i > t_j$, and keep only hint ID j_i;
Compute preimage set $O_\beta = \mathsf{iF.F}^{-1}(k_2, \beta)$, remove preimages $j_i||v_i$, if $j_i \geq M + Q$ or $v_i \neq \alpha$, and keep only hint ID j_i;
$B = P_\alpha \cap O_\beta \setminus [M]$;
for $j \in P_\alpha \cap O_\beta$ **do**
  if $j < M$ *and hint* j *exists* **then**
    $p_j = p_j \oplus \xi \oplus \xi'$;
for $j \in \{M, M+1, \cdots, M+Q-1\}$ **do**
  if hint j lies in main hints **then**
    if $j \in B$ and $\eta = 0$ **then**
      $p_j = p_j \oplus \xi \oplus \xi'$;
    if $j \in B$ and $\eta = 1$ **then**
      if $h(w) = e_j$ **then**
        $p_j = p_j \oplus \xi \oplus \xi'$;
    if $j \notin B$ and $\eta = 0$ **then**
      if $h(w) = e_j$ **then**
        $p_j = p_j \oplus \xi \oplus \xi'$;
    if $j \notin B$ and $\eta = 1$ **then**
      $p_j = p_j \oplus \xi \oplus \xi'$;
  if hint j lies in backup hints **then**
    if $j \in B$ **then**
      $p_j = p_j \oplus \xi \oplus \xi'$;
    if $j \notin B$ **then**
      $p'_j = p'_j \oplus \xi \oplus \xi'$;

D. Theoretical Analysis

Complexity. We analyze the computational complexity of Eureka in both search and update procedures from the perspective of the client and the server. To perform a search, the client must first locate the hint that contains the queried index. This is achieved by evaluating an iPRF to identify a candidate subset of size $\mathcal{O}(M + Q + \log c)$[4] with constant overhead. The client then enumerates the indices covered by the corresponding partitions of the desired hint with an

[4]We map $\mathcal{O}((M + Q) \cdot c)$ elements into c partitions using iPRF. Thus, the expected number of elements per partition is $\mathcal{O}(M + Q)$. By standard balls-into-bins analysis, with high probability, the maximum partition load is $\mathcal{O}(M + Q + \log c)$.

additional $\mathcal{O}(c)$ overhead and sends them to the server. The server computes the corresponding parities in $\mathcal{O}(c)$ time and returns them. If the queried index is not found in the initial set of main hints, the client iteratively scans through the promoted hints from the backup hints (up to $\mathcal{O}(Q)$ in worst case), performing the same process for the desired hint. The server-side computation remains $\mathcal{O}(c)$ for the search.

For an update, the client must modify both the target entry in the database and all hints (including backup hints) that contain the entry. This requires using the iPRF to identify a subset of $\mathcal{O}(M + Q + \log c)$ candidate hints and updating each of them accordingly. The cost is thus proportional to the number of hints that may include the updated entry.

Correctness and privacy. We highlight that the primary distinction between Eureka and the construction in [41] lies in the mechanism for hint generation: whereas [41] employs a PRF, Eureka replaces it with an iPRF. Given that iPRFs are computationally indistinguishable from truly random functions, the distribution of generated hints in Eureka is computationally identical to that of [41]. For update operations, the client can exploit the invertibility of the iPRF to efficiently locate and modify all relevant hints. Consequently, Eureka inherits the correctness and privacy properties of [41] without introducing additional leakage or functional discrepancies.

Expansion. Eureka encodes the keyword-value database into an indexable database using hashing-to-bins techniques. A key parameter in this encoding is the bin expansion factor, which depends on the maximum load across bins. Assuming the hash function behaves like a uniform random oracle, this reduces to the classical balls-into-bins problem, where n balls are assigned uniformly at random into m bins. In the special case where $n = m$, it is well established that the maximum bin load is $\Theta\left(\frac{\log n}{\log \log n}\right)$ with high probability.

V. Hardware-Assisted Multi-User Searchable Encryption

A. System Architecture

The system architecture of Archon is illustrated in Figure 3, involving four participating entities.

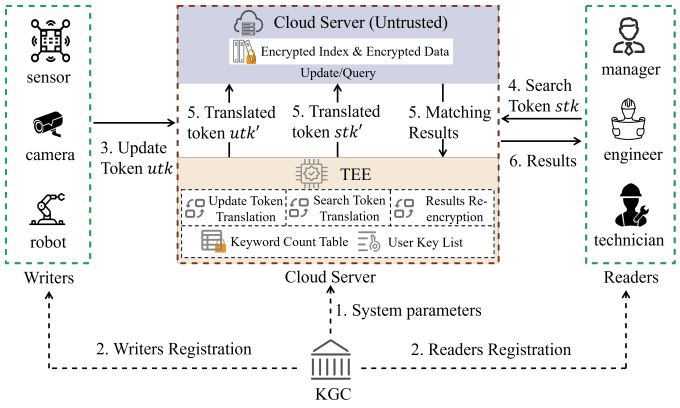


Fig. 3: M/M Architecture of Archon

- *A key generation center (KGC).* The KGC is a trusted authority responsible for system setup (Step 1) and user management (Step 2). It communicates with the TEE to provision system parameters and distribute cryptographic keys. The KGC handles both user enrollment and revocation for writers and readers. For each newly enrolled user, it assigns a unique identifier (typically numeric) and generates a corresponding secret key. The KGC informs the TEE of every user status update, whether enrollment or revocation, to ensure consistent access control.
- *A cloud server (CS).* The CS is modeled as a semi-honest entity equipped with a TEE. While the CS correctly follows the prescribed protocol, it may attempt to infer sensitive information from the stored data and observed interactions, thereby acting as a passive adversary. The CS is responsible for storing the encrypted index and encrypted data, and for processing update and search requests through interactions with the TEE (Step 5). The TEE serves as a trusted component that mediates interactions between users and the CS. It transforms user-submitted requests before forwarding them for execution. In particular, for update requests from writers and search queries from readers, the TEE performs the necessary cryptographic transformations so that the CS can execute the intended operations without learning sensitive information. The TEE also re-encrypts the matching results returned by the CS, ensuring that the ciphertexts can be correctly decrypted by the intended reader using their secret key. Throughout this paper, unless otherwise specified, the term "cloud server" refers exclusively to the portion of the infrastructure excluding the TEE module.
- *Writers.* Each writer (e.g., sensors, cameras, etc.) outsources its data to the CS for storage and shares it with all authorized readers (Step 3). Prior to outsourcing, each writer must register with the KGC and obtain secret keys to become an authorized writer. Furthermore, writers may be corrupted by the adversary, meaning that the adversary could obtain the writer's secret key, update history, or even take control of the writer.
- *Readers.* Each reader must register with the KGC and obtain the necessary secret keys to access the shared data (Step 4) and decrypt the returned matching results (Step 6). However, both authorized and revoked readers may be corrupted by the adversary, enabling them to collude with the CS or other malicious entities, thereby posing serious threats to data privacy.

The TEE acts as the mediator between users and the CS-hosted encrypted index. During setup and enrollment, the KGC provisions the TEE-side secret state and synchronizes user status information with the TEE, ensuring that the enclave maintains an up-to-date list of authorized users. During an update, the CS forwards a writer's update token to the TEE, and then the TEE verifies the writer's authorization, transforms the token into an effective update token for the shared encrypted index, updates the protected keyword state, and invokes Eureka to update the CS-hosted database. During a search, the CS forwards a reader's search token to the TEE, and then the TEE verifies the reader's authorization, derives the effective retrieval token, retrieves the required encrypted

entries through Eureka, and re-encrypts the results for the intended reader. During revocation, the KGC instructs the TEE to remove the corresponding user entry, after which requests from the revoked user are rejected by the TEE.

B. Multi-user Searchable Encryption

Definition 5 (Archon)**.** *A multi-user searchable encryption scheme assisted by a trusted execution environment, denoted as* $\mathsf{Archon} = (\mathsf{Setup}, \mathsf{Enroll}, \mathsf{Update}, \mathsf{Search}, \mathsf{Revoke})$*, consists of the following algorithms:*

- $(\mathsf{st}_{pub}; \mathsf{st}_{KGC}; \mathsf{st}_{TEE}; \mathsf{st}_{CS}) \leftarrow \mathsf{Setup}(1^\lambda, D)$*: given a security parameter* $\lambda \in \mathbb{N}$ *and a keyword-value database* D *(which may be empty), this algorithm outputs the public parameter* st_{pub} *along with the state information of participating entities, including the KGC's state* st_{KGC}*, the TEE's state* st_{TEE}*, and the CS's state* st_{CS}*. The public parameter* st_{pub} *is implicitly provided as input to all subsequent algorithms.*
- $(\mathsf{sk}_u; \mathsf{st}'_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Enroll}(u; \mathsf{st}_{KGC})$*: given a new user (writer or reader) identity* u *and the KGC's state* st_{KGC}*, this algorithm outputs the user's secret key* sk_u*, and updates the states of both the KGC and the TEE.*
- $(\mathsf{st}'_{TEE}; \mathsf{st}'_{CS}) \leftarrow \mathsf{Update}(u, w, val, \mathsf{sk}_u; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$*: given a writer's identity* u *and the secret key* sk_u*, a keyword-value pair* (w, val)[5]*, the CS's state* st_{CS}*, and the TEE's state* st_{TEE}*, this algorithm updates both the TEE's state and the CS's state accordingly.*
- $(\mathsf{st}'_{CS}; \mathcal{R}) \leftarrow \mathsf{Search}(u, w, \mathsf{sk}_u; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$*: given a reader's identity* u *and the secret key* sk_u*, a keyword* w*, the CS's state* st_{CS}*, and the TEE's state* st_{TEE}*, this algorithm returns the search result* $\mathcal{R}$ *and updates the state of the CS accordingly.*
- $(\mathsf{st}_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Revoke}(u)$*: given an existing user (writer or reader) identity* u*, this algorithm updates the states of both the KGC and the TEE.*

Definition 6 (**Correctness**)**.** *We say the* Archon *scheme is correct if for any security parameter* $\lambda \in \mathbb{N}$*, any keyword-value database* D *(possibly empty) of size at most* n*, any valid user enrolled via* Enroll *algorithm and assuming no users have been revoked, for any sequence of search or update requests issued by enrolled users, the following holds with probability at least* $1 - \mathsf{negl}(\lambda)$ *over the randomness of all algorithms:*

$$\begin{aligned}&\Pr[\mathsf{Search}(u, w, \mathsf{sk}_u; \mathsf{st}_{CS}; \mathsf{st}_{TEE}) = D(w) : \\ &\quad (\mathsf{st}_{pub}; \mathsf{st}_{KGC}; \mathsf{st}_{TEE}; \mathsf{st}_{CS}) \leftarrow \mathsf{Setup}(1^\lambda, D), \\ &\quad (\mathsf{sk}_u; \mathsf{st}'_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Enroll}(u; \mathsf{st}_{KGC}), \\ &\quad (\mathsf{st}'_{TEE}; \mathsf{st}'_{CS}) \leftarrow \mathsf{Update}(u, w, val, \mathsf{sk}_u; \mathsf{st}_{CS}; \mathsf{st}_{TEE})] \\ &\quad \geq 1 - \mathsf{negl}(\lambda),\end{aligned}$$

where $D(w)$ *indicates all the identifiers of documents containing the keyword* w *in the database* D*.*

Definition 7 (**Access Pattern** [44])**.** *Access pattern reveals which documents are returned in response to a search query for a keyword* w*. Formally, it is defined as*

$$\mathsf{ap}(w) = D(w).$$

[5]The method of deletion is the same as that of addition [19]

Definition 8 (**Search Pattern** [44])**.** *Let* $Q = \{(j, w_j)\}_{j \in [q]}$ *denote the query history, where* j *is the timestamp of the query and* w_j *is the queried keyword at timestamp* j*. The search pattern reveals which queries are performed on the same keyword. Formally, for a keyword* w*, it is defined as*

$$\mathsf{sp}(w) = \{j \in [q] : w_j = w\}.$$

Hence, $\mathsf{sp}(w)$ *captures all timestamps at which* w *is queried.*

C. Threat Model

We consider an adversary who can corrupt any subset of writers, readers, and the CS. By default, the CS is assumed to be corrupted and is modeled as a semi-honest adversary, meaning that it follows the prescribed protocol correctly but may attempt to infer sensitive information from the data it observes during protocol execution. Corrupted users may attempt to correlate their own inputs and observations with those of honest users in order to infer private information, such as queried keywords or associated documents.

We further allow collusion between the CS and corrupted users (writers or readers). In this setting, the adversary can jointly analyze the access patterns of honest users observed by the CS and the internal state of corrupted users. The internal state of a corrupted user is limited to the user's own secret keys and its own update or search history; it does not include TEE-internal global states or intermediate values used to transform honest users' tokens. Although multiple users operate over a shared global encrypted index, they do not share secret states or intermediate cryptographic values with each other.

The TEE is modeled as a trusted entity that faithfully executes the prescribed protocol. It is assumed to provide confidentiality and integrity protection for all computations and data inside the enclave. In particular, we assume that cryptographic keys stored inside the enclave cannot be extracted by the adversary. This assumption is consistent with prior work [45], which shows that when cryptographic operations are implemented using well-tested constant-time libraries, secret keys can be effectively protected against known side-channel attacks.

Nevertheless, TEEs may still be vulnerable to certain side-channel attacks. In particular, attackers controlling the system software may attempt to infer sensitive information by observing enclave memory access patterns through microarchitectural channels, such as cache timing attacks or page-fault attacks. To mitigate such leakage, our design adopts data-oblivious operations when processing enclave data structures, ensuring that memory access patterns do not depend on sensitive inputs. We do not consider side-channel attacks arising from platform-specific hardware vulnerabilities, which can be avoided by adopting more secure TEEs.

The KGC is modeled as a fully trusted entity, potentially operated by an authoritative organization or the system owner. It is responsible for managing system-wide cryptographic secrets and initial user credentials. We assume the KGC is honest and does not collude with any adversary or malicious participant. This is a standard and reasonable assumption in many practical settings, where the KGC is managed by a

centralized, trustworthy administrator. The KGC is assumed to distribute cryptographic keys to authorized users via secure channels. Furthermore, while critical for system reliability, ensuring data integrity is outside the scope of this work. We assume that such concerns can be addressed through the deployment of authenticated encryption (e.g., AES-GCM) and standard transport protocols (e.g., TLS), allowing us to focus on the fundamental challenges of privacy-preserving retrieval and collusion resistance in the M/M setting.

D. Security

The adaptive security of Archon is defined with respect to a leakage function $\mathcal{L}$, which characterizes the information leaked by the system during the execution of each algorithm. The adversary $\mathcal{A}$ is granted adaptive access to oracles of enrollment $\mathsf{Enroll}\mathcal{O}$, revocation $\mathsf{Revoke}\mathcal{O}$, update $\mathsf{Update}\mathcal{O}$, search $\mathsf{Search}\mathcal{O}$, and corruption $\mathsf{Corr}\mathcal{O}$, where each query may depend on the history of prior interactions. In particular, the corruption oracle allows $\mathcal{A}$ to adaptively corrupt arbitrary users (writers or readers) and obtain their internal state.

Figures 4 and 5 illustrate the real and ideal security experiments under the presence of corrupted users. In the real experiment, oracle queries are answered by the actual Archon algorithms. In contrast, in the ideal experiment, a simulator $\mathcal{S}$ responds to the adversary's queries using only the information provided by the leakage function $\mathcal{L}$. The adversary $\mathcal{A}$ may corrupt users to gain access to their secret keys and the corresponding search or update histories. We say that Archon is $\mathcal{L}$-adaptively secure if no PPT adversary can distinguish whether it is interacting with the real or ideal experiment, except with negligible probability. The details of $\mathsf{UpdtBy}(u_i)$ and $\mathsf{SrchBy}(u_i)$ are defined in subsection F2.

$\mathsf{Real}_{\mathcal{A}}^{\mathsf{Archon}}(1^\lambda)$

$(\mathsf{st}_{\mathcal{A}}, D) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{st}_{pub}; \mathsf{st}_{KGC}; \mathsf{st}_{TEE}; \mathsf{st}_{CS}) \leftarrow \mathsf{Setup}(1^\lambda, D)$

$\mathcal{O} = \left\{ \begin{array}{c} \mathsf{Enroll}\mathcal{O}, \mathsf{Revoke}\mathcal{O}, \\ \mathsf{Update}\mathcal{O}, \mathsf{Search}\mathcal{O}, \\ \mathsf{Corr}\mathcal{O} \end{array} \right\}$

$b \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{st}_{\mathcal{A}}, \mathsf{st}_{pub}, \mathsf{st}_{CS})$

return b

$\mathsf{Enroll}\mathcal{O}(u_i)$

$(\mathsf{sk}_{u_i}; \mathsf{st}'_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Enroll}(u_i; \mathsf{st}_{KGC})$

return u_i

$\mathsf{Revoke}\mathcal{O}(u_i)$

$(\mathsf{st}_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Revoke}(u_i)$

return u_i

$\mathsf{Update}\mathcal{O}(u_i, w, val)$

$(\mathsf{st}'_{TEE}; \mathsf{st}'_{CS}) \leftarrow \mathsf{Update}(u_i, w, val, \mathsf{sk}_{u_i}; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$

$//utk_i$ denotes the update token

return $(\mathsf{st}'_{CS}, utk_i)$

$\mathsf{Search}\mathcal{O}(u_i, w)$

$(\mathsf{st}'_{CS}; \mathcal{R}) \leftarrow \mathsf{Search}(u_i, w, \mathsf{sk}_{u_i}; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$

$//stk_i$ denotes the search token

return $(\mathsf{st}'_{CS}, \mathcal{R}, stk_i)$

$\mathsf{Corr}\mathcal{O}(u_i)$

if u_i is a writer **then**

  return $(\mathsf{sk}_{u_i}, \mathsf{UpdtBy}(u_i))$

if u_i is a reader **then**

  return $(\mathsf{sk}_{u_i}, \mathsf{SrchBy}(u_i))$

Fig. 4: Real experiment for Archon with corrupted users

Definition 9 (Adaptive security). *Let a multi-user searchable encryption scheme be defined as* Archon = (Setup,Enroll, Update,Search,Revoke)*, where* $\lambda \in \mathbb{N}$ *is the security parameter. Let* $\mathcal{A}$ *be a PPT adversary, and let* $\mathcal{S}$ *be a PPT simulator. Consider the probabilistic experiments defined in Figures 4 and 5. We say that* Archon *is adaptively secure in the presence of corrupted participants with respect to the leakage* $\mathcal{L}$*, if for all adversaries* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that:*

$$\Pr[\mathsf{Real}_{\mathcal{A}}^{\mathsf{Archon}}(1^\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathsf{Archon}}(1^\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the probabilities are taken over the randomness of all parties in the respective experiments.

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathsf{Archon}}(1^\lambda)$

$(\mathsf{st}_{\mathcal{A}}, D) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{st}_{pub}; \mathsf{st}_{CS}) \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}^0))$

$\mathcal{O} = \left\{ \begin{array}{c} \mathsf{Enroll}\mathcal{O}, \mathsf{Revoke}\mathcal{O}, \\ \mathsf{Update}\mathcal{O}, \mathsf{Search}\mathcal{O}, \\ \mathsf{Corr}\mathcal{O} \end{array} \right\}$

$b \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{st}_{\mathcal{A}}, \mathsf{st}_{pub}, \mathsf{st}_{CS})$

return b

$\mathsf{Enroll}\mathcal{O}(u_i)$

$\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Enroll}, u_i)\}$

$(\mathsf{sk}_{u_i}; \mathsf{st}'_{KGC}; \mathsf{st}_{TEE}) \leftarrow \mathsf{Enroll}(u_i; \mathsf{st}_{KGC})$

return $u_i \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

$\mathsf{Revoke}\mathcal{O}(u_i)$

$\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Revk}, u_i)\}$

return $u_i \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

$\mathsf{Update}\mathcal{O}(u_i, w, val)$

$\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Updt}, u_i, w, val)\}$

$(\mathsf{st}_{CS}, utk_i) \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

return $(\mathsf{st}_{CS}, utk_i)$

$\mathsf{Search}\mathcal{O}(u_i, w)$

$\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Srch}, u_i, w)\}$

$(\mathsf{st}_{CS}, \mathcal{R}, stk_i) \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

return $(\mathsf{st}_{CS}, \mathcal{R}, stk_i)$

$\mathsf{Corr}\mathcal{O}(u_i)$

if u_i is a writer **then**

  $\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Corr}, u_i)\}$

  $(\mathsf{sk}_{u_i}, \mathsf{UpdtBy}(u_i)) \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

  return $(\mathsf{sk}_{u_i}, \mathsf{UpdtBy}(u_i))$

if u_i is a reader **then**

  $\mathcal{H} = \mathcal{H} \cup \{(\mathsf{Corr}, u_i)\}$

  $(\mathsf{sk}_{u_i}, \mathsf{SrchBy}(u_i)) \leftarrow \mathcal{S}(\mathcal{L}(\mathcal{H}))$

  return $(\mathsf{sk}_{u_i}, \mathsf{SrchBy}(u_i))$

Fig. 5: Ideal experiment for Archon with corrupted users

Leakage function. Let $\mathcal{L}$ denote the leakage function that captures the information leaked by the system during the execution of each algorithm. We define the history of Archon as a query sequence $\mathcal{H}^t = (q_0, q_1, \cdots, q_t)$, where each query q_j is associated with a timestamp j indicating when the query occurred. Each q_j takes one of the following forms:

$$q_j \in \left\{ \begin{array}{c} (\mathsf{Enroll}, u_i), (\mathsf{Revk}, u_i), (\mathsf{Corr}, u_i), \\ (\mathsf{Updt}, u_i, w, val), (\mathsf{Srch}, u_i, w) \end{array} \right\},$$

with the initial query $q_0 = (\mathsf{Setup}, D)$ representing the system setup with database D. This query leaks the size of the encoded database as part of the setup leakage. The leakage associated with the history $\mathcal{H}^t$ is denoted as $\mathcal{L}(\mathcal{H}^t) = \overline{q}_0 || \overline{q}_1 || \cdots || \overline{q}_t$, where each $\overline{q}_j$ represents the information leaked by the j-th query. The formal specification of $\overline{q}_j$ is described in subsection F2.

Forward privacy. Forward privacy ensures that update operations do not leak information about the updated keywords and that past search tokens cannot be used to match newly inserted entries, even if they pertain to the same keyword [46, 47]. While traditional definitions of forward privacy assume an S/S architecture, our setting considers a M/M architecture in which users may be corrupted. Therefore, the leakage introduced by corrupted users must also be explicitly considered.

Informally, the forward privacy of Archon requires that the leakage incurred during an update is limited to the timestamp j and the identity of the writer u_i. The actual update content (w, val) is leaked only if u_i is corrupted. This guarantees that no additional information about the updated keyword is leaked beyond what can be inferred from these values.

Definition 10 (Forward privacy). *An $\mathcal{L}$-adaptively secure* Archon *is said to satisfy forward privacy if, for any update q_j issued by a writer u_i, the corresponding leakage $\overline{q}_j$ is defined as:*

$$\overline{q}_j = \begin{cases} (\mathsf{Updt}, u_i) & \text{if } u_i \text{ is honest}, \\ (\mathsf{Updt}, u_i, w, val) & \text{if } u_i \text{ is corrupted}. \end{cases}$$

VI. Our Proposed Archon

Following the discussion in Section I-A, this section presents Archon, a hardware-assisted multi-user searchable encryption scheme, built upon an efficient and updatable KPIR construction, Eureka. We present our construction in Figure 6.

A. Technical Overview

We target multi-writer/multi-reader searchable encryption that supports flexible enrollment and revocation, efficient search, forward privacy, and corruption-resistant privacy. To realize these properties, Archon builds on two core technical components: a TEE-assisted mediator for user-operation transformation and sensitive state management, and an updatable keyword PIR protocol, called Eureka.

To reconcile flexible enrollment and revocation with efficient search, the TEE acts as a token-transformation mediator. It converts randomized tokens generated under different user keys into consistent effective tokens for the shared global index. Each user holds only its own key share and can generate randomized tokens, while the corresponding TEE-side key share is required to transform them into effective tokens applicable to the encrypted index. This enables enrollment and revocation through the TEE-maintained user list, without reconstructing the index or redistributing keys to all remaining users, while allowing all authorized users to operate over the same global encrypted index.

Archon also provides forward privacy without requiring users to synchronize secret states, such as keyword update counters. The TEE maintains these states internally and uses them to derive fresh effective update tokens. Since these states are never exposed to users or the CS, corrupted users cannot exploit global keyword states to infer honest users' update activities.

For corruption-resistant privacy, Archon considers collusion between corrupted users and the CS. Corrupted users may compare their own search or update histories with the transcripts and access observations collected by the CS to infer honest users' queried or updated keywords. Archon mitigates this threat at two levels. At the user-to-TEE token level, search and update tokens include fresh randomness, making tokens for the same keyword computationally unlinkable before TEE-side transformation. At the TEE-to-CS retrieval level, Archon invokes Eureka, whose PIR queries hide the queried keyword and accessed locations from the CS. Thus, Eureka protects both the PIR-level search pattern and the access pattern of encrypted index retrieval. Together, these mechanisms prevent corrupted users and the CS from correlating token transcripts or access observations to infer honest users' queried or updated keywords, except for the leakage explicitly captured by the security model. The overall construction is shown in Figure 6.

B. Setup

In the setup phase, each system entity initializes its local state and shares essential information. The KGC generates three cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ of prime order p, along with generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, and a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. It defines three cryptographic hash functions: $h_1 : \{0,1\}^* \to \mathbb{G}_1$ that maps arbitrary strings to elements in $\mathbb{G}_1$, $h_2 : \{0,1\}^* \to \mathbb{Z}_p^*$ that maps arbitrary strings to elements in $\mathbb{Z}_p^*$, and $H : \{0,1\}^* \to \{0,1\}^*$ for general-purpose hashing, and instantiates a PRE construction over $\mathbb{G}_1$ with generator g_1. The KGC randomly selects a secret key $x \in \mathbb{Z}_p^*$ and a secret key $k \in \mathcal{K}$ for a pseudorandom function F, and securely shares k with the TEE. These secret keys x and k are used to derive user-specific cryptographic keys on demand. Finally, the KGC initializes an empty user list UL_{KGC} for managing system users.

To avoid storing many user-specific keys, the TEE derives individual user keys on-the-fly by applying F to a user identity under key k. The same key k is also used to derive per-user keys for generating randomness, ensuring that even for identical keywords yield distinct tokens across requests. The TEE instantiates the Eureka scheme to perform update and search operations over the database stored at the CS. It also initializes an empty keyword count table (KCT) to track updates per keyword. The KCT is stored as a table encrypted using AES, and the TEE re-encrypts the entire KCT upon each update. Meanwhile, the CS is responsible for storing users' encrypted data. As is common in many searchable encryption schemes [10, 14, 15], Archon returns the identifiers of matching documents in response to search queries.

C. Enrollment and Revocation

The KGC maintains a user list to handle user enrollment and revocation within the system. For each enrolled user u, the KGC derives secret keys $\frac{x}{x_u}$ and k_r, where $x_u = h_2(F(k,u))$ and $k_r = H(F(k, u||\text{“}random\text{”}))$, respectively, using its secret keys x and k. The TEE also maintains a synchronized copy of the user list and, given the shared key k, can compute x_u and k_r independently to authenticate and process user requests. User revocation is handled in a straightforward manner, the revoked user is simply removed from the list.

While the TEE maintains a local copy of the user list in our construction, this design choice is not strictly necessary. To reduce TEE-side storage, the user list could alternatively be maintained solely by the KGC. In such a case, each user request would need to be accompanied by a digital signature from the KGC attesting to the user's authorization. However, since the user list only contains lightweight identity

Archon Construction

Setup$(1^\lambda, D)$:

1. The KGC initializes cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ of prime order p, along with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ and generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$. It defines three cryptographic hash functions: $h_1 : \{0,1\}^* \to \mathbb{G}_1$, $h_2 : \{0,1\}^* \to \mathbb{Z}_p^*$ and $H : \{0,1\}^* \to \{0,1\}^*$, as well as a PRF $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$. Then, it invokes $\mathsf{PRE.Setup}(1^\lambda)$ using group $\mathbb{G}_1$ and sets $param = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, p, e, g_1, g_2, h_1, h_2, H, F)$.
2. The KGC randomly samples a secret key $x \in \mathbb{Z}_p^*$ and a key $k \in \mathcal{K}$ for PRF F, and shares the key k with the TEE. It initializes an empty user list $UL_{KGC} \leftarrow \emptyset$ to manage authorized users.
3. The KGC instructs the TEE to initialize an empty user list $UL_{TEE} \leftarrow \emptyset$ to manage authorized users. It then instantiates the KPIR scheme Eureka and invokes $\mathsf{Eureka.Setup}(1^\lambda, D)$ to obtain (st_{KPIR}, D'). Additionally, the TEE initializes an empty keyword count table $KCT \leftarrow \emptyset$ to track keyword update counts.
4. Output the public state as $\mathsf{st}_{pub} = (param)$, the KGC state as $\mathsf{st}_{KGC} = (x, k, UL_{KGC})$, the TEE state as $\mathsf{st}_{TEE} = (k, UL_{TEE}, \mathsf{st}_{KPIR}, KCT)$, and the CS state as $\mathsf{st}_{CS} = D'$.

Enroll$(u, \mathsf{st}_{pub}; \mathsf{st}_{KGC})$:

1. The KGC appends the user u to the user list, i.e., $UL_{KGC} = UL_{KGC} \cup \{u\}$, and assigns the user secret keys $\frac{x}{x_u} \in \mathbb{Z}_p^*$ and k_r, where $x_u = h_2(F(k, u))$ and $k_r = H(F(k, u||\text{“}random\text{”}))$, respectively.
2. The KGC securely sends the generated secret keys $\frac{x}{x_u}$ and k_r to the user u and notifies the TEE of the new enrollment, upon which the TEE adds u to its user list UL_{TEE}.

Update$(u, w, val, \mathsf{sk}_u; \mathsf{st}_{pub}; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$:

1. The writer randomly selects a string $iv \in \{0,1\}^*$, and computes $r_w = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$. It then computes the update token as $e_w = e(h_1(w), g_2^{\frac{x}{x_u} \cdot r_w})$ and encrypts the val as $e_{val} = (C_1, C_2) = \mathsf{PRE.Enc}(val, g_1^{\frac{x}{x_u}}, param)$. The writer sends the update token $utk = (u, iv, e_w, e_{val})$ to the TEE via the CS.
2. The TEE computes the random value $r_w = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$ and transforms e_w into a new value $e'_w = e_w^{\frac{x_u}{r_w}}$, where $k_r = H(F(k, u||\text{“}random\text{”}))$ and $x_u = h_2(F(k, u))$, respectively. Then, the TEE retrieves the current keyword update count $j = KCT[H(e'_w)]$ and computes the key as $\mathsf{key} = H(e_w'^{j+1})$, the value as $\mathsf{value} = (C_1, C_2) = \mathsf{PRE.ReEnc}(e_{val}, x_u, param)$, and increments $KCT[H(e'_w)]$.
3. The TEE invokes $\mathsf{Eureka.Update}(\mathsf{key}, \mathsf{value}, \mathsf{st}_{KPIR}; D')$.

Search$(u, w, \mathsf{sk}_u; \mathsf{st}_{pub}; \mathsf{st}_{CS}; \mathsf{st}_{TEE})$:

1. The reader randomly selects a string $iv \in \{0,1\}^*$ and a value $y \in \mathbb{Z}_p^*$, computes $r_r = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$, $Y = g_1^y$, $e_r = e(h_1(w), g_2^{\frac{x}{x_u} \cdot r_r})$, and sends the search token $stk = (u, iv, e_r, Y)$ to the TEE via the CS.
2. The TEE computes the random value $r_r = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$ and transforms e_r into a new value $e'_r = e_r^{\frac{x_u}{r_r}}$, where $k_r = H(F(k, u||\text{“}random\text{”}))$ and $x_u = h_2(F(k, u))$, respectively. Then, the TEE retrieves the current keyword update count $j = KCT[H(e'_r)]$.
3. For each $\ell \in [j]$, the TEE computes $\mathsf{key} = H(e_w'^{\ell})$ and invokes $\mathsf{Eureka.Query}(\mathsf{key}, \mathsf{st}_{KPIR}; D')$. It then re-encrypts the retrieved value as $e_{val} = (C_1, C_2) = \mathsf{PRE.ReEnc}(\mathsf{value}, \frac{1}{x_u}, param)$. After re-encryption, the TEE randomly selects $y' \in \mathbb{Z}_p^*$, computes $C_0 = g_1^{y'}$ and updates C_2 as $C_2 = C_2 \oplus H(Y^{y'})$. It then forms the final ciphertext as (C_0, C_1, C_2) and returns the search results to the reader.

Revoke(u):

1. The KGC removes the user u from UL_{KGC} and instructs the TEE to remove u from its user list UL_{TEE}.

Fig. 6: A Multi-User Searchable Encryption Construction

information and incurs minimal storage overhead, we choose to store it within the TEE to streamline protocol design and reduce reliance on real-time KGC involvement.

D. Update

Each authorized writer possessing valid keys is permitted to update the encrypted database. To prevent leakage of keyword update patterns, particularly the frequency of updates to the same keyword, we randomize the generation of update tokens. Specifically, each update token incorporates a fresh randomness r_w, and is computed as $e_w = e(h_1(w), g_2^{\frac{x}{x_u} \cdot r_w})$, where the randomness is deterministically derived from a random string $iv \in \{0,1\}^*$ and the secret key k_r, $r_w = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$. For the corresponding val, the writer encrypts it using PRE.

Upon receiving an update request, the TEE reproduces the random value r_w using the writer's secret key k_r and the random string iv, and transforms the token e_w accordingly. The TEE maintains a KCT that tracks the number of updates associated with each keyword. This count is embedded in the transformed token, thereby ensuring that even repeated updates to the same keyword yield unlinkable tokens. In addition, the TEE re-encrypts the encrypted val and invokes the underlying

keyword PIR scheme Eureka to update the encrypted database accordingly.

E. Search

Each authorized reader with valid keys is allowed to search the encrypted database, which contains data contributed by multiple writers. Similar to the update procedure, to preserve search pattern privacy, the generation of the search token incorporates fresh randomness. Specifically, the search token is computed as $e_r = e(h_1(w), g_2^{\frac{x}{x_u} \cdot r_r})$, where r_r is a random value deterministically derived from a random string $iv \in \{0,1\}^*$ and the secret key k_r, $r_r = h_2(F(k_r, u||iv)) \in \mathbb{Z}_p^*$.

When the CS receives a search request, it forwards the search token to the TEE. The TEE then reconstructs the corresponding random value r_r and transforms the token e_r accordingly. It then retrieves the KCT to determine the number of updates associated with the queried keyword. Using this information, the TEE invokes the Eureka to retrieve all matching encrypted results from the database.

F. Theoretical Analysis

1) Complexity: We analyze the computational complexity of the Archon with respect to its main procedures. For both update and search, the user incurs only constant overhead, generating an update or search token in $\mathcal{O}(1)$ time. Then the TEE transforms the token and invokes the Eureka procedure to update the database, which incurs a cost of $O(M+Q+\log c)$, where M is the number of main hints, Q the number of backup hints, and c the number of partitions.

For search operations, the TEE similarly retrieves the KCT to determine the number of updates associated with the searched keyword, denoted by $|D(w)|$. It then performs Eureka queries for each search, incurring a total cost of $\mathcal{O}(|D(w)|\cdot(M+Q+\log c+c))$, where the additional $\mathcal{O}(c)$ term accounts for the server-side parity reconstruction required by each Eureka query. Finally, the TEE re-encrypts all matched results and returns them to the reader.

2) Security: In the following, we formalize the security guarantees of Archon under a threat model where users may be corrupted and may collude with the cloud server.

Theorem 2. *Assume that all hash functions are modeled as random oracles, F is a pseudorandom function, the* PRE *and the AES are semantically secure,* Eureka *satisfies query privacy, and the DDH and DBDH assumptions hold in their respective groups. Then, the* Archon *scheme is $\mathcal{L}$-adaptively secure and satisfies forward privacy in the presence of user corruption.*

For our analysis, we denote the set of corrupted users at time t by $\mathcal{C} = \{u_i | (\mathsf{Corr}, u_i) \in \mathcal{H}^t\}$. We define two functions to describe the information that corrupted users will leak: $\mathsf{UpdtBy}^t(u_i)$, which captures the update history associated with the corrupted writer u_i, and $\mathsf{SrchBy}^t(u_i)$, which captures the search history associated with the corrupted reader u_i:

$$\mathsf{UpdtBy}^t(u_i) = \left\{ (w, val) \;\middle|\; \begin{array}{l} (\mathsf{Updt}, u_i, w, val) \in \mathcal{H}^t \\ \wedge \quad u_i \in \mathcal{C} \end{array} \right\},$$

$$\mathsf{SrchBy}^t(u_i) = \left\{ (w, D'(w)) \;\middle|\; \begin{array}{l} (\mathsf{Srch}, u_i, w) \in \mathcal{H}^t \\ \wedge \quad u_i \in \mathcal{C} \end{array} \right\},$$

where $D'(w)$ denotes the set of search results for keyword w.

We now analyze the information leaked under different query types q_t individually.

- In the case where $q_t = (\mathsf{Enroll}, u_i)$, the leakage reveals only the identity of the enrolled user. Therefore, the leakage can be written as $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Enroll}, u_i)$.
- In the case where $q_t = (\mathsf{Revk}, u_i)$, the leakage similarly leaks only the identity of the revoked user. Therefore, the leakage can be written as $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Revk}, u_i)$.
- In the case where $q_t = (\mathsf{Corr}, u_i)$, the corruption of user u_i leaks the secret keys and the interaction history. Specifically:
  - If u_i is a writer, the leakage is updated as: $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Corr}, u_i, \mathsf{sk}_{u_i}, \mathsf{UpdtBy}(u_i))$.
  - If u_i is a reader, the leakage is updated as: $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Corr}, u_i, \mathsf{sk}_{u_i}, \mathsf{SrchBy}(u_i))$.
- In the case where $q_t = (\mathsf{Updt}, u_i, w, val)$. The CS handles the update via the Eureka scheme, which ensures minimal disclosure. Specifically, the CS observes the writer's identity and the received update token: $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Updt}, u_i)$.
  If the writer $u_i \in \mathcal{C}$ is corrupted, the keyword-value pair (w, val) is considered leaked through the corruption interface. This leakage is already accounted for in the corruption query at the time of compromise and is not redundantly included here.
- In the case where $q_t = (\mathsf{Srch}, u_i, w)$. The CS processes the search using Eureka, which prevents exposure of the queried keyword or result contents. However, the CS may learn auxiliary metadata such as the reader identity and the number of matching entries: $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Srch}, u_i, |D'(w)|)$.
  If u_i is a corrupted reader, the full query result $D'(w)$ is leaked via the corruption interface: $\mathcal{L}(\mathcal{H}^t) = \mathcal{L}(\mathcal{H}^{t-1})||(\mathsf{Srch}, u_i, w, D'(w))$.

Proof. We prove the Theorem 2 via a sequence of games that transition from the real experiment $\mathsf{Real}_{\mathcal{A}}^{\mathsf{Archon}}$ to the ideal experiment $\mathsf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathsf{Archon}}$, where $\mathcal{S}$ is a simulator that receives only the leakage function $\mathcal{L}$.

Game G_0. This game is identical to the $\mathsf{Real}_{\mathcal{A}}^{\mathsf{Archon}}$ as defined in Figure 4.

Game G_1. In Game G_1, whenever a PRF evaluation is required, the game responds with a uniformly random string instead of evaluating it using the PRF $F(k,\cdot)$, while maintaining a list of input-output pairs to ensure consistency. If the adversary can distinguish between G_0 and G_1, then one can construct a distinguisher that breaks the pseudorandomness of F by distinguishing it from a truly random function.

Game G_2. This game is identical to Game G_1 except that the hash functions h_1, h_2, and H are now modeled as random oracles. In the random oracle model, the outputs of these hash functions are indistinguishable from random to the adversary. Therefore, no adversary can distinguish G_2 from G_1 with non-negligible probability.

Game G_3. G_3 maintains a simulated keyword count table KCT^* to track the number of updates observable to the adversary. When generating update tokens for $\mathsf{Updt}\mathcal{O}$, G_3 increments KCT^* and re-encrypts it. When generating search tokens for $\mathsf{Srch}\mathcal{O}$, G_3 leaves KCT^* unchanged. If the adversary can distinguish G_3 from G_2, then one can construct a distinguisher that breaks the semantic security of the underlying AES encryption.

Game G_4. G_4 simulates the update token utk for $\mathsf{Updt}\mathcal{O}$. If the writer $u_i \notin \mathcal{C}$ (i.e., not corrupted), G_4 randomly samples $iv^* \in \{0,1\}^*$, $e_w^* \in \mathbb{G}_t$, and $e_{val}^* \in \mathbb{G}_1$. Any non-negligible advantage of $\mathcal{A}$ in distinguishing the simulated e_w^* from the real e_w can be turned into a reduction adversary $\mathcal{B}$ that breaks the DBDH assumption with the same advantage.

Specifically, the reduction adversary $\mathcal{B}$ is given a DBDH challenge tuple $(u_0 = g_0^{\frac{x}{x_u}}, u_1 = g_1^{\frac{x}{x_u}}, v_0 = g_0^{\tau}, z_1 = g_1^{r_w}, e_w)$, where $\frac{x}{x_u}, \tau, r_w \in \mathbb{Z}_p^*$ are randomly chosen exponents, and $e_w \in \mathbb{G}_t$ is either the valid pairing $e(g_0, g_1)^{\frac{x}{x_u} \cdot \tau \cdot r_w}$ or a random element in $\mathbb{G}_t$. The goal of the $\mathcal{B}$ is to distinguish which is the case, using any advantage from the adversary $\mathcal{A}$.

To simulate the environment for $\mathcal{A}$, $\mathcal{B}$ provides u_1 and answers a sequence of up to q hash queries to a programmable random oracle h_1. For each query $w^{(j)}$ with $j = 1, \cdots, q$, the $\mathcal{B}$ samples a fresh $\rho_j \in \mathbb{Z}_p^*$ and sets $h_1(w^{(j)}) = g_0^{\rho_j}$. This enables $\mathcal{B}$ to answer all of $\mathcal{A}$'s queries.

To embed the challenge component $v_0 = g_0^{\tau}$, $\mathcal{B}$ selects a random index $\alpha \in \{1, \cdots, q\}$, and programs the hash oracle at the α-th query to be $h_1(w^{(\alpha)}) = v_0$. If $\mathcal{A}$ never queries $w^{(\alpha)}$, the simulation remains consistent. However, if $\mathcal{A}$ queries this point independently, then $\mathcal{B}$ must abort and fail.

Eventually, $\mathcal{A}$ submits a challenge keyword w. The reduction $\mathcal{B}$ performs an internal query for $h_1(w)$. If it happens that $w = w^{(\alpha)}$, then $h_1(w) = v_0$, and $\mathcal{B}$ can respond with the challenge component e_w as the "encryption" of w.

If e_w is a valid pairing, then the simulation is perfectly consistent with the real scheme. If e_w is random, the challenge appears uniformly distributed. Therefore, under the DBDH assumption, the view of $\mathcal{A}$ is computationally indistinguishable in both cases. Consequently, any non-negligible advantage by $\mathcal{A}$ implies a contradiction to the DBDH assumption.

Similarly, if $\mathcal{A}$ can distinguish the simulated e_{val}^* from the real e_{val}, we can construct a distinguisher that breaks the semantic security of the underlying PRE encryption.

By combining these arguments, we conclude that any advantage an adversary might gain in differentiating the two games is negligible. Consequently, no adversary can distinguish between G_4 and G_3.

Game G_5. G_5 simulates the search token stk and results for $\mathsf{Srch}\mathcal{O}$.

- If the reader $u_i \notin \mathcal{C}$ (i.e., not corrupted), G_5 randomly samples $iv^* \in \{0,1\}^*$, $y^* \in \mathbb{Z}_p^*$, $e_r^* \in \mathbb{G}_t$ and sets $Y^* = g_1^{y^*}$ to form a simulated token stk. Any non-negligible advantage of $\mathcal{A}$ in distinguishing the simulated e_r^* from the real e_r can be turned into a reduction adversary $\mathcal{B}$ that breaks the DBDH assumption with the same advantage. For the search results, G_5 obtains the leakage consisting of the number of matching results $|D'(w)|$ and returns that many random values sampled uniformly from $\mathbb{G}_1$. If the adversary $\mathcal{A}$ can distinguish the simulated search results from the real ones with non-negligible advantage, then we can construct a reduction adversary $\mathcal{B}$ that leverages $\mathcal{A}$ to break the DDH assumption with the same advantage. This reduction follows the standard argument as in the security proof of the hashed ElGamal encryption scheme [48].
- If the reader $u_i \in \mathcal{C}$ (i.e., corrupted), G_5 generates the search token stk and gets the leakage $\mathsf{SrchBy}(u_i)$. For each value val in the search history, G_5 programs the random oracle so that $H(C_0^y) = val \oplus C_1^{\frac{x_u}{x}} \oplus C_2$ on the corresponding inputs.

Since the simulator uses only $|D'(w)|$ and not $D'(w)$ or w, the simulated transcript is independent of the access pattern and search pattern, except for volume leakage. Importantly, the simulator does not use the queried keyword w, the matching identifier set $D'(w)$, or the equality relation among honest-reader queries. It only uses the reader identity and the volume $|D'(w)|$ specified by the search leakage function. Therefore, any two honest-reader searches with the same leakage but different matching sets or different underlying keywords induce computationally indistinguishable views.

By combining the above arguments with the privacy guarantee provided by Eureka, we conclude that no adversary can distinguish G_5 from G_4 except with negligible advantage.

Since the adversary cannot distinguish between G_0 (the real experiment) and G_5 (the ideal experiment), and the simulator receives only the leakage $\mathcal{L}$, which fulfills the requirement in Definition 10 that $\mathsf{Updt}\mathcal{O}$ leaks no information beyond what is already captured by the leakage function, we conclude that Archon is $\mathcal{L}$-adaptively secure and achieves forward privacy. □

Corollary 1. *For honest readers,* Archon *hides the access pattern and search pattern up to the leakage* $(\mathsf{Srch}, u_i, |D'(w)|)$. *In particular, the CS, even when colluding with corrupted users, cannot distinguish two histories* $\mathcal{H}_0$ *and* $\mathcal{H}_1$ *with different honest-reader access patterns or search patterns, provided that* $\mathcal{L}(\mathcal{H}_0) = \mathcal{L}(\mathcal{H}_1)$.

Proof. The corollary follows from Theorem 2. In the simulation of honest-reader search queries, the simulator receives only the leakage specified by the search leakage function, namely the reader identity and the search volume. It does not receive the queried keyword, the exact matching identifier set, or the equality relation among honest-reader queries. Therefore, any distinguisher for honest-reader access patterns or search patterns would distinguish the real execution from the simulated execution, contradicting $\mathcal{L}$-adaptive security. □

Archon achieves strong privacy properties through a combination of cryptographic techniques and system-level design. It is response-hiding: all search results are returned in ciphertext. It also hides the access pattern from the CS by performing a keyword search via a KPIR protocol. To protect the search pattern, each token includes fresh randomness, ensuring that identical queries yield unlinkable tokens. For enhanced

privacy, Archon can optionally perform dummy queries that return dummy results, thereby obfuscating the volume pattern.

The scheme remains secure in adversarial settings where users may be corrupted and collude with the CS. In particular, corrupted writers cannot infer the content or frequency of updates issued by honest writers. This is because each update token is independently randomized using per-user keys and fresh randomness, ensuring unlinkability. Moreover, the secret keys and the update histories of corrupted writers do not aid in compromising the privacy of honest writers, as update tokens are computed using keys that are unlinkable across users and unknown to the adversary. Thus, even with full knowledge of their own credentials, corrupted writers gain no advantage in uncovering the activities of honest writers. Likewise, corrupted readers cannot deduce the queried keywords of honest readers. In the presence of collusion between the CS and corrupted users, Archon preserves the privacy of honest users. Its use of KPIR and randomized token generation thwarts correlation attacks across different users and operations, ensuring robust protection of both data and query privacy.

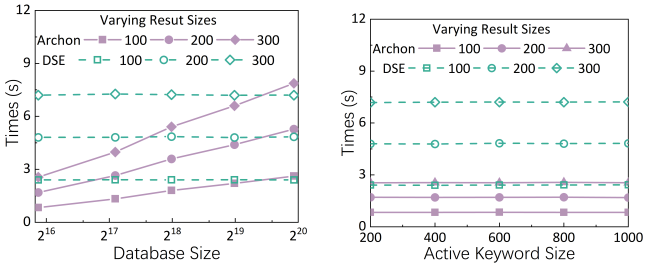
VII. Experiment

We implemented our proposed scheme, Archon, as well as the baseline DSE [10] scheme, in C++. The implementation uses the OpenSSL[6] and PBC[7] libraries for cryptographic primitives and bilinear group operations, respectively. We set the security parameter to $\lambda = 80$, which is consistent with the setting adopted in prior work [41]. In our system, n data items are mapped to n bins (i.e., $m = n$). Under the standard balls-into-bins model with a uniform hash function, it is well known that the maximum bin load is $\Theta\left(\frac{\log n}{\log \log n}\right)$ with high probability. Given that the maximum dataset size in our experiments is 2^{20}, we set the bin expansion factor to 5, which ensures that the probability of bin overflow is negligible. Our experiments were conducted on a TEE-enabled server equipped with a 2.2 GHz Intel Xeon Gold 5520+ CPU, 256 GB RAM, and a 2 TB SSD, running Ubuntu 24.04 LTS. The SGX enclave is configured with a stack size of 4 MB and a heap size of 32 MB.

A. Evaluation with Synthetic Dataset

We evaluate Archon and DSE on a synthetic dataset by systematically varying the database size, the number of keywords, and the number of matching results per keyword, in order to capture the system behavior under diverse parameter settings. This experimental design allows us to investigate how each factor individually affects the efficiency and scalability of Archon. The results are presented in Figures 7 and 8, which summarize the observed performance trends.

Search. Figure 7a shows the keyword search time as the database size increases from 2^{16} to 2^{20}, with the keyword set size fixed at $1{,}000$ and the number of matching documents set to $\{100, 200, 300\}$. As expected, the search time of Archon increases roughly linearly with the database size. This behavior is consistent with the design of the underlying Eureka protocol, where the computation cost grows with the database size. In contrast, the search time of DSE remains almost unchanged as the database size increases, since its search cost mainly depends on the number of matching documents rather than the total database size.



(a) Varying Database Size (b) Varying Active Keyword Size

Fig. 7: Search Performance (Synthetic Datasets)

Figure 7b shows the effect of the keyword set size (ranging from 200 to 1,000) on search time, while the database size is fixed at 2^{16}. The results show that the search time of both schemes remains nearly constant as the keyword set grows, indicating that the search complexity is independent of the keyword set size. Under the same settings, Archon performs significantly better than DSE. This is because Archon mainly uses lightweight XOR operations during search, while DSE requires expensive bilinear pairing operations, which introduce higher computation overhead.

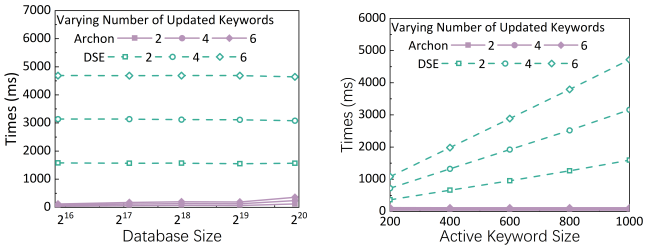
Update. Figure 8a shows the keyword update time with the keyword set size fixed at $1{,}000$, while the database size varies from 2^{16} to 2^{20} for different numbers of updated keywords $\{2, 4, 6\}$. The results show that the update time of Archon increases slightly as the database size grows, because the update procedure needs to refresh all hints that contain the modified data. Fortunately, Eureka uses an iPRF to generate the data indices stored in each hint, which avoids scanning all main and backup hints when locating the ones related to the updated data. In contrast, the update time of DSE does not change with the database size, but its overall cost is much higher than that of Archon. This is because the update complexity of DSE depends on the total size of the keyword set and requires expensive homomorphic operations over the entire keyword set.

Figure 8b shows the update time when the database size is fixed at 2^{16} and the keyword set size varies. As shown in the figure, the update time of Archon remains almost constant as the keyword set size increases. This is because, in our design, the number of generated hints depends on the database size rather than the keyword set size. In contrast, the update time of DSE increases roughly linearly as the keyword set size grows.

B. Evaluation with the Enron Dataset

We evaluate Archon on the Enron email dataset [49], which contains data from approximately 150 users and about 0.5 million emails in total. Since many of the emails are quite short and do not contain key information, we discard emails

[6]https://github.com/openssl/openssl
[7]https://crypto.stanford.edu/pbc/



(a) Varying Database Size (b) Varying Active Keyword Size

Fig. 8: Update Performance (Synthetic Datasets)

with a size smaller than 2KB. We then extract approximately 2.3 million keyword-email pairs.

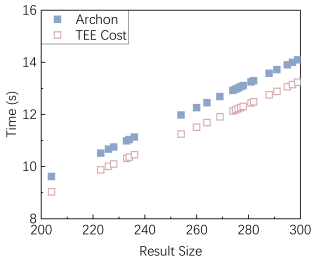


Fig. 9: Search Performance (Enron Dataset)

We select 23 keywords, each matching between 200 and 300 emails. Figure 9 reports the search time and TEE-related overhead for these keywords. As shown in the figure, the TEE overhead accounts for a large portion of the overall search time. This is because the TEE needs to perform several operations during the search process, including the bilinear-operation-based translation of search tokens, the generation of query requests for Eureka, and the re-encryption of the retrieved results so that users can correctly decrypt them.

VIII. Conclusion

M/M SE holds significant promise for collaborative applications but existing approaches fail to address critical security challenges, particularly collusion resistance and user revocation. We introduce Archon, a novel hardware-assisted multi-user SE construction that combines TEE-assisted key management with advanced KPIR techniques.

In Archon, cryptographic keys are split between users and the TEE, ensuring that both search and update tokens require contributions from both components, enabling fine-grained control and dynamic user revocation. As a key component of Archon, we propose Eureka, an updatable KPIR that enables keyword search while concealing access patterns, effectively mitigating collusion risks.

References

[1] H. Kagermann, W.-D. Lukas, and W. Wahlster, "Industrie 4.0: Mit dem internet der dinge auf dem weg zur 4. industriellen revolution," *VDI nachrichten*, vol. 13, no. 1, pp. 2–3, 2011.

[2] J. Guo, Z. Liu, S. Tian, F. Huang, J. Li, X. Li, K. K. Igorevich, and J. Ma, "TFL-DT: A trust evaluation scheme for federated learning in digital twin for mobile networks," *JSAC*, vol. 41, no. 11, pp. 3548–3560, 2023.

[3] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *CSUR*, vol. 47, no. 2, pp. 1–51, 2014.

[4] F. Li, J. Ma, Y. Miao, X. Liu, J. Ning, and R. H. Deng, "A survey on searchable symmetric encryption," *CSUR*, vol. 56, no. 5, pp. 1–42, 2023.

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

[6] P. Xu, Q. Wu, W. Wang, W. Susilo, J. Domingo-Ferrer, and H. Jin, "Generating searchable public-key ciphertexts with hidden structures for fast keyword search," *TIFS*, vol. 10, no. 9, pp. 1993–2006, 2015.

[7] Z. Gui, K. G. Paterson, S. Patranabis, and B. Warinschi, "Swissse: System-wide security for searchable symmetric encryption," *PoPETs*, vol. 2024, no. 1, pp. 549–581, 2024.

[8] T. Le, R. Behnia, J. Guajardo, and T. Hoang, "MUSES: Efficient multi-user searchable encrypted database," in *USENIX Security*, 2024, pp. 2581–2598.

[9] Y. Wang and D. Papadopoulos, "Multi-user collusion-resistant searchable encryption with optimal search time," in *AsiaCCS*, 2021, pp. 252–264.

[10] J. Wang and S. S. Chow, "Unus pro omnibus: Multi-client searchable encryption via access control," in *NDSS*, 2024.

[11] F. Bao, R. H. Deng, X. Ding, and Y. Yang, "Private query on encrypted data in multi-user settings," in *ISPEC*, 2008, pp. 71–85.

[12] Y. Yang, H. Lu, and J. Weng, "Multi-user private keyword search for cloud computing," in *CCTS*, 2011, pp. 264–271.

[13] J. G. Chamani, Y. Wang, D. Papadopoulos, M. Zhang, and R. Jalili, "Multi-user dynamic searchable symmetric encryption with corrupted participants," *TDSC*, vol. 20, no. 1, pp. 114–130, 2021.

[14] J. Wang and S. S. Chow, "Omnes pro uno: Practical multi-writer encrypted database," in *USENIX Security*, 2022, pp. 2371–2388.

[15] T. Le and T. Hoang, "Hermes: Efficient and secure multi-writer encrypted database," in *S&P*, 2025, pp. 2865–2884.

[16] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-abuse attacks against forward and backward private searchable symmetric encryption," in *CCS*, 2023, pp. 3003–3017.

[17] X. Zhang, W. Wang, P. Xu, L. T. Yang, and K. Liang, "High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption," in *USENIX Security*, 2023, pp. 5953–5970.

[18] P. Mondal, J. G. Chamani, I. Demertzis, and D. Papadopoulos, "I/O-Efficient dynamic searchable encryption meets forward & backward privacy," in *USENIX Security*, 2024, pp. 2527–2544.

[19] S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *NDSS*, 2021.

[20] C. Liu, H. Guo, M. Xu, S. Wang, D. Yu, J. Yu, and X. Cheng, "Extending on-chain trust to off-chain–trustworthy blockchain data collection using trusted execution environment (tee)," *TC*, vol. 71, no. 12, pp. 3268–3280, 2022.

[21] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with sgx," in *EuroSec*, 2019, pp. 1–6.

[22] B. Li, F. Zhou, Q. Wang, J. Xu, and D. Feng, "Sedcpt: A secure and efficient dynamic searchable encryption scheme with cluster padding assisted by tee," *JSA*, vol. 154, p. 103221, 2024.

[23] H. Wu, Z. Peng, J. Xiao, L. Xue, C. Lin, and S.-H. Chung, "Hex: Encrypted rich queries with forward and backward privacy using trusted hardware," *TDSC*, vol. 22, no. 4, pp. 3751 – 3765, 2025.

[24] Q. Jiang, E.-C. Chang, Y. Qi, S. Qi, P. Wu, and J. Wang, "Rphx: Result pattern hiding conjunctive query over private compressed index using intel sgx," *TIFS*, vol. 17, pp. 1053–1068, 2022.

[25] X. Yang, K. Li, S. Qi, and H. Zhao, "Practical volume-hiding range searchable symmetric encryption using trusted execution," *FGCS*, vol. 174, p. 107930, 2026.

[26] Y. Zheng, H. Zhu, R. Lu, S. Zhang, Y. Guan, F. Wang, J. Shao, and H. Li, "Secure similarity queries over vertically distributed data via tee-enhanced cloud computing," *TIFS*, vol. 19, pp. 6237–6251, 2024.

[27] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2020.

[28] A. Hoover, S. Patel, G. Persiano, and K. Yeo, "Plinko: single-server PIR with efficient updates via invertible PRFs," in *EUROCRYPT*, 2025, pp. 3–33.

[29] M. Scott, "On the efficient implementation of pairing-based protocols," in *IMACC*, 2011, pp. 296–308.

[30] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," in *EUROCRYPT*, 1998, pp. 127–144.

[31] B. Libert and D. Vergnaud, "Unidirectional chosen-ciphertext secure proxy re-encryption," in *PKC*, 2008, pp. 360–379.

[32] G. Platform, "Global platform made simple guide: Trusted execution environment (tee) guide," *Derniere visite*, vol. 12, no. 04, 2013.

[33] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *CSUR*, vol. 51, no. 6, pp. 1–36, 2019.

[34] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint*

Archive, 2016.
[35] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *EuroSec*, 2017, pp. 1–6.
[36] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution side channels for untrusted operating systems," in *USENIX ATC*, 2017, pp. 299–312.
[37] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P*, 2015, pp. 640–656.
[38] A. Vadapalli, R. Henry, and I. Goldberg, "Duoram: A Bandwidth-Efficient distributed ORAM for 2-and 3-party computation," in *USENIX Security*, 2023, pp. 3907–3924.
[39] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *INFOCOM*, 2018, pp. 1097–1105.
[40] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of sgx and countermeasures: A survey," *CSUR*, vol. 54, no. 6, pp. 1–36, 2021.
[41] L. Ren, M. H. Mughees, and I. Sun, "Simple and practical amortized sublinear private information retrieval using dummy subsets," in *CCS*, 2024, pp. 1420–1433.
[42] M. Zhou, A. Park, W. Zheng, and E. Shi, "Piano: extremely simple, single-server pir with sublinear server computation," in *S&P*, 2024, pp. 4296–4314.
[43] S. Patel, J. Y. Seo, and K. Yeo, "Don't be dense: Efficient keyword pir for sparse databases," in *USENIX Security*, 2023, pp. 3853–3870.
[44] Q. Song, Z. Liu, J. Cao, K. Sun, Q. Li, and C. Wang, "Sap-sse: Protecting search patterns and access patterns in searchable symmetric encryption," *TIFS*, vol. 16, pp. 1795–1809, 2020.
[45] P. Wu, J. Ning, J. Shen, H. Wang, and E.-C. Chang, "Hybrid trust multi-party computation with trusted execution environment." in *NDSS*, 2022.
[46] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *CCS*, 2016, pp. 1143–1154.
[47] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS*, 2017, pp. 1465–1482.
[48] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *cryptology eprint archive*, 2004.
[49] https://www.cs.cmu.edu/~enron/.

Feng Li received the Ph.D. degree from the School of Cyber Engineering, Xidian University, Xi'an, China, in 2024. He is currently a research fellow at the School of Computing and Information Systems, Singapore Management University. His research interests include information security and applied cryptography.

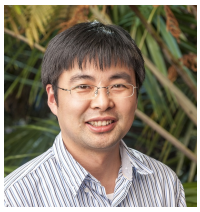
Chengyan Ma received the B.S. degree in computer science and technology from Yangzhou University, China, in 2016, and the M.S. degree and Ph.D. degree in cryptography and cyberspace security from Xidian University, China, in 2019 and 2024, respectively. Now he is a Research Scientist at Singapore Management University. His research interests include unmanned system security, database security.

Xiaoguo Li received his Ph.D. degree in computer science from Chongqing University, China, in 2019. He is currently a Professor with the College of Computer Science, Chongqing University. He worked at Hong Kong Baptist University and Singapore Management University as a Research Fellow from 2019-2024. His current research interests include trusted computing, secure computation, and public-key cryptography.

Pengfei Wu received the Ph.D. degree from Peking University, Beijing, China, in 2020. Now he is an assistant professor with National Engineering Research Center for Software Engineering, Peking University. Before joining PKU, he was a senior research scientist with School of Computing and Information Systems, Singapore Management University and a research fellow with the School of Computing, National University of Singapore. His research interests include cloud data security and privacy-preserving computing.

Lisha Yao received her Ph.D. degree from Jinan University, Guangzhou, China, in 2023. She is currently a research fellow at the School of Computing and Information Systems, Singapore Management University. Her research interests include lattice-based cryptography and applied cryptography.

Jianting Ning received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, in 2016. He is currently a Professor with the Zhejiang Key Laboratory of Digital Fashion and Data Governance, Zhejiang Sci-Tech University; and the Faculty of Data Science, City University of Macau. Previously, he was a Research Scientist with the School of Computing and Information Systems, Singapore Management University, and a Research Fellow with the Department of Computer Science, National University of Singapore. His research interests include applied cryptography and information security.



Guomin Yang received a Ph.D. degree in computer science from the City University of Hong Kong in 2009. He has been an Associate Professor at the University of Wollongong, Australia. He is currently an Associate Professor at the School of Computing and Information Systems, Singapore Management University. His research interests include applied cryptography and privacy-enhancing technologies. He was the Program Co-Chair of ACISP 2018 and ACISP 2022. He is currently serving as some associated editors on IEEE TSC and TCS.

Jianfeng Ma received the Ph.D. degree in computer software and telecommunication engineering from Xidian University, Xi'an, China, in 1995. He was a Research Fellow with Nanyang Technological University, Singapore, from 1999 to 2001. He is currently a Professor and a Ph.D. Supervisor with the Department of Cyber Engineering, Xidian University. His current research interests include information and network security, wireless and mobile computing systems, and computer networks.

Robert H. Deng is AXA Chair Professor of Cybersecurity, Director of the Secure Mobile Centre, and Deputy Dean for Faculty & Research, School of Computing and Information Systems, Singapore Management University (SMU). His research interests are in the areas of data security and privacy, network security, and applied cryptography. He received the Outstanding University Researcher Award from National University of Singapore, Lee Kuan Yew Fellowship for Research Excellence from SMU, and Asia-Pacific Information Security Leadership Achievements Community Service Star from International Information Systems Security Certification Consortium. He serves/served on the editorial boards of ACM TOPS, IEEE Security & Privacy, IEEE TDSC, IEEE TIFS, and Steering Committee Chair of the ACM AsiaCCS. He is a Fellow of IEEE and Fellow of Academy of Engineering Singapore.