Monash University

Efficient Mitigation of Leakage-abuse Attacks for Searchable Encryption

Author
Viet Quang VO

Supervisors
Dr. Xingliang YUAN
Dr. Shi-Feng SUN
A. Prof. Joseph K. LIU

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy at

Monash University

in the

Faculty of Information Technology

Dec 15, 2021

Copyright notice

Abstract

The adaptation of cloud storage nowadays by individual, various governments, and businesses is rapid and inexorable. In such client-server setting, the assumption that the user's data can be protected from insiders of the cloud storage, i.e., database administrators, does not hold. Therefore, security researchers aim to design new practical schemes that allow an untrusted server to search over encrypted database. The research field is known as searchable encryption.

The history of searchable encryption has come a long way from generic cryptographic tools, such as private information retrieval (PIR) and Oblivious RAM (ORAM), to more dedicated privacy-preserving query schemes. Although generic tools provide data confidentiality and can (almost fully) protect the data accesses of client's queries, they have been known as expensive due to a high search latency/computation complexity, or large communication overhead. In contrast, privacy-preserving schemes such as index-based searchable encryption (SE) are more efficient in the client-server communication as well as lower query latency, for which only depends on the query result size, not the database size. However, as a trade-off between security and efficiency, SE reveals necessary information to the server during search, known as acceptable leakage. SE has also been extended to support secure updates of addition and deletion, known as dynamic SE.

The real-world consequences of the acceptable leakage of SE are still being exploited. Leakage-abuse attacks show that a small information leakage known by an attacker can be exploited to compromise the client's query privacy. Unfortunately, the leakage-abuse attacks have not been explored in dynamic SE. Yet, file-injection attacks also exploit the leakage in addition updates of SE to compromise query privacy. Furthermore, the information during deletion updates would also be possible revealed to the server. As a result, advanced security notions of forward and backward privacy are formalised. While forward privacy can prevent such file-injection attacks, the backward privacy defines different types of information leakage known by the server during deletion (i.e., Types I, II, and III). Current forward and backward-private SE schemes still rely on expensive cryptographic tools (i.e., ORAM) to achieve strong forward and backward privacy.

In this thesis, we focus on two aims. First, we improve the security of SE by exploring the leakage-abuse attacks in dynamic setting. Second, we design

new efficient dynamic SE schemes that can achieve strong forward and backward privacy.

With regarding the first aim, we conceptualise the leakage-abuse attacks in dynamic setting by presenting two new threat models of non-persistent and persistent adversaries. We define new constraints to capture the knowledge of these adversaries and provide new security definitions for dynamic SE. Accordingly, we design new padding countermeasures to mitigate them. Then, we develop ShieldDB, an encrypted streaming database with an underlying dynamic SE scheme, and equip it with the padding countermeasures. We show that our proposed padding strategy is practical and deployable to real-world streaming applications/systems that require the privacy preservation on data stream.

Regarding the second one, we carefully analyse the limitations of the prior forward and backward-private constructions, with and without using trusted execution support (TEE). We find that non-TEE constructions suffer high communication overhead between the client and the server due to either multiple round-trip communication or ORAM bandwidth overhead. In addition, existing TEE-supported constructions have high search latency due to the computation bottleneck in the SGX enclave. To resolve this research gap, we provide SGX-SE1 and SGX-SE2 for Type-II backward privacy, and Maiden achieving the strongest backward privacy. We implement prior works and our schemes, and conduct extensive evaluation on the performance under different schemes. The results show that our designs are more efficient in the update operation (addition/deletion) and query latency.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: .....

Print Name: .....

Date: .....

List of Publications

- Vo, V., Yuan, X., Sun, S., Liu, J.K., Nepal, S., Wang, C.: ShieldDB: An Encrypted Document Database with Padding Countermeasures. In IEEE Transactions on Knowledge and Data Engineering. (2021)
- Vo, V., Lai, S., Yuan, X., Sun, S., Nepal, S., Liu, J.K.: Accelerating Forward and Backward Private Searchable Encryption Using Trusted Execution. In Proceedings of 2020 Applied Cryptography and Network Security, pages 83-103.
- Vo, V., Lai, S., Yuan, X., Nepal, S., Liu, J.K.: Towards Efficient and Strong Backward Private Searchable Encryption with Secure Enclaves. In Proceedings of 2021 Applied Cryptography and Network Security, pages 50-75.

Other publication in the Ph.D. course

- Sun, S., Yuan, X., Liu, J.K., Steinfeld, R., Sakzad, A., Vo, V., Nepal, S.: Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), pages 763-780.

Acknowledgments

This thesis would not have been possible without blessings, encouragement and support of a number of people.

Foremost, I wish to express my most sincere gratitude and appreciation to my main supervisor, Dr Xingliang Yuan, for his passionate guidance, patience and encouragement throughout my Ph.D. study at Monash University. All his valuable advice have helped me perform to the best of my abilities. Without his endless support, this dissertation would not have existed. He helped in various aspects even well before the commencement of my Ph.D.

I thank my co-supervisors Joseph K. Liu and Shi-Feng Sun for their help and support during my Ph.D. journey. They did not hesitate to provide their help whenever needed. I still remember the time when Joseph K. Liu advised me with my Ph.D. choices prior the commencement of my Ph.D. Shi-Feng Sun is a great supervisor and collaborator, who also invited me to join his project which later paved the way for my Ph.D. study.

I would also like to express my gratitude to my Data 61 mentor, Dr. Surya Nepal, who closely follows my Ph.D journey. His valueable suggestion always makes the papers' writing and presentation more solid.

I owe great thanks to many friends and colleagues at Monash University, including Shangqi Lai, Cong Zuo, Dimaz Wijaya and Maxime Buser. Shangqi Lai is a great collaborator who works very hard and always brings useful discussion during paper writing and experiment analysis. Also, neither my Ph.D. would have been as enjoyable as it is without their friendship.

I am grateful to Data61, CSIRO for providing me a stipend scholarship throughout the course of my Ph.D. study, and supporting me to attend conferences and similar events in multiple occasions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Searchable encryption (SE) investigates the problem of outsourcing encrypted data to an untrusted server and enabling the server to search against the encrypted data. As the server searches on its storage, it is clear that SE generates some unavoidable leakage information for which the server can learn. This information include the query tokens sent by the client and the list of encrypted documents as the search result. If the search is repeated, the repetition of the search results is also revealed. Note that, the leakage does not include the content of the documents in the search results as the client decrypts them locally. The security of a SE scheme is guaranteed by not allowing the server to perform any inference relating to client's query and the underlying database, beyond the acceptable leakage during search operations. A SE scheme is dynamic if it allows the server to dynamically update the encrypted database. Naturally, we also want the dynamic SE reveals at least as possible the leakage during search and update operations to the server. As SE addresses the practical problem of encrypted database, it is desired that new dynamic SE schemes with minimal leakage should be efficient so as to deployable in practice.

In this chapter, we will more precisely describe the motivation behind efficient mitigation of leakage-abuse attacks in dynamic SE and formalise research gaps that define this thesis.

1.1 The need for searchable encryption

The adaptation of cloud storage by various governments and businesses is rapid and inexorable. From small businesses to large interprises, they find that outsourcing the storage and management of their data to the third party is more convenient and cost effective. As a result, such adaptation has brought many benefits to the global economy and the productivity improvement for the businesses. Yet, storing ever daily growing amount of data in the cloud also has caused a huge upheaval in the cloud infrastructure development.

Unfortunately, only protecting the communication between users to the cloud servers, by using network security measures, e.g., TLS [4], does not fully protect the confidentiality of sensitive data stored in the storage. The servers still see outsourced plaintext data and certainly are able to trace what data the users have queried and updated over the time, such as query keywords, and keywords in newly added/deleted documents. It has been showing that data breaches in cloud servers are happening quite frequently in recent time, affecting millions of individuals [5, 6, 7, 8, 9]. This phenomenon calls for increased control and security for private and sensitive data stored in the untrusted servers [10, 11, 12, 13].

To combat against "breach fatigue" at untrusted cloud servers and enhance user's privacy, comprehensive efforts have been made by both law and regulation makers, and security researchers. Digital data protection law enforcement, such as the Gramm-Leach-Bliley Act, the Health Insurance Portability and Accountability Act [14], the European Union General Data Protection Regulation (GDPR) [15], and Australian Privacy Act [16], mandate involving institutions and businesses to protect customer information by implementing proper security measures. From security perspective, it is clear that using private-key encryption to encrypt the customer's sensitive data at the client side before outsourcing to the cloud can guarantee the security of outsourced data against compromised storage servers and data breaches. However, such simple scheme design is not practical at both the client and server sides. Indeed, considering a health care center as a client of a cloud service provider, the client with its master secret key, is able to upload encrypted documents of health records of its patients, employee contracts, and confidential business contracts to the cloud. In order to make queries or updates, the client has to download and decrypt the entire database, then retrieve or update appropriate documents, before re-encrypting and outsourcing the encrypted data again to the server. Clearly, the client performs queries and updates, not the server. This solution is impractical as it requires the client's storage and computation power comparable to the hardware configuration of the cloud server, and large communication bandwidth overhead (i.e., database size) between the client and the server. Hence, the shift to computational and high-capacity cloud servers motivates security researchers to design new practical schemes that allow the server to search over encrypted data, i.e., searchable encryption.

1.2 The history of searchable encryption

As for preserving encrypted search, encrypted database systems can be designed by using generic cryptographic tools or specific schemes supporting different query functionalities. In this section, we briefly overview these constructions in terms of security and efficiency, either of computation or communication overhead.

The first tool is private information retrieval (PIR) [17] and its variants of

information theoretic (IT-PIR) [18] and computationally-PIR (cPIR) [19]. At a high level, the protocol enables the client to privately retrieve a data record from a server, without the server learns which result selected. With IT-PIR, it is necessary to duplicate the database to non-colluding servers so as to the servers collectively retrieve the query result. It is known that the client does not have to download the whole database for every query. Such protocol does not rely on any hardness security assumption. However, the secrecy of the query is guaranteed when the servers are maintained by competitive service providers in practice. With cPIR, single-database computational PIR schemes [20, 21] trade a low communication bandwidth for a high computational complexity, i.e., $\Omega(N)$ public key operations to answer an query in the database of size N. This prevents cPIR from being practical to support a very large database as shown in [22].

Another privacy-preserving tool is Oblivious RAM (ORAM), which was first introduced by Goldreich and Ostrovsky et. al. [23]. At a high level, the scheme enables data re-encryption and shuffling after every data access. Traditional ORAM-based constructions [24, 23, 25, 26] only require a single server, which is considered as untrusted. The scheme protects the data confidentiality and data access pattern on the server. The server is implemented as a regular search engine to manage its storage, and the storage can be layouted by a hierarchical/square root [24, 23] or a tree-based design [25, 26]. In those constructions, only symmetric encryption is used since the client decrypts and re-encrypts data. Since the noticeable Path ORAM construction proposed by Stefanov et. al. [26], ORAM design has been optimised in various directions to improve efficiency (e.g., server's computation [27, 24, 28], bandwidth blowup between the server and its storage [29, 30], query throughput between the client and the server [31, 32]). The running time of ORAM-based constructions only depends on the number of results, unlike the dependence of database size of cPIR schemes. However, ORAM-based constructions (e.g., Path ORAM) subject to a lower bound of client-server bandwidth overhead as mentioned in the seminal works [26, 23]. That is, an encrypted data block requires the server to return $\mathcal{O}(\log N)$ blocks to the client, in the case of the server does not do any computation. The client can perform data updates on the fetched data before evicting them back to the server side. It is clear that ORAM-based constructions reveal the number of data accesses to the untrusted server, for which the server can infer the search result length.

We have seen that above generic tools, i.e., PIR-based and ORAM-based constructions, suffer inefficiency of either computation depending on the database size (i.e., N) or the communication overhead depending on $\mathcal{O}(\log N)$. Thus, another generation of encrypted database systems had found to overcome such lower bounds, and it received a wide attention [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]. In this generation, many works [33, 46, 34, 47, 48, 49, 50, 36, 51, 52] implement property-preserving encryption (PPE) in a way that a ciphertext inherits equality and/or order properties of the underlying plaintext. However,

inference attacks can compromise these encryption schemes by exploiting the above properties preserved in the ciphertexts [53, 54].

In parallel, dedicated privacy-preserving query schemes are investigated intensively in the past decade for encrypted databases [38, 48, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]. Among others, searchable encryption (SE) [65, 66, 67, 68] is well known for its applications to ubiquitous keyword based search. In general, SE schemes utilise an encrypted index to enable the server to search over encrypted documents. The server is restricted such that only if a query token (keyword ciphertext) is given, the search operation against the index will be triggered to output the matched yet encrypted documents. This ensures that an adversary with a full image of the encrypted database learns no useful information about the documents. In that sense, SE outperforms PPE in terms of security. We call a database is static if the state of the database remains unchanged over the time. In 2006, Curtmola et al. [66] provided the first index-based SE scheme to achieve a sublinear search time in a static database. The study also provided necessary acceptable leakage of SE (a.k.a access to the database during search) and fundamental security definitions for SE. We note that the leakage contains access and search patterns if the database is static. Informally, the access pattern of a query token reveals the identifiers of documents containing the query keyword and the number of documents (i.e., query result length). The search pattern reveals the repetition of query tokens and results if the client queries matching documents for the same query keywords. The security of a SE scheme is guaranteed by not allowing the server to learn any information relating to client's query and the underlying database, for that beyond the acceptable leakage during search operations.

In 2012, SE had been extended to support secure updates (addition/deletion) on dynamic databases. Kamara et al. [69] proposed the first dynamic SE scheme with sublinear search time. However, the scheme leaked the hashes of keywords contained in the updated documents. Then, Cash et al. [70], conducted another work to optimise searching operations for very large databases.

1.3 Leakage exploits in searchable encryption

Although SE provides an efficient solution compared to traditional encrypted databases using generic tools (e.g., PIR [19], homomorphic encryption [33, 34] and oblivious RAMs [71, 72]), it has a trade off between security and efficiency, which is necessary acceptable leakage as defined by Curtmola et al. [66]. The real-world consequences of the leakage of SE are still being exploited. Since 2012, more works tend to explore the adversarial capability of knowing SE's leakage to break its claimed security.

Islam et al. [73] and Cash et al. [74] were the first studies show that a small information leakage known by an attacker can be exploited to compromise

the client's query privacy. In details, Cash et al. [74] proposed a practical attack that exploits the leakage in the search operation of static SSE. It is assumed that an adversary with full or partial prior knowledge of DB can uncover keywords from query tokens via search results. Namely, the adversary identifies the matching between the search results to known documents to infer the query keywords. This assumption is not impractical if we think the adversary can obtain the plaintext data in previous data breaches. Recent works [74, 75] show that using padding countermeasure is simple but effective to make adversaries harder when uncovering query keywords. However, the leakage-abuse attacks and current padding efforts were only investigated in static databases, and have not been investigated in real-world dynamic database applications. In this dynamic setting, the state of database changes over the time. Specifically, the updates of documents change the access pattern for a given keyword, and new keywords can be introduced randomly at any time.

Not just the leakage-abuse attacks that can break the claimed security of SE, the claimed secure update of dynamic SE is also overlooked. In 2016, Zhang et al. [76] proposed the first instantiation of active attacks called file-injection attacks through the exploitation of the leakage in data addition. At a high level, if an adversary can reuse a known query token to find out her injected documents, it is clear that the keyword used to generated that token must be in those added documents. The attacks are successful due to the fact that the query token of the keyword does not reflect the change of newly matching documents. Yet, if data deletion on the encrypted database is allowed, a dynamic SE scheme should not reveal the subsequent query keywords in previous updates. Otherwise, it reveals the history of document deletion of the client.

Since 2016, many dynamic SE schemes focus on the forward and backward privacy introduced by Stefanov et al. [77] and later formalised by Bost et al. [78, 75]. The reason is because forward privacy can mitigate adaptive file injection attacks [76] as it guarantees the updates cannot be associated with prior queries [78, 75, 79]. On the other hand, backward privacy ensures that the queries do not link to deleted documents.

Although forward and backward privacy are desirable, recent works [75, 80] demonstrate that forward and strong backward-private (Type-I and Type-II) schemes are difficult to achieve in an efficient way in practice. Namely, new efficient SE schemes have to hide the access pattern on updated data while only revealing the currently matching documents to the server. One can think of re-using expensive cryptographic primitives, e.g., ORAM, to achieve that security goal. However, this design introduces formidable computation and communication costs, which bring stupendous obstacle in deploying SE in practice.

1.4 Contributions of this Thesis

The history of SE has come a long way to formalise secure data updates on encrypted databases. However, it has raised doubt whether dynamic SE is secure to protect the privacy of queries and the confidentiality of the underlying database against existing leakage-abuse attacks [74, 76].Yet, although the security notions of forward and backward privacy are formalised [75], existing dynamic SE schemes still rely on generic tools, e.g., ORAM or trapdoor permutations, to achieve these notions [75, 80]. Thus, it is desired to answer whether dynamic SE with these advance notions are applicable in practice if ones care about the efficiency of client-server bandwidth, client's storage, server's computation, and search latency. Otherwise, dynamic SE is provably secure, but not practical. Therefore, this thesis aims to firmly answer the following research question.

How to design efficient dynamic SE schemes with less leakage to mitigate existing and even prevent prospective active attacks?

In this thesis, we study and investigate the practical security and efficiency of dynamic SE against attacks that exploit the inherent leakage of update and search operations. It brings in at once new constructions without using generic expensive tools (e.g., ORAMs [23, 26], PIR [17]), new security considerations, new theoritical results and practical efficiency analysis of dynamic SE against the attacks. These results come from three papers, that we quickly summarise into following sections.

1.4.1 Efficient mitigation against leakage-abuse attacks in dynamic SE [1]

As explained above, SE leaks some necessary information about data accesses to the server in static database. Leakage-abuse attacks [74] had been developed, exploiting this leakage to compromise the claimed security of SE. Indeed, these attacks helped to understand the importance of considering leakage when analysing the security of SE as well as when deploying SE in practice. In 2017, Bost et al. [75] raised the necessity of using constraints to formalise the background knowledge of adversary in the security definition of SE. Unfortunately, the work was still theoritical, without further investigating empirical results of how to mitigate the attacks. More importantly, SE research community only investigated the security and countermeasures of SE against the attacks in static database setting, but did not formalise the security of SE against the attacks and how efficiently mitigate them in dynamic database setting.

Our paper [1] addressed these questions by proposing new attack models of leakage abuse attacks and formalising them in new security definitions. By using these security definitions, we devised new efficient countermeasure approaches and applied them in a real streaming encrypted database system, called as ShieldDB, in order to hide the query results even the adversary has the back-

ground knowledge of database over the time. We provided the provable security of two new schemes with specific leakage profiles in that setting. We also provided the experiment results to demonstrate the efficiency of our countermeasures and showed that the design can be easily adapted to any SE scheme. The contribution summary of ShieldDB is as follows.

- ShieldDB is the first encrypted database that supports encrypted keyword search, while equipping with padding countermeasures against inference attacks launched by adversaries with database background knowledge.
- We define two new types of attack models, i.e., non-persistent and persistent adversaries, which faithfully reflect different real-world threats in a continuously updated database. Accordingly, we propose padding countermeasures to address these two adversaries.
- ShieldDB is designed with a dedicated system architecture to achieve the functionality and security goals. Apart from the client and server modules for encrypted keyword search, a Padding Service is developed. This service leverages two controllers, i.e., Cache Controller and Padding Controller, to enable efficient and effective database padding.
- ShieldDB implements advanced features to further improve the security and performance. These features include: 1) forward privacy that protects the newly inserted document, 2) flushing that can reduce the load of the padding service, and 3) re-encryption that refreshes the ciphertexts while realising deletion and reducing padding overhead.
- We thoroughly investigate the security of ShieldDB against the proposed non-persistent and persistent adversaries.
- We present the implementation and optimization of ShieldDB, and deploy it in Azure Cloud. We build a streaming scenario for evaluation. In particular, we implement an aggressive padding mode (*high* mode) and a conservative padding mode (low mode), and compare them with padding strategies against non-persistent and persistent adversaries, respectively. We perform a comprehensive set of evaluations on the load of the cache, system throughput, padding overhead, and search time to demonstrate its practical deployment.

1.4.2 Efficient forward and backward-private SE schemes with trusted execution environment [2, 3]

With our previous paper [1], we saw why it is desirable that update (i.e., addition) in dynamic SE should not reveal any information about updated keywords even

though the adversary has the background knowledge of the dynamic database. Otherwise, the data addition updates and query keywords are revealed to the server over the time. Noting that dynamic SE also supports secure deletion, not just addition update [69], and the security notions of forward and backward privacy were formalised by Bost et al. [75]. The backward privacy notion contains different types of information leakage for which a semi-honest server can learn about historial data deletion from Type-I to Type-III. This leakage is inherently generated from SE if it supports secure deletion before search operations. More precisely, the server can distinguish at what time which document identifiers of the query keywords added and then deleted before the keyword is searched. Thus, it is important to design new schemes such that deleted results remained hidden to the server.

There have been many SE schemes supporting the advance security notion of backward privacy (e.g., Type-I with Moneta and Orion [75], Type-II with Fides and Mitra[75], Type-III with Janus [75], Horus [80], and Janus++ [81]). However, Type-I schemes Moneta and Orion rely on ORAM-based constructions, and current Type-II schemes Fides [82] and Mitra [80] require multiple roundtrips and high communication cost, while Horus [80] relies on Path-ORAM [26]. Until recently, Amjad et al. [83] proposed the first forward and backward private schemes using trusted execution environment (TEE) (i.e., Intel SGX [84]). As generic ORAM or ORAM-like data structures can natively be adapted to achieve the strongest forward and backward privacy (i.e., Type-I backward privacy [82]), one of their schemes is built from ORAM, where data addition and deletion are completely oblivious to the server [83]. It is noteworthy that such an approach could still be inefficient due to the high I/O complexity between the SGX and server. Like prior forward and backward private SE studies, Amjad et al. [83] also proposed Type-I and Type-II schemes, Fort and Bunker-B, respectively. Unfortunately, only the theoretical constructions of the scheme are given in [83], and we observe that they are not scalable, especially when handling large datasets to support very large document updates. Therefore, we aim to explore how to use TEE to design new strong forward and backward-private SE.

- We design and implement two forward and backward private SE schemes, named SGX-SE1 and SGX-SE2. By using SGX, the communication cost between the client and server of achieving forward and backward privacy in SE is significantly reduced.

- Both SGX-SE1 and SGX-SE2 leverage the SGX enclave to carefully track keyword states and document deletions, in order to minimise the communication overhead between the SGX and untrusted memory. In particular, SGX-SE2 is an optimised version of SGX-SE1 by employing Bloom filter to compress the information of deletions, which speeds up the search operations and boosts the capacity of batch processing in addition and deletion.

- We formalise the security model of our schemes and perform security analysis accordingly.
- We conduct comprehensive evaluations on both synthetic and real-world datasets. Our experiments show that the latest art Bunker-B takes 10× more *ecall/ocalls* than our schemes SGX-SE1 and SGX-SE2 when inserting 10^6 documents. Even more, Bunker-B needs 30× *ecall/ocalls* when deleting 25% of the above documents. W.r.t. search latency, SGX-SE1 and SGX-SE2 are 30% and 2× faster than Bunker-B, respectively.
- Regarding Type-I backward privacy, we thoroughly analyse a basic scheme named Orion*, i.e., direct migration of the latest strong backward-private DSSE scheme Orion [80] to TEE, and the latest art of TEE-based scheme Fort [83]. We identify their limitations both theoretically and empirically. As the implementation of Fort and Orion* is not available, we implement them from scratch for evaluations and comparisons.
- We propose Maiden, the first Type-I backward-private scheme without relying on ORAM. Maiden is designed to keep the states of updates, the deletion information, and a sketch of insertions inside TEE, so as to eliminate the leakage in updates and allow minimally necessary leakage during the search. We formalise the security model of the scheme and perform security analysis accordingly.
- We conduct comprehensive evaluations on our proposed scheme Maiden, Fort, and Orion*. Our experiment shows that the addition throughput of Maiden is 13 $\sim$ 36× higher than Orion*. Maiden takes a negligible time to perform document deletion. The search latency in Maiden is 70 $\sim$ 90× faster than Fort and Orion* when using a large synthesis dataset. With a real-world dataset, Maiden is 575× and 291× faster than those schemes, respectively.

1.5 Thesis Structure

In this thesis, chapter 2 highlights the most related works of this Ph.D project. The chapter first covers the fundamental background of searchable encryption (SE) and summarises research directions in practical SE deployment. Then, we highlight leakage-abuse attacks in static SE and leakage exploits in the dynamic SE. After that, we present significant works that leverages the trusted execution environment (TEE) to enable encrypted search. Chapter 3 revisits formal definition of security notions relevant to this study, then highlight related schemes.

In Chapter 4, we explore the leakage-abuse attacks in dynamic (streaming) setting and how to mitigate them effectively. In section 4.1 we first overview the

design of ShieldDB. Then, we demonstrate new proposed threat models that includes non-persistent and persistent adversaries in section 4.2. Section 4.3 demonstrates padding countermeasures and other optimisation features supported in the system. Section 4.4 analyses the security of the system. In section 4.5, we present the system deployment and perform intensive evaluation to investigate the performance of the proposed padding strategies. Then, we discuss the performance and assumption of the system in section 4.6.

In chapter 5, we explore how to improve the efficiency of dynamic SE schemes that supports advanced security notions of forward and backward privacy. First, we discuss related SGX-supported schemes in section 5.1. Then, in section 5.2, we present our implemented system that supports our new schemes. In details, in section 5.4, we detail our design for SGX-SE1 and SGX-SE2 that efficiently achieve Type-II backward privacy. The evaluation of these schemes are presented in section 5.5. In section 5.6, we present our design for Maiden, which achieves Type-I backward privacy. Then, in section 5.7, we evaluate Maiden and compare it with other related works. Then, in section 5.8 discusses SGX side-channels and how existing countermeasures can be applied to our design.

Chapter 6 concludes the contributions of this thesis and opens future research directions.

Chapter 2

Literature Review

This chapter summarises the state of the art in the research area relevant to our contribution further as of the time of writing (June 2021). We start with searchable symmetric encryption (SSE), which forms the basis of many schemes described in the thesis. SSE enables an untrusted server to search over encrypted documents upon client's request with *acceptable* information leakage. Then, we discuss leakage-abuse attacks in SSE and other information leakage revealing to the server during updates in dynamic SSE (DSSE). Finally, we highlight some significant works that leverage trusted execution environment (TEE) to enable encrypted search. The discussion in this chapter is kept informal, and relevant formal definitions are given in the following and subsequent chapters[1].

2.1 Searchable Encryption

We consider an immediate application of SSE is cloud storage system, which contains a trusted client and an untrusted server. In setup, the client would like to encrypt its document collection and then to outsource the encrypted database to the server for storage purpose. Then, in search operation, the client would like to retrieve currently matching documents of query keyword w. To do so, the client generates and sends a query token for w which the server uses to run the search operation and returns appropriate (encrypted) documents. We call such setting, which only contains setup and search operations, as *static* SSE. Clearly, in that setting, we note that multiple search operations can repeatedly happen against the static encrypted database maintained by the server. In contrast, we call a SE scheme as dynamic SSE (DSSE) if it additionally supports update operations, which dynamically enables the server to perform *secure* updates (addition/deletion) on the encrypted database.

At the beginning, the problem of SSE can be resolved by using *oblivious* RAM that offers the strongest privacy guarantee [23, 25, 26, 85]. More precisely, the

[1]This chapter is partly based on [1, 2, 3]

technique does not reveal *any* information to the server, including the update or search operations, and even the "access pattern" (i.e., which documents contain w). Unfortunately, the approach requires a logarithmic (in the database size) the number of rounds of interaction between the client and the server and has a high overhead at each side.

In the year 2000, SSE was first considered explicitly by Song et al. [65]. The work provided a non-interactive scheme that achieves a linear search time in the length of the database by weakening the privacy guarantee. Informally, according to [65, 69, 66], a SSE scheme is secure if: (1) the encrypted database alone (without query tokens) reveals no information about the underlying data; (2) the database together with the query token reveals at most the result of the search to the serverl; (3) query tokens can only be generated using the client's secret key. Note that, during search, the server learns the "access pattern" and the "search pattern" of the query upon receiving the query tokens, where the "access pattern" reveals the list of documents containing the query keyword, and the "search pattern" reveals the repetition of queries.

After that, in 2006, Curtmola et al. [66] provided the first two *index*-based static SSE schemes, namely SSE-1 and SSE-2, against *non-adaptive* and *adaptive* adversarial models, respectively. They both only require one communication round between the client and the server. Informally, the terms of *index*, *non-adaptive*, and *adaptive* adversarial models are explained as follows. The data structure *index* stores the identifiers of documents (*id*s) in the database while supporting efficient keyword search. That is, given the query keyword w, the index returns pointers pointing to the documents containing w. The security definition of *non-adaptive* security model refers to adversaries that only make search queries independently from query tokens and search results of previous queries. In contrast to *non-adaptive* security model, the *adaptive* adversaries can choose their queries as a function of previously obtained query tokens and search results. Although Curtmola et al. [66] showed that SSE-1 achieves sublinear search time, which the asymptotic complexity is proportional to the number of documents currently matching the query keyword w (not the whole database size like [65]), the scheme only provides security against *non-adaptive* adversarial model. In contrast, SSE-2 achieves the stronger notion of *adaptive* security, but the scheme's search time costs an asymptotical search time depending on the total number of documents in the database. During the period from 2005 to 2012, supporting DSSE was not considered explicitly. One cannot add or remove files without either treating the newly added data as a completely new databaset or requiring re-indexing approaches. For instance, Chang et al. [86] considered the new set of encrypted documents to be added as a separate document collection. Therefore, the client needs a different query token to query that newly added database since it is indexed using different secrets. Yet, Chase et al. [87] requires re-indexing the entire database to support the updates. Alternatively, Curtmola et al. [66] considered a technique that asks the server to return the

data duplication of the previously and newly added databases so that the client can do re-indexing.

Until 2012, Kamara et al. [69] formalised the security definition for DSSE that supports secure updates and proposed the first practical DSSE scheme satisfying all the following properties: sublinear search time, security against the *adaptive* adversarial model, compact indexes and supporting document addition/deletion. In a high level, the scheme requires the server to store two physical arrays which have the same size, including a search array and a delete array. While the search array stores the linked list of each keyword's matching document list, the delete array forms the link lists between keywords in the same documents. Then, document addition/deletion require homomorphically updates on the pointers in those two arrays.

Since DSSE formalised, the research area has attracted a long line of studies to propose many diffrent schemes that improve different levels of query efficiency, support search functionalities. Some highlighted works in these directions are briefly described as follows.

Improving query efficiency: Most SSE schemes have optimal search times that scale with the number of documents matching the query. Practical factors of I/O latency, storage utilisation, and database distribution downgrade the performance of in-memory SSE in practice. However, when scaling SSE to big data using external memory, Cash et al. [70] showed that non-contiguous reads to memory create a throughput-bottleneck on very large databases. Hence, previous works [70, 88, 89, 90] achieved optimal locality by increasing read efficiency—the number of additional memory allocation (false positives) that the server reads per result item. Later, Demertzis et al. [91] provided new SE schemes with tunable locality to enable the tradeoffs between space, read efficiency, locality, and communication overhead. Soon after, Asharov et al. [92] and Demertzis et al. [93] provided new SSE schemes with built-in memory allocation algorithms to adapt keywords for different document size ranges. These works helped SSE to achieve linear space, constant locality, and sublogarithmic read efficiency. In another direction, Demertzis et al. [94] improved the efficiency of SSE schemes by compressing the list of document identifiers matching to keywords before encryption. As a result, it reduces the accessed result identifier list without compromising security. The proposed solution is compatible with any existing SSE scheme as a black-box. Another approach to improve search efficiency in large-scale database is using distributed SSE [45].

Supporting search functionalities: SSE also has been extended to enable range and boolean queries [95, 70, 69, 60, 96, 97, 98]. Hence, improving the efficiency of SSE schemes supporting those expressive queries has been advanced in recent years [99, 100, 101, 102, 103]. For instance, Li et al. [57] proposed the first range query processing scheme by organising indexing elements in a complete binary tree and with traversal depth and width minimisation algorithms. The query processing efficiency was achieved at the worst complexity of $O(|R|\log n)$,

where n is the total number of data items and R is the set of data items in the query result. Until recently, Wu et al. [41] proposed a secure verifiable and efficient framework to support encrypted multi-dimensional range query by developing a new hierarchical cube based method to pre-process dataset and query range, respectively. Then, an encrypted index based on tree structure is built to achieve $O(R)$ query time complexity. The efficiency of SSE schemes supporting conjunctive search and boolean queries also have been investigated to support very large databases. Cash et al. [95] provided the first interactive sublinear SSE scheme to achieve conjunctive keyword search with the complexity is independent of the number of documents in the database. Instead, it scales with the number of documents matching the least frequent keyword in the conjunction. After that, Sun et al. [104] proposed an efficient non-interactive multi-client SSE scheme with support for boolean queries by enabling the client to obtain the search-authorised private key from data owner on permitted search keywords. Other significant range-supporting SSE works are highlighted in [105, 41, 97, 106].

2.2 Leakage-abuse Attacks in Static SSE

Typically, efficient SSE schemes like [95, 70, 69] often expose some information, called *leakage*, to the server. This information can be statistically obtained via the "access pattern" (i.e., which documents contain query keywords) and "search pattern" that reveals repeated queries. In 2012, Islam, Kuzu, and Kantarcioglu [73] (IKK) provided the first leakage-abuse attacks that study the emperical security of SE in practical deployment. The study shows that a client's query keywords can be guessed when the underlying plaintext database DB is known to the untrusted (honest-but-curious) server. In a high level, the IKK attacks leverage simulated annealing technique to probabilistically match queries' results with the keywords in the database based on the access patterns revealed during search operations.

In 2015, Cash et al. [74] proposed a significantly simpler, faster, and more accurate attack called as the count attack. The attack exploits the leakage in the search operation of SSE. It is assumed that an adversary with full or partial prior knowledge of DB can uncover keywords from query tokens via *access pattern*. Specifically, the prior knowledge allows the adversary to learn the documents matching a given keyword before queries.

In the meanwhile, padding countermeasures [73, 107, 75] are considered as an effective approach to obfuscate the leakage during search operations of SE. In particular, Islam et al. [73] propose the first padding countermeasure for SE; keywords are grouped into different clusters, where each keyword in a cluster matches a set of identical document ids. This requires another data structure to help the client to differentiate real and bogus document ids after search, since all bogus ids are selected from the real ones. After that, Cash et al. [107] propose another approach; the number of ids in each keyword matching list is padded up to

the nearest multiple of an integer, aka padding factor. To guarantee effectiveness, this factor needs to be increased until no unique result size exists. However, this padding factor is a system-wide parameter, and incrementing it introduces redundant padding for all other padded matching lists. To reduce padding overhead, Bost and Fouque [75] propose to pad the keyword matching lists based on clusters of keywords with similar frequency. Their proposed clustering algorithm achieves minimised padding overhead while thwarting the count attack in the static setting. Very recently, Xu et al. [108] investigate the formal method to quantify the padding security strength, and propose a padding generation algorithm which makes the bogus and documents similar. Again, all the above padding countermeasures focus on the static setting, where the dataset remains unchanged after the setup. We note that the assumption in this setting is not always true in practice due to the dynamic changes of keywords.

We also note that there is another research direction that focuses on recovering the client's search queries using volumetric leakage (i.e., query result lengths) [109, 110]. As a result, volume-hiding schemes like [111, 112] are proposed recently. However, we note that those schemes are focused on the *static* setting, as they resort to specialised data structures and constructions. First, they are not dynamic friendly. In [111, 112], multi-hashing and cuckoo hashing techniques are adopted as the underlying data structures. It is not easy to insert new data into those data structures, and we are not aware any existing volume-hiding schemes support efficient updates. Second, volume hiding schemes may hide the size of the query result, but it is not clear whether they can protect the relationships between different query keywords when applying them into the context of keyword search. Also, there is no clear evidence in the literature that volume hiding schemes can defeat leakage-abuse attacks (i.e., the generalised count attacks) against SSE.

2.3 Leakage Exploits in DSSE

Since the leakage-abuse attacks proposed in 2015, Zhang et al. [76] continously investigated the new consequences of leakage in SE schemes via file-injection attacks in 2016. In this active attacks, the adversary (or the untrusted server) sends her crafted (known) documents to the client as the input for the update (addition) operation. Then, the client follows the protocol to encrypt and upload them to the server as defined by a DSSE scheme. After that, if the client issues the same (known) query tokens to the server and the corresponding query results additionally contain the injected documents, the adversary can learn the keywords searched in those injected documents. Compared to prior IKK attacks [73] or leakage-abuse attacks [74], Zhang et al. [76] showed that the file-injection attacks are very effective when recovering a high fraction of query keywords by just injecting a small number of documents. In addition, the attacks do not require the auxiliary knowledge of the database.

Forward privacy is an advanced security notion of DSSE to mitigate the powerful file-injection attacks. Informally, *forward privacy* prevents the server from using the previous query tokens to retrieve newly added documents. Although Stefanov et al. [77] investigated the security for the first time in an Oblivious RAM-based scheme, but it was not efficient due to the large communication bandwidth and the server's storage blowup. Then, in 2016, Bost et al. [78] formally defined *forward privacy*, and designed the first insertion-only forward-private scheme (i.e., Sophos). In Sophos, the client locally keeps the keyword's latest state and uses that state as an input when generating the new encrypted entry for the newly updated (w, id) pair in update. In search, the client generates the query token based on the latest state. Note that, the state is continously re-generated for every (w, id) pair of the query keyword by using the inverse of one-way trapdoor permutation based on public key cryptography. Then, upon receiving the token, the server can recover all previous states from the given public key. Since the server does not know the private key, it cannot guess future states. Unfortunately, Bost et al. [78] showed that the public key operation is the performance bottleneck in the scheme. In 2018, Song et al. [113] provided Fast and FastIO that only use symmetric-key based trapdoor permutation. In a high level idea, the ephermeral key of the permutation is embedded inside the encrypted database so that the server can use it to recover the state of previously added entries. We refer readers to [113] for the detailed protocols and formal security analysis. Other recent forward-private SSE shemes also include [114, 115, 79].

Another important notion of privacy in DSSE is *backward privacy*. Informally, deleted documents cannot be revealed to the server in any subsequent search queries. At the beginning, the idea of hiding deleted documents was experimentally evaluated in oblivious RAM by Stefanov et al. [77] in 2014. Then, in 2017, Bost et al. [82] formally defined the security notion of *backward privacy* that categories different leakage information on the updates of the query keyword that the server can learn upon receiving the search token of the keyword. There are three types of backward privacy from Type-I to Type-III in the descending order of security. Informally, Type-I only leaks the timestamps when the currently matching documents added into the database, and the total number of updates on that query keyword. Then, Type-II additionally reveals the timestamps when all the updates on the query keyword happened. Type-III is the least secure, it additionally leaks the confliction pattern that reveals when documents matching the keyword added and then deleted before the search operation. Since 2017, many different backward-private schemes have been proposed [82, 81, 80, 98], which target different leakage types. However, strong backward-private (Type-I and Type-II) schemes are known to be inefficient in the computation and communication overhead between the client and the server. For example, Type-I backward-private schemes all rely on ORAM, including Moneta [82] and Orion [80]. Some significant works for Type-II schemes include Fides [82] and Mitra [80]. However, both of them require multiple roundtrips

and high communication cost, while Horus [80] still relies on ORAM [26]. In a less secure manner, Type-III schemes include Janus [82] and Janus++ [81]. They only require one roundtrip as a tradeoff between communication cost and security guarantees.

2.4 Encrypted Search Based on Trusted Execution Environment

Trusted execution environment (TEE) like Intel SGX is a set of x86 instructions designed for improving the security of application code and data executed on the untrusted server. On SGX-enabled platforms, ones need to partition the application into both trusted part and untrusted part. The trusted part, dubbed enclave, is located in a dedicated memory portion of physical RAM with strong protection enforced by SGX. The untrusted part is executed as an ordinary process and can invoke the enclave only through the well-defined interface, named *ecall*, while the enclave can encrypt clear data and send to untrusted code via the interface named *ocall*. Furthermore, decryption and integrity checks are performed when the data is loaded inside the enclave. All other software, including OS, privileged software, hypervisor, and firmware cannot access the enclave's memory. The actual memory for storing data in the enclave is only up to 96 MB. Above that, SGX will automatically apply page swapping. SGX also has a remote attestation feature that allows to verify the creation of enclaves on a remote server and to create a secure communication channel to the enclaves.

Since TEE commercialised [116], there has been an active line of research [83, 117, 118, 119] that leverage the hardware support to enable search over encrypted data. In general, TEE such as Intel SGX can reduce the network roundtrips between the client and server and enrich the database functions in the encrypted domain. Fuhry et al. [119] proposed HardIDX that organises database index in a B^+-tree structure and utilises enclave to traverse a subset of the tree nodes to do searches. The scheme achieves search time complexity at the logarithm in the size of index, but it does not support dynamic update operations. Later, Mishra et al. [118] designed a doubly-oblivious SE scheme that supports inserts and deletes, named Oblix. In this scheme, one oblivious data index resides in the enclave to map the search index of each keyword to a location in another oblivious structure located in untrusted memory. However, the performance of their implementation on large databases is less efficient due to the fact of using ORAM. To support boolean queries, Borges et al. [102] migrated secure computation to the enclave to improve the search efficiency. When two or more keywords are queried, the result set can be unionised or intersected within the enclave. Note that this work focuses on a different problem with ours. However, none of the previous works leverage the TEE to efficiently support the advanced security notions (i.e., *forward* and

backward privacy of dynamic SSE. Until recently, Amjad et al. [83] proposed three schemes to enable single-keyword query with different search leakage (i.e., information that the server can learn about the query and data). They are, the TEE-supported Type-I scheme Fort, Type-II scheme Bunker-B, and Type-III scheme Bunker-A.

However, the work only provided theoritical schemes without investigating their practical efficiency to support very large document addition/deletion. Fort requires an oblivious map (OMAP) similar to the one in Orion [80] to do the update, causing high computation overhead. Bunker-A [83] improves the update computation, but it downgrades the security guarantees. Bunker-B also suffers practical limitations of intensive communication between the trusted execution environment (i.e., enclave) and the server, and search latency due to re-encryption. Also, the computation/communication of these three TEE-supported schemes have not been investigated in the work [83]. Hence, strong backward private (Type-I and Type-II) schemes are known to be inefficient in both computation and communication overhead. Therefore, we are motivated to explore how to design efficient schemes using TEE to achieve strong *backward privacy*.

Chapter 3

Preliminaries

This chapter covers the preliminary made use of in the next chapters. First, we highlight commonly used notations and crytographic primitives in DSSE. Then, we cover the security background of DSSE by presenting key security notions and fundamental security definition. After that, we detail the leakage-abuse attacks in static SSE, which serves as the stepstone for our investigation regarding the attacks in the DSSE. Then, we present the fundamental an Index-based DSSE scheme, which serves as the basic scheme for many shemes later supporting advanced notions of forward and backward privacy. After that, we detail the formal security of forward and backward privacy.

3.1 Notations and Cryptographic Primitives

3.1.1 Notations

We follow the formalisation of Curtmola et al. [66], Kamara et al. [69], and Cash et al. [70] to restate the key notations used in (dynamic) SSE as follows.

Security parameter
We write $\{0,1\}^n$ to denote the set of all binary strings of length n, and the set of all finite binary strings as $\{0,1\}^*$. We denote by $\mathsf{Func}[n,m]$ the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. We write $[n]$ to represent the set of integers in the range $\{1,\ldots,n\}$. For a finite set X, we write $x \xleftarrow{\$} X$ to represent an element x being sampled uniformly from X. We also write $x||y$ to denote the concatenation of two strings x and y.

A symmetric key K is a string of λ bits, and a key generation algorithm uniformly samples K in $\{0,1\}^\lambda$. In (dynamic) SE, we only consider probabilistic algorithms and protocols executing within the time polynomial in the security paramater λ. Thus, adveraries are considered as probabilistic polynomial time algorithms. We can write $x \leftarrow \mathcal{A}$ to denote the output x of a probabilistic algorithm $\mathcal{A}$. Throughout, $\lambda \in \mathbb{N}$ refers to the security parameter and we consider

that all algorithms take λ as input. Thus, a function $\nu : \mathbb{N} \mapsto \mathbb{N}$ is negligible in λ if for every positive polynomial $p(\cdot)$ and a large λ, $\nu(\lambda) < 1/p(\lambda)$. We write $f(\lambda) = negl(\lambda)$ to mean that there exists a negligible function $\nu(\cdot)$ such that $f(\lambda) \leq \nu(\lambda)$ for all sufficiently large λ. Thus, two distribution ensembles X and X' are computationally indistinguishable if for all probabilistic polynomial-time (PPT) distinguisher $\mathcal{D}$, we have:

$$|\Pr[\mathcal{D}(X) = 1] - \Pr[\mathcal{D}(X') = 1]| \leq negl(\lambda)$$

In (dynamic) SSE, a trusted user is modelled as probabilistic polynomial-time Turing machines, while the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$ are modeled as deterministic polynomial-size circuits. Since every probabilistic polynomial-time algorithm can be simulated by a deterministic polynomial-size circuit [66], (dynamic) SSE schemes guarantee the security against any probabilistic polynomial-time adversary.

Data Structures
A database $\mathsf{DB} = (id_i, \mathsf{W}_i)_{i=1}^{d}$ is a tuple of d-tuple of identifier/keyword-set pairs where $id_i \in \{0,1\}^\lambda$ and $\mathsf{W}_i \subseteq \{0,1\}^*$. The set of keywords of the database is $\mathsf{W} = \cup_{i=1}^{D} \mathsf{W}_i$, where D is the number of documents in DB. We write $W = |\mathsf{W}|$ to represent the total number of keywords, and $N = \Sigma_{i=1}^{D} |\mathsf{W}_i|$ to denote the number of document/keyword pairs in DB. We denote by $\mathsf{DB}(w)$ the set of the identifiers of the documents containing the keyword w, i.e., $\mathsf{DB}(w) = \{id_i | w \in \mathsf{W}_i\}$. Then, an inverted index can be presented as $M_I = \{\mathsf{DB}(w_i)\}$, where $i \in [1, W]$. Typically, the index is encrypted and then outsourced to the untrusted server to enable encrypted search functionality. We refer readers to seminal works [66, 69] for more details regarding how *static* SE schemes are constructed.

Throughout this report, we also mention other data structures of linked list and arrays, and dictionaries. Thus, we also present relevant notations of these data strutures as follows. If A is an array (or a linked list) then we denote by $|A|$ the total number of elements in A, and $A[i]$ is the value stored at the index $i \in |A|$. We also write $A[i] \leftarrow v$ to denote the assignment operation that assigns v at that index. A dictionary M (aka. a key-value map) is a data struture that stores key-value pairs (k, v). If (k, v) in M, then $M[k]$ is the value v associated with k.

3.1.2 Basic Cryptographic Primitives

Symmetric encryption scheme
A symmetric encryption scheme contains three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, where Gen is a probabilistic algorithm that takes the security parameter λ and outputs a secret key K, Enc is a probabilistic algorithm that takes the inputs of K and a message m and outputs a ciphertext c, and Dec takes the inputs of c and the key K and returns m if c was the encrypted message of m

by using the same key. Informally, SKE is chosen-plaintext attacks (CPA)-secure if the ciphertext does not reveal any useful information about the underlying plaintext even to an adversary that can query to an encryption oracle.

Pseudo-random functions
In addition to encryption schemes, SE also commonly uses pseudo-random functions (PRF or F), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We restate the formal definition from [66] as follows.

Definition 1. *A function* $f : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$ *is pseudo-random if it is computable in polynomial time (in* λ*) and if for all polynomial-size* $\mathcal{A}$*,*

$$\left|\Pr\left[\mathcal{A}^{f_K(\cdot)} = 1 \ : K \xleftarrow{\$} \{0,1\}^k\right] - \Pr\left[\mathcal{A}^{g(\cdot)} = 1 \ : g \xleftarrow{\$} \mathsf{Func}[n,m]\right]\right| \leq negl(k)$$

where the probabilities are taken over the choice of K *and* g*.*

We note that in practice these primitives may be built out based on off-the-shelf primitives. For example, we may build a symmetric encryption using AES in counter mode and a pseudorandom function using the CBC-MAC.

3.2 Security Definitions for DSSE

In this section, we first introduce common security notions in a generic DSSE scheme that supports Search and Update to demonstrate the interaction between a client and a server. After that, we present an existing dynamic add-only index-based SSE scheme that efficiently supports very large database [70].

3.2.1 Security notions

A DSSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ consists of one algorithm and two protocols between a client and a server:

- $\mathsf{Setup}(\lambda, \mathsf{DB})$ is an algorithm that takes a security paramater λ and a database DB as input. Then, it outputs a pair (EDB, K, ST) where K is a secret key, EDB is the encrypted database, and ST is the state of keywords. The server maintains EDB, while the client keeps locally K and ST.

- $\mathsf{Search}(K, w, ST; \mathsf{EDB}) = (\mathsf{Search}_C(K, w, ST), \mathsf{Search}_S(q, \mathsf{EDB}))$ is the protocol, where the client runs $\mathsf{Search}_C(K, w, ST)$ to outputs the query token q for the query keyword w. Then, the server executes $\mathsf{Search}_S(q, \mathsf{EDB})$ to returns matching documents containing w to the client. In the scope of this report, a search query is restricted to a unique keyword w.

- $\mathsf{Update}(K, ST, \mathsf{op}, \mathsf{in}; \mathsf{EDB}) = (\mathsf{Update}_C(K, ST, \mathsf{op}, \mathsf{in}), \mathsf{Update}_S(\mathsf{EDB}))$, where the client takes the input of the key K, the state ST, and an operation $\mathsf{op} = \{add, del\}$ meaning adding or deleting a document with an input in parsed as that document identifier id and a set of keywords W in that document. The client generates add/del tokens q and then the server executes $\mathsf{Update}_S(q, \mathsf{EDB})$ with the input of EDB upon receiving q.

We now borrow common notions from Bost et al. [78] to formulate the interaction between the client and the server in the scheme Σ.

Definition 2. (Access pattern). *Let Q be the list of all queries issued by the client, w be a query keyword in W, DB be a database of identifier/keyword-set pairs. The access pattern induced by a search query $q \in Q$ on w at timestamp i: $\mathsf{ap}(w) = (i, \mathsf{DB}(w))$, is the entry containing $\mathsf{DB}(w)$, a collection of document identifiers whose documents containing w.*

We also note that schemes leaking [70] the repetition of query token q to the server also reveal the repetition of thw queried keyword w. More formally, this leakage is named as search pattern.

Definition 3. (Search pattern). *Let Q be the list of all queries issued by the client, and whose entries are (i, w) for a search query on the keyword w at timestamp i. The search pattern induced by the repetition of the search query on w : $\mathsf{sp}(w) = \{i : (i, w) \in Q\}$, is the collection of different timestamps when the client uses the same search query for (i, w).*

Another important notation is query pattern, formally defined by Bost et al. [78]. The notation formalises the repetition of updated keywords in $\mathsf{Updates}$.

Definition 4. (Query pattern). *Let Q be the list of all queries issued by the client, and whose entries are (i, w) for a search query on the keyword w at timestamp i, or $(i, \mathbf{op}, \mathbf{in})$ for an $\mathbf{op}$ update query with input $\mathbf{in}$. The query pattern is formulated as:*

$$\mathsf{qp}(w) = \{i : (i, w) \in Q \text{ or } (i, \mathbf{op}, \mathbf{in}) \text{ and } w \text{ appears in } \mathsf{in}\}$$

DSSE schemes often leak more or less the above information to the server, in order to gain the efficiency. Thus, the security of DSSE schemes often guarantee no more information can be revealed to the server beyond the acceptable leakage [66, 69]. To capture the stateful leakage information, DSSE works often define leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch})$ and provide $\mathcal{L}$ to the simulator $\mathcal{S}$. Here, we provide the security definition for the generic scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ as follows.

Definition 5. (Adaptive Security). *Let $\Sigma =$(Setup,Search,Update) be a DSSE scheme, $\mathcal{A}$ be an adversary, $\mathcal{S}$ be a simulator, and $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch})$ be the stateful leakage function. We define two following probabilistic games as follows.*

- $\boldsymbol{Real}^{\Sigma}_{\mathcal{A}}(\lambda)$: *$\mathcal{A}$ choose a database* DB. *The game then runs* $\mathsf{Setup}(\lambda, \mathsf{DB})$ *and returns* EDB *to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively choose queries q_i in Q. If q_i is a search query for keyword w, the game executes the query by running* $\mathsf{Search}(K, w, ST; \mathsf{EDB})$ *and returns the transcript to $\mathcal{A}$. If q_i is an update query, the game answers the query by running* $\mathsf{Update}(K, ST, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$. *Finally, $\mathcal{A}$ outputs the bit of the game $b \in \{0, 1\}$.*

- $\boldsymbol{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda)$: *$\mathcal{A}$ choose a database* DB. *Given the leakage $\mathcal{L}^{Stp}$ in* Setup, *the simulator $\mathcal{S}$ generates an encrypted* $\mathsf{EDB} \leftarrow \mathcal{S}(\mathcal{L}^{Stp}(\mathsf{DB}))$ *and gives it to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively runs the query set Q. If q_i in Q is a search query, the simulator returns the transcript to $\mathcal{A}$ by running $\mathcal{S}(\mathcal{L}^{Srch}(q_i))$. If q_i is an update query, the simulator answers the query by running $\mathcal{S}(\mathcal{L}^{Updt}(q_i))$. Finally, $\mathcal{A}$ returns a bit $b \in \{0, 1\}$ that the game uses as its own input.*

We say Σ is an $\mathcal{L}-$adaptively-secure scheme if for any probabilistic polynominal-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that:

$$\left|\Pr\left[\boldsymbol{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\boldsymbol{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right]\right| \leq negl(\lambda)$$

3.2.2 An Index-based DSSE scheme

In this section, we first represent the a basic dynamic construction Π^{+}_{bas} [70] as an example to demonstrate how to support changes (i.e., additions) to the database. We then investigate the security analysis of the scheme. Note that the scheme serves as a stepstone to highlight the need for advanced notions of forward and backward privacy in following chapters.

High level design:
We note that Π^{+}_{bas} extends the basic static SSE scheme Π_{bas} presented by Cash et al. [70]. However, to support dunamic updates (i.e., additions), Π^{+}_{bas} requires the client to keep the keyword's state in a local map $ST[w]$ such that the latest state is used to generate a new update token when there is a new document (identified by id) containing w to be added. Then, in search, the client generates query tokens for the keyword w and the server will leverage the tokens to identify whether there is any entry in the encrypted database matching to a possibly generated state and the tokens. The scheme is formulated in Algorithm 1. We describe the scheme as follows.

In Setup, Π^{+}_{bas} requires the client to create K, K^{+} that are used to generate tokens for the existing DB in Setup and later for dynamic Updates. The client also generates a list L as a collection for encrypted entries to be outsourced to the server in Setup. In addition, the state map ST is to track the latest states of keywords. Then, given DB, the client generates encrypted entries using K and (w, id) pairs in $\mathsf{DB}(w)$.

Algorithm 1 Scheme Π^{+}_{bas} [70]

$\underline{\mathsf{Setup}}$ $(1^\lambda, \mathsf{DB})$

Client

$K, K^+ \xleftarrow{\$} \{0,1\}^\lambda$, allocate a list L, and init a map $ST \leftarrow \emptyset$
for each $w \in W$ **do**
  $K_1||K_2 \leftarrow F(K, w)$
  Initialise counter $c \leftarrow 0$
  for each $id \in \mathsf{DB}(\mathrm{w})$ **do**
    $l \leftarrow F(K_1, c); d \leftarrow \mathsf{Enc}(K_2, id); c++$
    Add (l, d) to the list L (in lex order)
  end for
end for
Send L to *Server*

Server

Set $\mathsf{EDB} = L$

$\underline{\mathsf{Update}}$ $(\mathsf{op} = add, \mathsf{in})$

Client

Init a list $L' \leftarrow \emptyset$
for $(w, id) \in \mathsf{in}$ **do**
  $K_1^+||K_2^+ \leftarrow F(K^+, w)$
  $c' \leftarrow ST[w]$; if $c' = \perp$ then $c' \leftarrow 0$
  $l' \leftarrow F(K_1^+, c'); d' \leftarrow \mathsf{Enc}(K_2^+, id); ST[w]++$
  add (l', d') to L'
end for
Send L to *Server*

Server

Add entries in L' to EDB

$\underline{\mathsf{Search}}$ (w)

Client

Compute $K_1||K_2 \leftarrow F(K, w)$ and $K_1^+||K_2^+ \leftarrow F(K^+, w)$
Send (K_1, K_2, K_1^+, K_2^+) to *Server*

Server

for $c = 0, c' = 0$ **do**
  $d \leftarrow (\mathsf{EDB}[F(K_1, c)])$; $id \leftarrow \mathsf{Dec}(K_2, d)$; $c++$
  $d' \leftarrow (\mathsf{EDB}[F(K_1^+, c')])$; $id' \leftarrow \mathsf{Dec}(K_2^+, d')$; $c'++$
  Output id and id'
  Continue until no entry found by both labels d and d'
end for

In Update, for new (w, id') to be added, the client leverages K^+ to generate new entries like the ones in Setup.

Intuitively, during Search, the server repeats the search process twice for different states c and c'key using tokens (K_1, K_2) and (K_1^+, K_2^+), respectively.

Security Analysis

The security of Π_{bas}^+ can be quantified via a stateful leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Add}, \mathcal{L}^{Srch})$. It defines the information exposed in Setup, Update (i.e., addition), and Search, respectively. We restate the leakage functions of Π_{bas}^+ and its proof sketch as presented in [70].

In Setup, upon an initial input database DB, $\mathcal{L}^{Stp} = \Sigma_{w \in W} |\mathsf{DB}(w)|$, presenting the size of the database.

Next, we investigate the leakage information in Search and Update. Let Q be the list of all queries issued by the client to the server so far, where an entry of Q is of the form $(i, \mathsf{op}, \dots)$, and ID be the set of all document identifiers in DB. The search pattern is defined as $\mathsf{sp}(w, Q) = \{i : (i, w) \in Q\}$ of a keyword w with respect to Q to be the timestamps of the queries searching for w. The access pattern $\mathsf{ap}(w) = (i, \mathsf{DB}(\mathrm{w})$ as defined in Def. 2.

For a document with identifier id and its keyword set W_{id}, let the addition pattern of (w, id) *with respect to* Q be the timestamp i when the pair is added:

$$\mathsf{ap}'(id, w, Q) = \{i : (i, \mathsf{add}, id, W_{id}) \in Q, w \in W_{id}\}$$

We note that the Π_{bas}^+ is a deterministic SSE. The reason is because the search protocol is deterministic with respect to query tokens of keywords regardless Updates operations over timestamps (see Algorithm 1). In search query for w, let the addition pattern AP(w,Q,ID) *with respect to* Q be the set of all document identifiers to which w was added. We see that the server can also learn the $\mathsf{ap}'(id, w, Q)$, which indicates when documents in $\{id\}$ were added. The reason for that is because the serve can rewind the Updates and re-run the Searches in the timestamp order using the deterministic query tokens. Therefore, the addition pattern of keyword w *with respect to both* Q *and* ID be the set of all document identifiers to which w was added along with the timestamps when they were added. Formally,

$$\mathsf{AP}(w, Q, \mathsf{ID}) = \{(id, \mathsf{ap}'(id, w, Q)) : id \in \mathsf{ID}, \mathsf{ap}(id, w, Q) \neq \emptyset\}$$

For a Search(w), Π_{bas}^+ reveals the search pattern, access pattern, and the addition pattern to the server. Formally, $\mathcal{L}^{Srch}(w) = \{\mathsf{sp}(w, Q), \mathsf{ap}(w), \mathsf{AP}(w, Q, \mathsf{ID})\}$

We now investigate $\mathcal{L}^{Updt}$. For an update query $(i, \mathsf{op}{=}\mathrm{add}, id, W_{id})$ in Q, $\mathcal{L}^{Updt}$ outputs $|W_{id}|$, (i.e., the number of keyword/document pairs in the document), and the set of following search patterns. It denotes the repetition of id in non-empty search patterns.

$$\{\mathsf{sp}(w, Q) : w \in W_{id}\}$$

Algorithm 2 The count attack algorithm [74]

Input: Unencrypted index Index, query tokens t and results

Initialise known query map K with queries (q, k) having unique result lengths
Compute the co-occurance counts C_q for observed queries and C_I for Index
while *size of K is increasing* **do**
    for *each unknown query q in* $(t - K)$ **do**
        Set candidate keywords $S \subseteq K = \{s : \mathsf{count}(s) = \mathsf{count}(q)\}$
        for $s \in S$ **do**
            for *known queries* $(q', k) \in K$ **do**
                if $C_q[q, q'] \neq C_I[s, k]$ **then**
                    remove s from S
                end if
            end for
        end for
        if *one word s remains in S* **then**
            add (q, s) to K
        end if
    end for
end while

We note that Π^{+}_{bas} is only secure against non-adaptive attacks. In particular, the adversary cannot repeat/choose a query for the keywords that were not previously searched for. For the formal security analysis of Π^{+}_{bas}, we refer readers to [70]. However, with regarding to adaptive attacks, if the adversary chooses the keyword that has been searched before, the server can learn the presence of the keyword in newly added documents. The reason is because the server can reuse its collected query tokens before to query against the newly added documents. This leakage is inherent when the query tokens used for searching the same keyword are deterministically generated and the same all the time.

3.3 Leakage-abuse Attacks - The count attacks

The count attack algorithm [74] is given in Algorithm 2. Based on the prior knowledge DB of the adversary, she can construct a keyword co-occurrence matrix indicating keyword coexisting frequencies in known documents. As a result, if the result length $|\text{DB}(w)|$ for a query token tk is unique and matches with the prior knowledge, the adversary directly recovers w. For tokens with the same result length, the co-occurrence matrix can be leveraged to narrow down the candidates. Experimentally, for the used Enron email dataset, the count attack achieves a perfect reconstruction for recovering the 500 most frequent query keywords since 63% of them have a unique result count. In addition, the attack only takes few seconds to run, compared with hours for the IKK attack. Clearly, the count

attack primarily relies on the unique occurrence of some queries' result lengths to infer the remaining queries. Therefore, padding countermeasure can reduce some of the number of uniqueness.

To overcome the limitation of the count attack, Cast et al. [74] generalised it by introducing two modifications. First, the generalised attack makes initial guesses that allow a candidate keyword sets could map to a query tokens and then run the count attack's remaining algorithm. For a given candidate keyword, if there is an inconsistency found later in the co-occurrence counting, eliminate it and move on. Second, the attack marks candidate keyword sets to a query token for which their counts matches within a given maximum number of false co-occurrences, rather than exact match (i.e., window size) as used in the original count attack. The generalised count attack also demonstrated a perfect query construction rate when a padding countermeasure is applied with the padding overhead of 1.2. Note that the padding overhead is the ratio between the total number of real and padding pairs over the real number of pairs.

Prior to the work of this thesis, there was not a study investigating the leakage-abuse attacks (i.e., the generalised count attacks) in dynamic setting where the DB's state changes over the time. In particular, investigating to what extends the attacks can exploit in DSSE and how to efficiently mitigate them have not been studied.

3.4 Forward-secure DSSE

Forward privacy is an advanced security notion in DSSE. Informally, it prevents the Updates from revealing any information about the updated keywords. In particular, the server should not learn that newly updated documents matches keywords that had been searched previously. The informal definition of *forward privacy* was first presented by Stefanov et al. [120], then it was formalised by Bost et al. [78] in 2016. Intuitively, if a scheme is not *forward-private*, the search token is deterministic and can be re-used by the server to retrieve documents added after the token being issued. This is exactly the leakage exploited by the file-injection attacks proposed by Zhang et al. [76] in 2016. If the adversary has partial knowledge of the database, the attacks can reveal previously query keywords by just tricking the client into encrypt $log2T$ new documents containing chosen keywords. If the adversary does not have such knowledge, the number of required documents is about $W/T + logT$, where W is the total number of keywords and T is a threshold parameter (e.g., $T = 200$ as used in [76]). With *forward privacy*, the attacks can be prevented since the previous query tokens do not relate to subsequent document addition.

Here, we restate the formal definition of *forward privacy* from [78] as the following:

Definition 6. (Forward privacy). *A $\mathcal{L}$-adaptively-secure DSSE scheme Σ is forward private if the update leakage $\mathcal{L}^{Updt}$ can be written as*

$$\mathcal{L}^{Updt}(\mathsf{op},\mathsf{in}) = \mathcal{L}'(\mathsf{op}, \{(id_i, \mu_i)\})$$

where $\{(id_i, \mu_i)\}$ is the set of modified documents paired with the number μ_i of modified keywords for the updated document id_i.

The definition shows the restricted $\mathcal{L}^{Updt}$ revealed to the server during Updates. In particular, the document addition does not reveal any information about their modified keywords. Therefore, we can see that the dynamic add-only scheme Π^{+}_{bas} [70], presented in section 3.2.2, fails to achieve *forward privacy*. The reason is because that the search tokens in Π^{+}_{bas} are deterministic, and $\mathcal{L}^{Updt}$ of the scheme reveals the search patterns of keywords in the newly added documents.

3.5 Backward-secure DSSE

Backward privacy limits the information of the deleted documents containing the keyword w that the server can learn upon subsequent search queries on w. The idea of this security notion was first proposed by Stefanov et al. [120], then it was formalised by Bost et al. [82] in 2017. There are three types of *backward privacy* based on the amount of information that the server can learn regarding the historial deletion upon the search queries. Considering the following sequence of updates and search, in the timestamp order:
$(t = 1, \mathsf{add}, id_1, \{w_1, w_2\})$, $(t = 2, \mathsf{add}, id_2, \{w_1\})$, $(t = 3, \mathsf{del}, id_1, \{w_1\})$,
$(t = 4, \mathsf{search}, w_1)$

We demonstrate the differences between these leakage types from the above example as following:

Type-I *backward privacy*: is the most secure. It only leaks the *insertion pattern*. That is, the identifiers of documents *currently* matching the query keyword w (i.e., id_2). In addition, the server knows that there are 3 updates on w, but it does not know the corresponding timestamps.

Type-II *backward privacy*: is less secure. It inherits the leakage from Type-I *backward privacy*, and it additionally leaks the *update pattern*. That is, the timestamps when the updates on w happended (but not their content). In particulate, the server knows w was updated at timestamps $t = 1$, $t = 2$, and $t = 3$.

Type-III *backward privacy*: is the weak secure. It inherits the leakage from Type-I *backward privacy*, and it additionally leaks the *cancellation* between deletion update and insertion update. In this example, it additionally reveals w was added at $t = 1$ and then deleted at $t = 3$.

We now introduce leakage functions that help to formulate these different types of *backward privacy*. Giving a list of queries Q sent by the client, the server

records the timestamps u for every query with $Q = \{q : q = (u, w) \text{ or } (u, \mathsf{op}, \mathsf{in})\}$. Following the verbatim from [78, 82], we let $\mathsf{TimeDB}(w)$ be the access pattern which consists of the non-deleted documents *currently* matching w and the timestamps of inserting them to the database. Formally,

$$\mathsf{TimeDB}(w) = \{(u, id) : (u, add, (w, id)) \in Q \text{ and } \forall u', (u', del, (w, id)) \notin Q\}$$

and let $\mathsf{Updates}(w)$ be the list of timestamps of updates:

$$\mathsf{Updates}(w) = \{u : (u, \mathsf{op}, (w, id)) \in Q\}$$

To capture the weakest notion of *backward privacy*, Bost et al. [82] defined $\mathsf{DelHist}$ that reveals the *cancellation* between deletion update and insertion update the can. Intuitively, $\mathsf{DelHist}(w)$ contains the list of timestamps of all deletion operations and inserted entries containing $w$it removes. Formally, $\mathsf{DelHist}(w)$ is constructed as:

$$\mathsf{DelHist}(w) = \{(u^{\mathsf{add}}, u^{\mathsf{del}}) : \exists id \; s.t. (u^{\mathsf{del}}, \mathsf{del}, (w, id)) \in Q \text{ and } (u^{\mathsf{add}}, \mathsf{add}, (w, id)) \in Q\}$$

With these functions, the three notions of *backward privacy* proposed in [82] can be defined as follows.

Definition 7. (Backward privacy). *Let a_w be the number of documents currently matching w, and $\mathsf{TimeDB}(w)$ be the access pattern which consists of the non-deleted documents currently matching w, $\mathsf{Updates}(w)$ be the list of timestamps of updates, and $\mathsf{DelHist}(w)$ be the list of timestamps of all deletion operations and inserted entries containing $w$it removes. A $\mathcal{L}$-adaptively-secure SE scheme Σ is insertion pattern revealing backward-private iff the search and update leakage $\mathcal{L}^{Srch}, \mathcal{L}^{Updt}$ can be written as:*

$$\mathcal{L}^{Updt}(\mathsf{op}, w, id) = \mathcal{L}'(\mathsf{op})$$

$$\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), a_w)$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

A $\mathcal{L}$-adaptively-secure SE scheme Σ is update pattern revealing backward-private iff the search and update leakage $\mathcal{L}^{Srch}, \mathcal{L}^{Updt}$ can be written as:

$$\mathcal{L}^{Updt}(\mathsf{op}, w, id) = \mathcal{L}'(\mathsf{op}, w)$$

$$\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{Updates}(w))$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

A $\mathcal{L}$-adaptively-secure SE scheme Σ is weakly backward-private revealing backward-private iff the search and update leakage $\mathcal{L}^{Srch}, \mathcal{L}^{Updt}$ can be written as:

$$\mathcal{L}^{Updt}(\mathsf{op}, w, id) = \mathcal{L}'(\mathsf{op}, w)$$

$$\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), \mathsf{DelHist}(w))$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

Chapter 4

Leakage-abused Attacks in Dynamic SSE and Efficient Mitigation

As a noteworthy threat in the static SSE, the leakage-abuse attacks [74] show that an adversary with full or partial knowledge of database can uncover keywords from query tokens via the query results. The attacks directly exploits the leakages of SSE to break the protection on data confidentiality. Using oblivious-RAM (ORAM) is an quintessential approach to enable encrypted search without exposing access pattern [71, 72], but it is shown as an expensive tool [73, 74, 118]. Alternatively, using padding (bogus documents) for inverted index solution [66] is proven as a conceptually simple but effective countermeasure to obfuscate the access pattern against the aforementioned attacks [73, 74, 75]. Unfortunately, existing padding countermeasures only consider a static database, where padding is only added at the setup [75, 74]. They are not sufficient for real-world applications using DSSE. That is a stream setting where the states of database change over time, and the updates of documents also change continously. Therefore, in this chapter, we first explore to what extend adversaries can exploit the leakage in the streaming setting to compromise the privacy of data. Then, we explore how padding countermeasures can be applied in that dynamic environment. These questions are essential to make DSSE deployable in practice[1].

4.1 System Overview

We first design the system that supports a dynamic (addition) SSE scheme to further investigate the capability of the leakage-abuse attacks in that. ShieldDB is a document-oriented database, where semi-structured records are modeled and stored as documents, and can be queried via keywords or associated attributes.

[1]This chapter is based on [1]

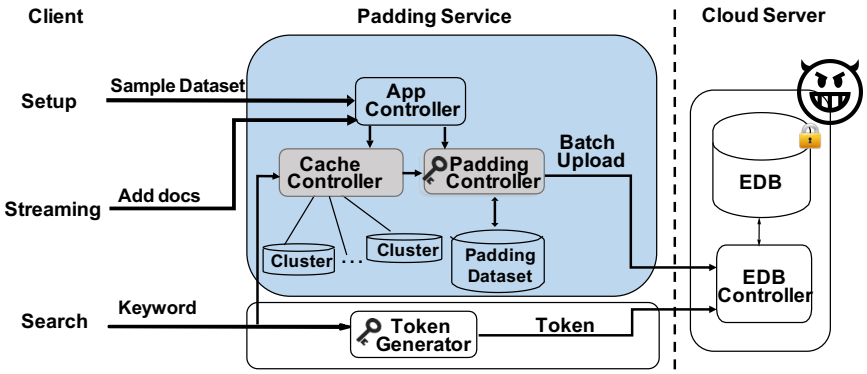


Figure 4.1: High-level design of ShieldDB

Algorithm 3 The setup protocol in ShieldDB

setup $(1^\lambda, \Delta_{stp})$

Client

1: Transfer dataset Δ_{stp} to P

Padding Service

1: $\{k_1, k_2\} \xleftarrow{\$} \{0,1\}^\lambda$
2: Initialise a map ST' and a tuple T
3: Run $\texttt{Setup}(\Delta_{stp})$ (see Section 4.3.1)

Server

1: Initialise an index map EDB

Participants and scenarios: As illustrated in Figure 4.1, ShieldDB consists of a query client C, a trusted padding service P and an untrusted storage server S. In our targeted scenario, new documents are continuously inserted to S, and required to be encrypted. Meanwhile, C expects S to retain search functionality over the encrypted documents. To enhance the security, P adapts padding countermeasures during encryption. In this paper, we consider an enterprise that utilises outsourced storage. P is deployed at the enterprise gateway and in the same network with C, and P encrypts and uploads the documents created by its employees, while C is deployed for employees to search the encrypted documents at S. Note that the deployment of P is flexible. It can be separated from or co-located with C.

Overview: ShieldDB supports three main operations, i.e., setup (see Algorithm 3), streaming (see Algorithm 4), and search (see Algorithm 5). Apart from the main functions, ShieldDB also supports optimisation features deletion and re-encryption, and flushing operations (Section 4.3.3).

During setup (see Algorithm 3), P receives a sample training dataset Δ_{stp} from C, and then it groups keywords into clusters $L = \{L_1, \cdots, L_m\}$. After that,

Algorithm 4 The streaming protocol in ShieldDB with *forward privacy*

streaming (in $= \{(\mathsf{doc}, id)\}$)

Client

Transfer in to P

Padding Service

Parse in to $M = \{(w, id)\}$
//cache and check padding constraints
$V \leftarrow$ `PaddingCheck`(M) (see Algorithm 6)
//if there is no real/bogus pairs returned by *Padding Controller*
if $V = \{\emptyset\}$ **then**
    return; //not sending to *Server*
else
    for each w **in** V **do**
        $k_e \xleftarrow{\$} \{0,1\}^\lambda; k_w \leftarrow F(k_1, w); k_{id} \leftarrow F(k_2, w)$
        //let b is the current batch
        if $ST'[w] \neq \perp$ **then**
            $(st_{w_{(b-1)}}, c_{w_{(b-1)}}) \leftarrow ST'[w]$
        else
            $st_{w_0} \xleftarrow{\$} \{0,1\}^\lambda$, $c_{w_0} \leftarrow 0$
        end if
        $st_{w_b} \leftarrow F(k_e, st_{w_{(b-1)}})$;
        $i \leftarrow 0$
        for each id *matches* w **do**
            $u \leftarrow H_1(F(st_{w_b}, i) \parallel k_w); v \leftarrow H_2(F(st_{w_b}, i) \parallel k_{id}) \oplus id$
            $T \leftarrow T \cup (u, v)$
            $i \leftarrow i + 1$;
        end for
        $c_{w_b} \leftarrow i$;
        $ST'[w] \leftarrow (st_{w_b}, c_{w_b})$
        $u_{w_b} \leftarrow H_1(F(st_{w_b}, c_{w_b}) \parallel k_w)$
        $v_{w_b} \leftarrow H_2(F(st_{w_b}, c_{w_b}) \parallel k_{id}) \oplus (k_e \parallel c_{w_{(b-1)}})$
        $T \leftarrow T \cup (u_{w_b}, v_{w_b})$
    end for
    Send T to Server
end if

Server

for each (u, v) **in** T **do**
    EDB$[u] = v$
end for

Algorithm 5 The search protocol in ShieldDB with *forward privacy*

search (w)

Client

Receive $ST'[w]$ from *Padding Service*
if $ST'[w] \neq \perp$ **then**
    $k_w \leftarrow F(k_1, w); k_{id} \leftarrow F(k_2, w)$
    $(st_{w_b}, c_{w_b}) \leftarrow ST'[w]$
    Send $(k_w, k_{id}, st_{w_b}, c_{w_b})$ to *Server*
else
    Search w in P, return R
end if

Server

$R \leftarrow \emptyset$, $st_i \leftarrow st_{w_b}$, $c_i \leftarrow c_{w_b}$
while $c_i \neq 0$ **do**
    for $j = 0$ to $(c_i - 1)$ **do**
        $u \leftarrow H_1(F(st_i, j) \parallel k_w)$
        $v \leftarrow \mathrm{EDB}[u]$
        $id \leftarrow v \oplus H_2(F(st_i, j) \parallel k_{id})$;
        $R \leftarrow R \cup (u, v)$
    end for
    $u_k \leftarrow H_1(F(st_i, c_i) \parallel k_w)$
    $v_k \leftarrow \mathrm{EDB}[u_k]$
    $(k_i \parallel c_{i-1}) \leftarrow v_k \oplus H_2(F(st_i, c_i) \parallel k_{id})$
    $st_{i-1} \leftarrow F^{-1}(k_i, st_i)$
    $st_i \leftarrow st_{i-1}, c_i \leftarrow c_{i-1}$;
end while
send R to *Client*

App Controller in P notifies L to the module *Cache Controller* to initialise a cache capacity L_i for each keyword cluster. *App Controller* also notifies L to the module *Padding Controller* to generate a padding dataset B.

During streaming (see Algorithm 4), P receives an input, $\mathsf{in} = \{(\mathsf{doc}, id)\}$ containing a collection of documents, each element is a document doc with identifier id, sent from C. Then, P parses them into a set of keyword and document identifier (w, id) pairs, i.e., index entries for search. Then, *Cache Controller* stores these pairs to the caches of the corresponding keyword clusters. Based on the targeted attack model, *Cache Controller* applies certain constraints in `PaddingCheck` to flush the cache (Algorithm 6). Once the constraints on a cluster are met, *Cache Controller* notifies the satisfied cluster to *Padding Controller* for padding. In particular, *Padding Controller* adds bogus (w, id) pairs extracted from the padding dataset to make the keywords in this cluster

have equal frequency. Then, P encrypts and inserts all those real and bogus index entries as a data collection in a batch to EDB with *forward privacy* support.

In Algorithm 4, we note that k_e is an ephemeral key generated for batch insertion. P maintains the master state $ST'[w] = (st_{w_b}, c_{w_b})$ for each keyword w, where st_{w_b} is the master key to derive entries for (w, id) pairs in the same latest batch b, and c_{w_b} presents the result length w (i.e., the number of real and bogus ids containing w) in that batch b. The result length of w in the previous batch $c_{w_{(b-1)}}$ is embedded in v_{w_b} (see streaming protocol line 26). In search, st_i and c_i present the state key and the result length of w in batch i. H_1 and H_2 are hash functions, and F is AES cipher.

During search (see Algorithm 5), for a given single query keyword w, C wants to retrieve documents matching that keyword from S and P. First, C retrieves the local results from *Cache Controller* in P, since some index entries might have not been sent to EDB yet. After that, C sends a query token generated from this keyword to S to retrieve the rest of the encrypted results. After decryption, C filters padding and combines the result set with the local one. For security, C will not generate query tokens against the data collection which is currently in streaming; this constraint enforces S to query only over data collections which are already inserted to EDB. Following the setting of SE [66, 69], search is performed over the encrypted index entries in EDB, and document identifiers are pseudo-random strings. In response to query, S will return the encrypted documents via recovered identifiers in the result set after search.

Apart from padding countermeasures, ShieldDB provides several other salient features. First, it realises forward privacy [78] (an advanced notion of SE) for the streaming operation. Our realisation is customised for efficient batch insertion and can prevent S from searching the data collection in streaming. Second, ShieldDB integrates the functionality of re-encryption. Within this operation, index entries in a targeted cluster are fetched back to P and the redundant padding is removed. At the same time, deletion can be triggered, where the deleted index entries issued and maintained at P are removed and will not be re-inserted. After that, real entries combing with new bogus entries are re-encrypted and inserted to EDB. Third, *Cache Controller* can issue a secure flushing operation before meeting the constraints for padding. This reduces the overhead of P while preserving the security of padding.

Remark: ShieldDB assigns P for key generation and management, and P issues the key for C to query. In addition, C also gets the latest state of the query keyword from P, and together with the key, to generate query tokens and send them to S. In our current implementation, P and C use the same key for index encryption, just as most SSE schemes do. This is practical because SSE index only stores pseudorandom identifies of documents, and documents can separately be encrypted via other encryption algorithms. Advanced key management schemes of SE [104, 121] can readily be adapted; yet, this is not relevant to our problem.

Like many other SE works [75, 82, 81, 2] that focus on search document index, we only present that streaming in ShieldDB updates real/bogus document identifiers via (w, id) pairs to the index map EDB. Real and padding documents containing these pairs can be uploaded separately by P to S via other encryption algorithms. In search, once the identifiers of real/bogus documents matching the query keyword are uncovered, S retrieves the corresponding documents and returns them to C. Note that, we omit presenting the physical document management in S in the rest of the paper since it does not affect the security of ShieldDB against the non-persistent and persistent adversaries as proposed in Section 4.2.

4.2 Attack Models and Assumptions

ShieldDB mainly considers a passive adversary who monitors the server S's memory access and the communication between the *Server* and other participants. Following the assumption of the count attack [74], the adversary has access to the background knowledge of the dataset and aims to exploit this information with the access pattern in search operations to recover query keywords. In this paper, we extend this attack model to the dynamic (streaming) setting.

Before elaborating the attack models, we define the streaming setting. In our system, streaming performs batch insertion on a collection of encrypted (w, id) pairs to S. Giving a number of continuous streaming operations, encrypted collections are added to a sequence over time. Accordingly, S orders the sequence of data collections by the timestamp. We define the gap between any two consecutive timestamps is a time interval t, and C is allowed to search at any time interval. Note that at a given t, S can only perform search operations against the collections that have been completely inserted to EDB.

In the dynamic setting, we observe two new attack models, which we refer to as non-persistent and persistent adversaries, respectively.

Non-persistent adversary: This adversary controls S within one single arbitrary time interval t_i, where i is a system parameter that monotonically increases and $i \geq 0$. During t_i, she observes query tokens that C issued to S, and the access patterns returned by S. She knows the accumulated (not separate) knowledge of the document sets inserted from t_0 to t_i.

Persistent adversary: This adversary controls S across multiple arbitrary time intervals, for example, from t_0 to t_i. She persistently observes query tokens and access patterns at those intervals, and knows the separate knowledge of the document sets inserted from t_0 to t_i.

For both attack models, S cannot obtain the query tokens against the encrypted data collections streamed in the current time interval. It is enforced by our streaming operation with forward privacy (see Section 4.3.3).

Strawman padding service against the adversaries: We note that a basic

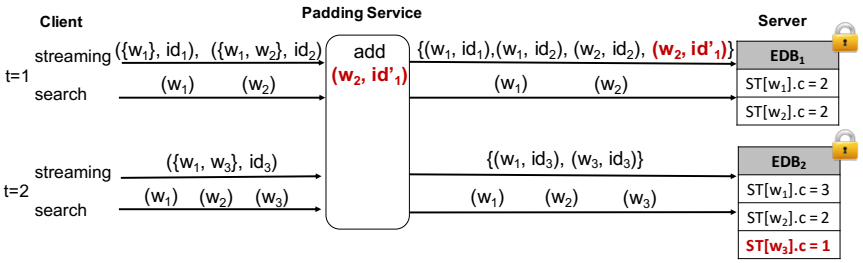


Figure 4.2: Strawman padding against non-persistent adversary

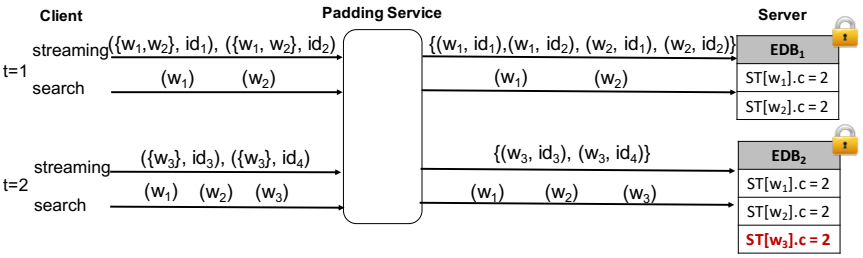


Figure 4.3: Strawman padding against persistent adversary

Padding Service P that only maintains one single cache for batch streaming cannot mitigate the proposed adversaries as presented in Figures 4.2 and 4.3.

In Figure 4.2, we show that the non-persistent adversary, capturing query tokens of w_1, w_2, and w_3, and their corresponding access patterns (i.e., result lengths $ST[w_1].c$, $ST[w_2].c$, and $ST[w_3].c$) at time $t = 2$, can uncover which tokens used for what keywords if she has the corresponding background knowledge of DB at time $t = 2$ (i.e., DB_2). The reason is due to the unique result lengths introduced in EDB at time $t = 2$ (i.e., EDB_2) when P adds bogus pairs to equalise the number of pairs for keywords sent to EDB during every streaming operation.

In Figure 4.3, we demonstrate that the persistent adversary can detect when new keywords are inserted in EDB. For example, she might know the states of the database DB at time $t = 1$ and $t = 2$ as her background knowledge, and the query results of w_3 in EDB_1 and EDB_2 are different. Then, she knows the occurrence of a new keyword w_3 is introduced in EDB_2 at $t = 2$. Then, she is able to identify the query token of w_3 during search at $t = 2$.

Real-world implication of the adversaries: We note that the proposed attack models are new for leakage-abuse attacks, which have not been investigated and formalised in any of the prior works [73, 74, 110]. We stress that non-persistent adversary could be any external attackers, e.g., hackers or organised cyber criminals. They might compromise the server at a certain time window. We also assume that this adversary could obtain a snapshot of the database via public channels, e.g., a prior data breach [53]. Because the database is changed dynamically, the snapshot might only reflect some historical state of

the database. On the contrary, the persistent adversary is more powerful and could be database administrators or insiders of an enterprise. They might have long term access to the server and could obtain multiple snapshots of the database via internal channels.

Other threats: Apart from the above adversaries, ShieldDB considers another specific rational adversary [76] who can inject documents to compromise query privacy. As mentioned, this threat can be mitigated via forward privacy SE. Note that ShieldDB currently does not address an active adversary who sabotages the search results.

4.3 Design of ShieldDB

In this section, we present the detailed design of ShieldDB in Setup, Stream, and Search. Then, we present some advanced features of ShieldDB to further improve the security and efficiency.

4.3.1 Setup

We consider $\Delta_{stp} = \{w_1, w_2, ..., w_l\}$ is the training dataset for the system. During $\texttt{Setup}(\Delta_{stp})$, P invokes *Cache Controller* to initialise the cache for batch insertion, and *Padding Controller* to generate bogus documents for padding.

To reduce padding overhead, ShieldDB implements cache management in a way that it groups keywords with similar frequencies together and performs padding at each individual keyword cluster. We denote $L = \{L_1, \cdots, L_m\}$ as caching clusters managed by *Cache Controller*, where m is the number of cache clusters. This approach is inspired from existing padding countermeasures in the static setting [74, 75]. The idea of doing this in a static database is intuitive; the variance between the result lengths of keywords with similar frequencies is small, which can minimize the number of bogus entries added to the database. We note that it is also reasonable in the dynamic setting, where the keyword frequencies in specific applications can be stable in the long run. If a keyword is popular, it is likely to appear frequently during streaming, and vice versa. Therefore, we assume that the existence of the training dataset, where the keyword frequencies are close to the real ones during streaming is reasonble (see Section 4.5.3 for that distribution evaluation). We further suggest alternative training data collection approaches in Section 4.5.4.

Given Δ_{stp}, *Cache Controller* partitions keywords based on their frequencies by using a heuristic algorithm. The objective function in Eq. 4.1, such that the clustering can be formed as $[(w_1, \ldots, w_i), (w_{i+1}, \ldots, w_j), \ldots, (w_k, \ldots, w_l)]$. We note that the minimum size of each group α is subjected to $\alpha \geq 2$. For security, the keyword frequency in each cluster after padding should be the same, i.e., the maximum one, and thus *Cache Controller* computes the padding overhead γ as

follows:

$$\gamma = \left(i * f_{w_i} - \sum_{t=1}^{i} f_{w_t} \right) + \left((j-i) * f_{w_j} - \sum_{t=i+1}^{j} f_{w_t} \right) + \\ \ldots + \left((l-k-1) * f_{w_l} - \sum_{t=k}^{l} f_{w_t} \right) \tag{4.1}$$

This algorithm iterates evaluating γ for every combination of the partition. We denote by m the number of clusters. After that, the *Cache Controller* allocates the capacity of the cache based on the aggregated keyword frequencies of each cluster, i.e., $|L| \sum_{t=1}^{i} f_{w_t}$, $|L| \sum_{t=i+1}^{j} f_{w_t}$, ..., $|L| \sum_{t=k}^{l} f_{w_t}$, where $|L|$ is the total capacity assigned for the local cache. We denote by $L_i.threshold()$ the function that outputs the caching capacity of cluster L_i.

After that, *Padding Controller* initialises a bogus dataset B with size $|B|$, where the number of bogus keyword/id pairs for each keyword w_i is determined via the frequency, i.e., $|B|(f_w - f_{w_i})$, where f_w is the maximum frequency in the cluster of w_i. The reason of doing so is that it still follows the assumption in cache allocation. If the keyword is less frequent in a cluster, it needs more bogus pairs to achieve the maximum result length after padding, comparing other keywords with higher frequency, and vice versa. Then the controller generates bogus index pairs. Once the bogus pairs for a certain keyword w_i is run out, the controller is invoked again to generate padding for it through the same way.

Remark: We assume that the distribution of the sample dataset is close to the one of the streaming data in a running period. We acknowledge that it is non-trivial to obtain an optimal padding overhead in the dynamic setting due to the variation of streaming documents in different time intervals. Nevertheless, if the distribution of the database varies during the runtime, the setup can be re-invoked. Namely, keyword clustering algorithm can be re-activated based on the up-to-date streaming data (e.g., in a sliding window), and the cache can be re-allocated. Additionally, our proposed re-encryption operation can further reduce the padding overhead (see Section 4.3.3) if the streaming distribution only differs on particular keyword clusters. We discuss the distribution difference detection in Sections 4.5.3 and 4.5.4. We also note that there are applications and scenarios where the distribution does not vary much, like IoT streaming data for environment sensors. In such applications, the range of numbers/indicators are already specified by the vendors.

4.3.2 Padding Strategies

During streaming, documents are continuously collected and parsed as $M = \{(w, id)\}$ in P. Then, P executes `PaddingCheck`(M) to cache and check padding

Algorithm 6 Padding strategies

function `PaddingCheck`()

Input: $M = \{(w, id)\}$: entries for streaming
$\{L_1, \cdots, L_m\}$: cache clusters, and B: bogus document set
ST: a map that tracks keyword states
mode: padding mode (*high* or *low*);

Output: V is a set of real and bogus entries

push entries in M to $\{L_1, \cdots, L_m\}$
$V \leftarrow \{\emptyset\}$
if padding against non-persistent adversary **then**
  for cluster $L_i \in \{L_1, \cdots, L_m\}$ **do**
    if $L_i.capacity() \geq L_i.threshold()$ **then**
      for $w \in L_i$ **do**
        if $ST[w].flag = false$ **then**
          skip padding for w when executing `PaddingByMode`()
        end if
      end for
      $M_i \leftarrow$ `PaddingByMode`$(L_i, ST, B, mode)$;
      add M_i to V;
    end if
  end for
else if padding against persistent adversary **then**
  for cluster $L_i \in \{L_1, \cdots, L_m\}$ **do**
    if L_i.firstBatch=true & $ST[w].flag = true$ for $\forall w \in L_i$ **then**
      $M_i \leftarrow$ `PaddingByMode`$(L_i, ST, B, mode)$
      add M_i to V
    else if $L_i.capacity() \geq L_i.threshold()$ **then**
      $M_i \leftarrow$ `PaddingByMode`$(L_i, ST, B, mode)$;
      add M_i to V;
    end if
  end for
end if
return V;

constraints. In details, these (w, id) pairs are cached at their corresponding clusters by *Cache Controller*. Once a cluster L_i is full, *Padding Controller* adapts the corresponding padding strategy to the targeted adversary, encrypts and inserts all real and bogus pairs to EDB in a batch manner. We elaborate on the padding strategies against the non-persistent and persistent adversaries, respectively. The details are given in Algorithms 6 and 7.

Padding strategy against the non-persistent adversary: Recall that this adversary controls S within a certain time interval t. From the high level point of view, an effective padding strategy should ensure that all keywords occurred in EDB at t do not have unique result lengths. There are two challenges to achieve this goal. First, t can be an arbitrary time interval. Therefore, the above guarantee needs to be held at any certain time interval. Second, not all the keywords in the keyword space would appear at each time interval. It is non-trivial to deal with this situation to preserve the security of padding.

To address the above challenges, ShieldDB programs *Padding Controller* to track the states of keywords over the time intervals from the beginning. Specifically, each keyword state $ST[w]$ includes two components, a flag $ST[w].flag$ that indicates whether the keyword has existed before in the streamed documents, and a counter $ST[w].c$ that presents the number of total real and bogus (w, id) pairs already uploaded in EDB of the keyword w. Note that $ST[w].flag = true$ is kept permanently once w has existed in the documents streamed to the server. *Padding Controller* only pads the keywords in a cluster L_i if the number of cached real (w, id) pairs of the cluster, denoted by $L_i.capacity()$, exceeds $L_i.threshold()$ defined in `setup` (see Algorithm 6 line 6).

Based on the states of keywords, *Padding Controller* performs the following actions. If the keyword has not existed yet, the controller will not pad it even its cluster is full (see Algorithm 6 line 8). The reason is that the adversary might also know the information of keyword existence. If C queries a keyword which does not exist, S should return an empty set. Otherwise, the adversary can identify the token of this keyword if padded. Accordingly, only when a keyword w appears at the first time (i.e., $ST[w].flag = true$), padding over this keyword will be invoked (see Algorithm 7 lines 8 and 17).

Once $ST[w].flag = true$, this keyword will always get padded later, as long as the cache of its cluster $L_i.capacity() \geq L_i.threshold()$, no matter it exists in a certain time interval or not. The padding ensures that all existing keywords in the cluster always have the same result length at any following time interval.

Padding strategy against the persistent adversary: Recall that this adversary can monitor the database continuously and obtain multiple references of the database across multiple time intervals. Likewise, the padding strategy against the persistent adversary should ensure that all keywords have no unique access pattern in all time intervals from the very beginning. However, directly using the strategy against the non-persistent adversary here does not address the leakage of keyword existence.

To address this issue, *Padding Controller* is programmed to enforce another necessary constraint to invoke padding. That is, all keywords in the cluster at the first batch have to exist before streaming. Formally, we let $L_i.firstBatch$ be the constraint that evaluates the existence of all predefined keywords in L_i. Then, $L_i.firstBatch = true$ implies $ST[w].flag = true$ for $\forall w \in L_i$. The

Algorithm 7 Padding modes

function `PaddingByMode`($L_i, ST, P, mode$)

Input: L_i: cluster for padding
ST: a map that tracks keyword states, and B: bogus document set
ST: a map that tracks keyword states
$mode$: padding mode ($high$ or low);

Output: M_i: a set of real and bogus entries

1: $M_i \leftarrow \emptyset$
2: $st_{max} \leftarrow \max\{ST[w].c\}$ for $\forall w \in L_i$
3: Let c_w is the length of the currently matching list of w cached in L_i
4: **if** $mode = high$ **then**
5: Let c_{max} is $\max\{c_w, \forall w \in L_i\}$
6: $C \leftarrow st_{max} + c_{max}$
7: **for** $w \in L_i$ with $ST[w].flag = true$ **do**
8: add $(C - c_w)$ bogus entries from $B[w]$ to M_i
9: add all c_w cached entries of w in L_i to M_i
10: $ST[w].c \leftarrow C$
11: **end for**
12: **else if** $mode$ is low **then**
13: Let c_{min} is $\min\{c_w > 0, \forall w \in L_i\}$
14: $C \leftarrow st_{max} + c_{min}$
15: **for** $w \in L_i$ with $ST[w].flag = true$ **do**
16: $m \leftarrow C - ST[w].c$;
17: **if** $m > c_w$ **then**
18: add $(m - c_w)$ bogus entries from $B[w]$ to M_i
19: add all c_w cached entries of w in L_i to M_i
20: **else**
21: //do not add bogus entries for w
22: put (m) cached entries of w in L_i to M_i
23: //the remaining $(c_w - m)$ entries of w are still cached in L_i
24: **end if**
25: $ST[w].c \leftarrow C$
26: **end for**
27: **end if**
28: return M_i;

constraint remains $false$ if there is $\exists w_j \in L_i$, $ST[w_j] = false$. As a trade-off, *Cache Controller* has to hold all the pairs in the cluster even the cache is full, if there are still keywords yet to appear (see Algorithm 6 line 18). For subsequent batches of the cluster, the padding constraint follows the same strategy for the non-persistent adversary (see Algorithm 6 line 21).

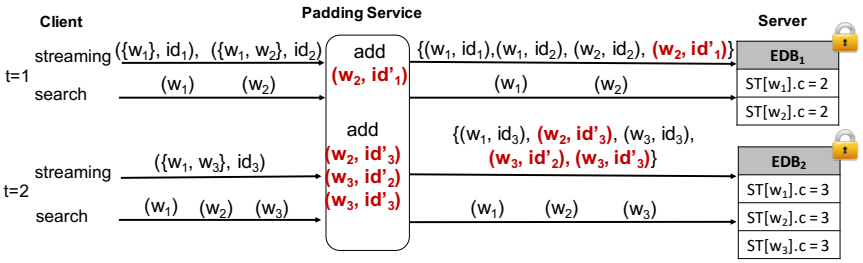


Figure 4.4: *High* mode padding against non-persistent adversary

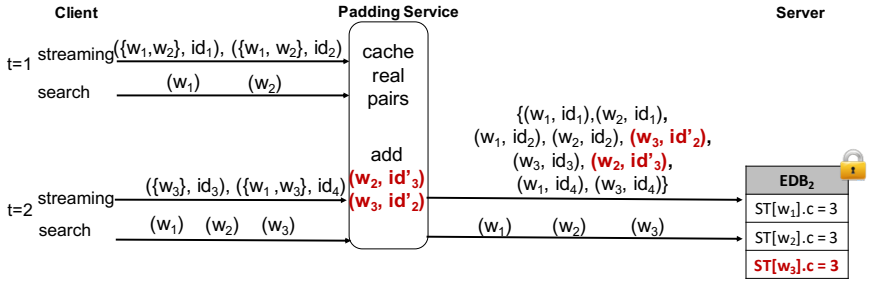


Figure 4.5: *High* mode padding against persistent adversary

Padding modes: ShieldDB implements two modes for padding, i.e., *high* and *low* modes. These two modes both are applicable to the above two padding strategies (see *mode* in Algorithm 6). The padding mechanism of these modes are described in Algorithm 7. In the *high* mode, once the constraint for the cache of a cluster is met, the keywords to be padded have the maximum result length of keywords in this cluster (see Algorithm 7 lines 9-11). Accordingly, the cache can be emptied since all entries are sent to *Padding Controller* for streaming. On the contrary, the *low* mode is invoked in a way that the keywords to be padded have the minimum result length of keywords in this cluster. Therefore, some entries of keywords might still be remained in the cache (see Algorithm 7 lines 24-25). Yet, this mode only introduces necessarily minimum padding for keywords which do not occur in random time intervals. The two modes have their own merits. The *high* mode consumes a larger amount of padding and execution time for padding and encryption, but it reduces the load of cache in P. In contrast, the *low* mode introduces relatively less padding overhead but heavier load of P.

Figure 4.4 demonstrates the padding strategy against the non-persistent adversary by using *high* padding mode. Given a cluster L_j containing three keywords $w_i, \forall i \in [1,3]$, P tracks their keyword state $ST[w_i].c$ and $ST[w_i].flag$ and applies padding when real cached pairs exceed the capacity of the cluster. Then, P adds bogus pairs to ensure they have the same result length in EDB at $t = 1$ and $t = 2$. Note that, there is no padding applied for w_3 at time $t = 1$ due to $ST[w_3].flag = false$ at that time. At $t = 2$, P pads all the keywords to the maximum result length among them in the cluster (i.e., $ST[w_i] = 3, \forall i \in [1,3]$).

Table 4.1: Time complexity of *Padding Service* and *Server*

Padding Service		*Server*
padding (streaming)	**encryption** (streaming)	**update/search**
$\|B\|(f_w - f_{w_i})$	$\mathcal{O}(n_r + n_b)$	$\mathcal{O}(n_r + n_b)$

In Figure 4.5, we demonstrate the padding strategy against the persistent adversary by using *high* padding mode for a cluster $L_j = \{w_i\}, \forall i \in [1, 3]$. P only performs padding and inserts encrypted real/bogus entries to EDB when all keywords in L_j have appeared (i.e., waiting for w_3 occurs at $t = 1$). Querying any w_i of the cluster prior this time does not make the client C send query tokens to S. The reason is because C only receives $ST'[w_i] = \perp$ sent by P to do search. We note that P only updates $ST'[w_i]$, (i.e., keyword state of w_i for encryption) when w_i in the cluster is ready for encryption after padding applied. We note that $L_j.firstBatch = true$ once all keywords in L_j have been appeared. Then, subsequent batches at $t > 1$ of L_j does not need to check keyword occurrence. Instead, it follows the padding strategy addressing the non-persistent adversary. The reason is because w_i is always padded in the following batches once it had appeared at the first time. The strategy aims to ensure there is no new keyword introduced at any random time interval to address the persistent adversary.

Complexity analysis of padding: We note that during setup, the *Padding Service* initialises a padding (bogus) dataset B with size of $|B|$. Then, the total number of bogus pairs used for the keyword w_i during the streaming phase is $|B|(f_w - f_{w_i})$, where f_w is the maximum frequency in the cluster of w_i found in the setup. The asymptotic complexity to encrypt real/bogus pairs of w_i in the streaming is $\mathcal{O}(n_r + n_b)$, where n_r and n_b present the total number of real and bogus pairs of w_i, respectively. In streaming (resp. search), upon receiving the update (resp. query) token(s) of w_i, the *Server* inserts (resp. retrieves) the entries (resp. search result) based on encrypted labels of w_i from the map EDB (i.e., $\mathcal{O}(n_r + n_b)$). The performance is summarised in Table 4.1.

Security guarantees: Our padding countermeasures ensure that no unique access pattern exists for keywords which have occurred in EDB. For the persistent adversary, the padding countermeasure also ensures that the keyword occurrence is hidden across multiple time intervals. Note that padding not only protects the result lengths of queries, but also introduces false counts in keyword co-occurrence matrix, which further increases the efforts of the count attack. Regarding the formal security definition, we follow a notion recently proposed by Bost et al. [75] for SSE schemes with padding countermeasures. This notion captures the background knowledge of the adversary and formalises the security strength of padding. That is, given any sequence of query tokens, it is efficient

to find another same-sized sequence of query tokens with identical leakage. We extend this notion to make the above condition hold in the dynamic setting in Section 4.4.

Remark: Our padding strategies are different from the approach proposed by Bost et al. [75], which merely groups keywords into clusters and pads them to the same result length for a static database. Directly adapting their approach for different batches of incoming documents will fail to address persistent or even non-persistent adversaries. The underlying reason is that the above approach treats each batch individually, while the states of database are accumulated. Effective padding strategies in the dynamic setting must consider the accumulated states of the database so that the adversaries can be addressed in arbitrary time intervals.

4.3.3 Optimisation Features

ShieldDB provides several other salient features to enhance its security, efficiency, and functionality.

Forward privacy: In streaming and search, ShieldDB realises the notion of forward privacy [78, 113] to protect the newly added documents and mitigate the injection attacks [76]. In particular, our system customises an efficient DSSE scheme with forward privacy [113] to our context of batch insertion. The detailed algorithm for encryption and search can be found in Algorithms 4 and 5 . Our forward-private scheme is built on symmetric-key based trapdoor permutation and is faster than the public-key based solution [78]. The ephemeral key k_e of permutation is embedded inside the index entry to recover the state $(st_{w_{(b-1)}}, c_{w_{(b-1)}})$ of the previous entries in batch $(b-1)$, lines 16-27 in streaming, and lines 17-21 in search protocol). To reduce the computation and storage overhead, we link a master state $ST'[w] = (st_{w_b}, c_{w_b})$ to a set of entries with the same keyword in the batch b (see streaming at lines 19-20).

Upon receiving $ST'[w]$ of the query keyword w sent from P, C generates a query token and sends to S. We note that $ST'[w]$ is different from the state $ST[w]$ used for padding in *Padding Controller*. The benefit of our forward-private design is that S can be enforced to perform search operations over the completed batches. The batches which are still transmitted on the fly cannot be queried without the latest keyword state $ST'[w]$ from C.

Re-encryption and deletion: ShieldDB also implements the re-encryption operation. This operation is periodically conducted over a certain keyword cluster. *Padding Service* P first fetches all entries in this cluster stored in EDB from S. After that, P removes all bogus entries and re-performs the padding over this cluster of keywords. All the real and bogus entires are then encrypted via a fresh key, and inserted back to EDB. The benefits of re-encryption are two-fold: (1) redundant bogus entries in this cluster can be eliminated; and (2)

the leakage function can be reset to protect the search and access patterns. During re-encryption, ShieldDB can also execute deletion. A list of deleted document ids is maintained at P, and the deleted entries are physically removed from the cluster before padding.

Cache flushing: During streaming, the keywords in some clusters might not show up frequently. Even the cache capacity of such clusters is set relatively small, the constraint might still not be triggered very often. To reduce the load of the cache at P and improve the streaming throughput, ShieldDB develops an operation called flushing to deal with the above "cold" clusters. In particular, *Cache Controller* monitors all the caches of clusters, and sets a time limit to trigger flushing. If a cluster is not full after a period of this time limit, all entries in this cluster will be sent to *Padding Controller*. Note that the padding strategies still need to be followed for security and the *high* mode of padding is applied to empty the cache.

4.4 Security of ShieldDB

4.4.1 Leakage Functions

ShieldDB implements a dynamic searchable symmetric encryption scheme (DSSE) $\Sigma = (\mathsf{Setup}, \mathsf{Streaming}, \mathsf{Search})$, consisting of three protocols between a padding service P, a storage server S, and an querying client C. A database $\mathrm{DB}_t = (w_i, id_i)_{i=1}^{|DB_t|}$ is defined as a tuple of keyword and document id pairs with $w_i \subseteq \{0,1\}^*$ and $id_i \in \{0,1\}^l$ at the time interval $t \geq 0$. We first formalise the DSSE-based leakage functions of ShieldDB as follows.

$\mathsf{Setup}(\mathrm{DB}_0)$ is a protocol that takes as input a database DB_0, and outputs a tuple of $(k_1, k_2, \{L_1, \ldots, L_m\}, st, B, \mathrm{EDB}_0)$, where k_1, k_2 are secret keys to encrypt keywords and document ids, a set $\{L_1, \ldots, L_m\}$ contains cache clusters, st maintains keyword states, and B is a bogus dataset to be used for padding, and EDB_0 is the encrypted database at $t = 0$.

$\mathsf{Streaming}(k_1, k_2, L_u, st, B, \{(w_i, id_i)\}; \mathrm{EDB}_{t-1}, \{(u_i, v_i)\})$ is a protocol between P with inputs k_1,k_2, and L_u $(1 \leq u \leq m)$ the cache cluster to be updated, the states st, the bogus dataset B, and the set of keyword and document id pairs $\{(w_i, id_i)\}$ to be streamed, and S with input EDB_{t-1} the encrypted database at time $t-1$ $(t \geq 1)$, and $\{(u_i, v_i)\}$ the set of encrypted keyword and document identifier pairs for batch insertion. Once P uploads $\{(u_i, v_i)\}$ to S, st and B gets updated, L_u is reset. At S, once EDB_{t-1} gets updated by $\{(u_i, v_i)\}$, it changes to EDB_t.

$\mathsf{Search}(k_1, k_2, q, st; \mathrm{EDB}_t)$ is a protocol between C with the keys k_1, k_2, the query q querying the matching documents of a single keyword w_i, and the state st, and S with EDB_t. Meanwhile, C queries P for retrieving cached documents of the query keyword.

The security of ShieldDB can be quantified via a leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Stream}, \mathcal{L}^{Srch})$. It defines the information exposed in Setup, Streaming, and Search, respectively. The function ensures that ShieldDB does not reveal any information beyond the one that can be inferred from $\mathcal{L}^{Stp}$, $\mathcal{L}^{Stream}$, and $\mathcal{L}^{Srch}$.

In Setup, $\mathcal{L}^{Stp} = |\text{EDB}_0|$ presenting the size of EDB_0, i.e., the number of encrypted keyword and document id pairs.

In Streaming, ShieldDB is *forward private* as presented in Streaming protocol. Hence $\mathcal{L}^{Stream}$ can be written as

$$\mathcal{L}^{Stream}(\{(w, id)\}) = \mathcal{L}'(\{id\})$$

where $\{(w, id)\}$ denotes a batch of keyword and id pairs w, and $\mathcal{L}'$ is a stateless function. Hence, $\mathcal{L}^{Stream}$ only reveals the number of pairs to be added to EDB. ShieldDB does not leak any information about the updated keywords. In particular, S cannot learn that the newly inserted documents match a keyword that being previously queried.

In Search, $\mathcal{L}^{Srch}$ reveals common leakage functions [66]: the *access pattern* ap and the *search pattern* sp as follows.

The ap reveals the encrypted matching document identifiers associated with search tokens. For instance, if an adversary controls EDB_t, she monitors the search query list $Q_t = \{q_1, \ldots, q_{n-1}\}$ by the time order. Then, $\mathsf{ap}(q_i)$ (with $1 \leq i \leq n-1$) for a query keyword w_i is presented as

$$\mathsf{ap}(q_i) = \text{EDB}(w_i) = \{(u_{w_i}, v_{w_i})\}$$

where u_{w_i} and v_{w_i} are an encrypted keyword and document id entry associated with w_i in EDB_t.

The sp leaks the repetition of search tokens sent by C to S, and hence, the repetition of queried keywords in those search tokens.

$$\mathsf{sp}(q_i) = \{\forall j \neq i, q_j \in Q_t, w_j = w_i\}$$

Next, we detail the leakage during the interaction between C and S over Q_t on a given DB_t. We call an instantiation of the interaction as a *history* $H_t = (\text{DB}_t, q_1, \ldots, q_{n-1})$. We note that the states of keywords in DB_t do not change during these queries. The leakage function of H_t is presented as

$$\mathcal{L}(H_t) = (|\text{EDB}(w_i)|, \ldots, |\text{EDB}(w_{n-1})|, \alpha(H_t), \sigma(H_t))$$

where $|\text{EDB}(w_i)|$ $(1 \leq i \leq n-1)$ is the number of matching documents associated with the keyword w_i mapping to the query q_i, $\alpha(H_t) = \{\mathsf{ap}(q_1), \ldots, \mathsf{ap}(q_{n-1})\}$ is the *access pattern* induced by Q_t, and $\sigma(H_t)$ is a symmetric binary matrix such that for $1 \leq i, j \leq n-1$, the element at i^{th} row and j^{th} column is 1 if $w_i = w_j$, and 0 otherwise.

4.4.2 Extended Constrained Security in ShieldDB

We note that the database knowledge of non-persistent and persistent adversaries falls outside the traditional SE formalisation [75]. The reason is because the notion is limited by the fact that knowing the DB, the query list is uniquely defined by the acceptable leakage of SE. Namely, there is already the uniqueness of a *history* given the knowledge of the adversary. Therefore, we want to define new constrained security that can formalise the adversary's knowledge in H_t. But, given the constraint, there are multiple *histories* at time t satisfying the leakage function (i.e., making H_t no longer unique). In this way, one needs to find two different lists of queries generating the exact same leakage with the same DB_t. As a starting point, we extend the Definition 3.1 in [75] to formalise H_t satisfying the constraint C iff $C(H_t) = \texttt{true}$ as in Definition. 8.

Definition 8. *A constraint $C = (C_0, C_1, \ldots, C_{n-1})$ over a database set $\mathcal{DB}_t$ and a query set $Q_t = \{q_1, \ldots, q_{n-1}\}$, is a sequence of algorithms such that, for $\mathrm{DB}_t \in \mathcal{DB}_t$, $C_0(\mathrm{DB}_t) = (flag_0, st_0)$, where $flag_0$ is* `true` *or* `false` *and st_0 captures C_0's state, and for $q \in Q_t$, $C_i(q, flag_{i-1}) = (flag_i), (i \geq 1)$. The constraint is consistent if $C_i(., \texttt{false}, .) = (\texttt{false}, .)$ (the constraint remains* `false` *if it once evaluates to* `false`*).*

For a history $H_t = (\mathrm{DB}_t, q_1, \ldots, q_{n-1})$, we note $C(H_t)$ the evaluation of

$$C(H_t) := C_{n-1}(q_{n-1}, C_{n-2}(\ldots, C_0(\mathrm{DB}_t))).$$

If $C(H_t) = \texttt{true}$, we say that H_t satisfies C. A constraint C is valid if there exists two different efficiently constructable histories H_t and H'_t satisfying C.

After defining the knowledge in H_t known by the adversary, we also formalise some elements (i.e., queries) in H_t that are unknown to the adversary. Namely, they are left *free* from the constraint C. We note that Bost et al. [75] already defined *free* components in static database setting (i.e., not time interval t) for constraint security. Therefore, we extend the Definition 3.2 in [75] to formalise *free* component in C regarding H_t in below Def. 9.

Definition 9. (Free history component) *We say that C lets the i-th query* free *if for history $H_t = (\mathrm{DB}_t, q_1, \ldots, q_{n-1})$ satisfying C, for every search (resp. update) query q if q_i is a search (resp. update) query, $H'_t = (\mathrm{DB}'_t, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_{n-1})$ also satisfies C, where $\mathrm{DB}'_t \in \mathcal{DB}_t$.*

The idea behind of letting *i-th* query *free* is that there exists some other queries in the history H'_t such that H'_t still satisfies C (and both $\mathcal{L}(H_t) = \mathcal{L}(H'_t)$) without modifying the leakage $\mathcal{L}(H'_t)$.

Now, we also define the *acceptable* constraint notion, so that, given a constraint C, and a leakage function $\mathcal{L}$, for every history H_t, we are able to find another *history* satisfying C with the same leakage.

Definition 10. *A constraint C is $\mathcal{L}$-acceptable for some leakage $\mathcal{L}$ if, for every efficiently computable history H_t satisfying C, there exists an efficiently computable $H'_t \neq H_t$ satisfying C, for $H'_t = (\mathrm{DB}'_t, q_1, \ldots, q_{n-1})$, such that $\mathcal{L}(H_t) = \mathcal{L}(H'_t)$.*

A set of constraints $\mathfrak{C}$ is said to be $\mathcal{L}$-acceptable if all its elements are $\mathcal{L}$-acceptable.

Now, after giving background definitions, we start to investigate the query at time t. We recall that the *Client* only triggers the search on completely outsourced data in EDB_t, where $(t > 0)$ is a random interval upon receiving search query tokens. Therefore, we consider that the leakage function only depends on the query itself, and on the state of DB_t: $\mathcal{L}(q)$ can be presented as a stateless function of $f_{\mathcal{L}}(q, \mathrm{DB}_t)$. We make an observation on EDB_t that: let C be a constraint, $H_t = (\mathrm{DB}_t, q_1, \ldots, q_{n-1})$ an *history* satisfying C, and q, q' be two queries such that $\widetilde{H}_t = H_t||q = (\mathrm{DB}_t, q_1, \ldots, q_{n-1}, q)$ and $\widetilde{H}'_t = H_t||q' = (\mathrm{DB}_t, q_1, \ldots, q_{n-1}, q')$. Then, if $f_{\mathcal{L}}(q, \mathrm{DB}_t) = f_{\mathcal{L}}(q', \mathrm{DB}_t)$, then both $\widetilde{H}_t$ and $\widetilde{H}'_t$ with the same leakage satisfying C. This observation can be iterated to create multiple (i.e., more than 2) *histories* using the same DB_t and they are both satisfying C with the same leakage. Therefore, we can define a clustering $\Gamma_t = \{G_1, \ldots, G_m\}$ of queries induced by the leakage $\mathcal{L}$ after history H_t is a partition of a query set Q_t, for which, in every cluster, queries share the same leakage after running the history H_t as below.

$$\bigcup_{i=1}^{m} G_i = Q_t$$

$$\forall i \neq j \quad G_i \cap G_j = \emptyset$$

$$\text{and } \forall q, q' \in G_i, \mathcal{L}(q, H_t) = \mathcal{L}(q', H_t)$$

where $\mathcal{L}(H_t, q)$ is the output of $\mathcal{L}(q)$ after having been run on each element of H_t. Note that, we omit the subscript t in Γ_t in the clear context of H_t; otherwise, we state it separately. We denote $\Gamma_{\mathcal{L}}(H_t)$ the clustering induced by $\mathcal{L}$ after H_t. We can see that it is impossible to merge different clusters in $\Gamma_{\mathcal{L}}(H_t)$ with the same leakage. Therefore, formally, for $\Gamma_{\mathcal{L}}(H_t) = \{G_1, \ldots, G_m\}$, where m is the total number of clusters, we have:

$$\forall i \neq j, \forall q \in G_i, \forall q' \in G_j, \mathcal{L}(H_t, q) \neq \mathcal{L}(H_t, q')$$

We present $\Gamma_{\mathcal{L},C}(H_t)$ the $\mathcal{L}$-induced clustering applied on history H_t satisfying C such that a subset of queries Q_t in queries q gives $C(H_t||q) = \texttt{true}$. We can see that, in the singular query q earlier, $\mathcal{L}(q)$ only depends on q and DB_t. Therefore, more generally, when q is a set of queries with $C(H_t||q) = \texttt{true}$, $\Gamma_{\mathcal{L},C}(H_t)$ only depends on DB_t. Indeed, the clustering $\Gamma_{\mathcal{L},C}(H_t)$ acquires at least two elements in every cluster. Otherwise, an *history* H_t can be constructed without any different *history* H'_t. Namely, we need at least $|G_i| > 2$ for $\forall i \in [1, m]$, to make sure that there are at least 2 constrained histories can be found. Therefore, we can extend Def. 10 to have an acceptable constraint C with α histories, where $|G_i| > \alpha$. We

note that the notation α here is inline with the clustering algorithm in setup in ShieldDB (Section 4.1).

Definition 11. (Extended acceptable constraint) *A constraint C is $(\mathcal{L}, \alpha)$-acceptable for some leakage $\mathcal{L}$ and integer $\alpha > 1$ if, for every efficiently computable history H_t^0 satisfying C (i.e., $C(H_t^0) = \texttt{true}$), there exists $(\alpha - 1)$ efficiently computable $\{H_t^i\}_{1 \leq i \leq \alpha - 1}$ such that $H_t^i \neq H_t^j$ for $i \neq j$, that are all satisfying C, and $\mathcal{L}(H_t^0) = \cdots = \mathcal{L}(H^{\alpha-1})$.*

Now, we can see that when $|G_i| \geq \alpha$, for $\forall i \in [1, m]$, i.e., strictly more than one element in each cluster of $\Gamma_{\mathcal{L},C}(H_t)$, C is $(\mathcal{L}, \alpha)$-acceptable, as formalised in below Proposition 1.

Proposition 1. *Let C be a constraint, and $\mathcal{L}$ a leakage function. If for every history H_t satisfying C, the clustering $\Gamma_{\mathcal{L},C}(H_t) = \{G_1, \ldots, G_m\}$ is such that $|G_i| \geq \alpha$ for all i, C is $(\mathcal{L}, \alpha) - acceptable$.*

4.4.3 Security IND Game against Non-persistent adversary

Prior knowledge of the database: Considering the adversary's knowledge of the database is DB_t when she captures $\mathcal{L}(H_t)$, we use the predicate C^{DB_t} to formalise this knowledge, by adapting the notion of server's knowledge in [75]. Formally, we have $C^{\mathrm{DB}_t}(H_t) = \texttt{true}$ if the database of the input history is DB_t. As used in Definition 10, C^{DB_t} ensures that all challenge histories' database is the same, i.e., DB_t. That also leave all queries in DB_t are left *free*, as defined in Definition 9. More generally, we can model the fact that the adversary know the database by considering the constraint set $\mathfrak{C}^{\mathcal{DB}_t} = \{C^{\mathrm{DB}_t}, \mathrm{DB}_t \in \mathcal{DB}_t\}$.

Now, we recall that the non-persistent adversary only captures an interval t, and search only triggers on encrypted entries inserted in EDB_t. Therefore, we consider the scheme $\Sigma_{NP} = (\mathsf{Setup}, \mathsf{Search})$ at time t for the non-persistent adversary. We start adding a padding mechanism presented in Algorithm 1 (i.e., Padding Strategies) to Σ such that, for every keyword in DB_t, there are at least different $(\alpha - 1)$ keywords with the same number of matching documents. Then, with the knowledge of DB_t, the leakage function of Σ_{NP} is formally defined as $\mathcal{L}_{NP} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{\alpha-pad})$, where $\mathcal{L}^{Stp}$ and $\mathcal{L}^{Srch}$ reveals the leakage in Setup at at t and Search against EDB_t, and the new leakage $\mathcal{L}^{\alpha-pad}$ reveals the minimum size of clusters induced by $\mathcal{L}^{\alpha-pad}$.

By using Proposition 1, we can show that $\mathfrak{C}^{\mathcal{DB}_t}$ is an $(\mathcal{L}_{NP}, \alpha)$-acceptable set of constraints, where α is the minimum cluster size (over all constructable databases). The reason is that, since constraints in $\mathfrak{C}^{\mathcal{DB}_t}$ leave all queries *free* for every history $H_t = (\mathrm{DB}_t, q_1, \ldots, q_{n-1})$, we can generate a different history H_t' with the same leakage by choosing another query $q \neq q_1$ that are both matching the same number of documents, and changing all queries $q_i = q_1$ in H_t to q. Also,

if there is queries $q_j = q$ in H_t, we can switch queries in q_j to q_1. This can give us a history $H'_t \neq H_t$ with the same leakage of H_t. We note that there are at least α choices of q to create $\Gamma_{\mathcal{L}_{NP}}(H_t)$ we can derive $\mathfrak{C}^{\mathcal{DB}_t}(\mathcal{L}_{NP}, \alpha)$-acceptable.

Now, we are ready to define the notion of constrained adaptive indistinguishability for Σ_{NP} given $\mathfrak{C}^{\mathcal{DB}_t}$ and the leakage $(\mathcal{L}_{NP})$.

Definition 12. *Let* $\Sigma_{NP} = (\mathsf{Setup}, \mathsf{Search})$ *be the SE scheme of ShieldDB,* λ *be the security parameter, and* $\mathcal{A}$ *be a non-persistent adversary. Let* $\mathfrak{C}^{\mathcal{DB}_t}$ *be a set of* $(\mathcal{L}_{NP}, \alpha)$*-acceptable constraints. Let* $\mathrm{Ind}_{SE,\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}$ *be the following game:*

$\mathrm{Ind}_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}(\lambda)$ *Game:*

$b \xleftarrow{\$} \{0, \ldots, \alpha - 1\}$
$(C_0^{\mathrm{DB}_t}, \mathrm{DB}_t^0, \ldots, \mathrm{DB}_t^{\alpha-1}) \leftarrow \mathcal{A}(1^\lambda)$
$(K, \mathrm{EDB}_t^b) \leftarrow \mathsf{Setup}(\mathrm{DB}_t^b)$
$(C_1^{\mathrm{DB}_t}, q_1^0, \ldots, q_1^{\alpha-1}) \leftarrow \mathcal{A}(\mathrm{EDB}_t^b)$
$\tau_1^b \leftarrow \mathsf{Search}(q_1^b)$
for $i = 2$ ***to*** n ***do***
  $(C_i^{\mathrm{DB}_t}, q_i^0, \ldots, q_i^{\alpha-1}) \leftarrow \mathcal{A}(q_{i-1}^b)$
  $\tau_i^b \leftarrow \mathsf{Search}(q_i^b)$
end for
$b' \leftarrow \mathcal{A}(\tau_n^b)$
if $b = b'$ ***return*** *1,* ***otherwise return*** *0*

where $\tau_i^b \leftarrow \mathsf{Search}(q_i^b)$ *presents the transcript of the query* q_i^b*, and with the restriction that, for all the* $H_t^i = (\mathrm{DB}_t^i, q_i^0, \ldots, q_i^{n-1})$*,*

- $C^{\mathrm{DB}_t} \in \mathfrak{C}^{\mathcal{DB}_t}$*, and* $\forall 0 \leq i \leq (\alpha - 1), C^{\mathrm{DB}_t}(H_t^i) = \mathtt{true}$
- $\mathcal{L}(H_t^0) = \cdots = \mathcal{L}(H_t^{\alpha-1})$

We say that Σ *is* $(\mathcal{L}_{NP}, \mathfrak{C}^{\mathcal{DB}_t}, \alpha)$*-constrained-adaptively-indistinguishable if for all probabilistic polynomial time adversary* $\mathcal{A}$*,*

$$\begin{aligned} &\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}(\lambda) = \\ &\Big|\mathbb{P}[\mathrm{Ind}_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}(\lambda) = 1] - \frac{1}{\alpha}\Big| \leq negl(\lambda). \end{aligned} \tag{4.2}$$

We can see that Σ_{NP} offers at least $log(\alpha)$ bits of security. Given $\mathfrak{C}^{\mathcal{DB}_t}$ $(\mathcal{L}_{NP}, \alpha)$-acceptable, we can analysing the transcripts τ_i^b under the choice of α. First, we make an observation on the keyword choice in $(\mathrm{DB}_t^0, \ldots, \mathrm{DB}_t^{\alpha-1})$ as follows. We denote by $\Delta_t^i = \{w_1^i, \ldots, w_{n-1}^i\}$ the keyword space of DB_t^i, where $i \in \{0, \alpha - 1\}$. Then, $C^{\mathrm{DB}_t}(H_t^i) = \mathtt{true}$ and all $\mathcal{L}(H_t^0) = \cdots = \mathcal{L}(H_t^{\alpha-1})$ imply $\Delta_t^0 = \cdots = \Delta_t^{\alpha-1}$. Let $f(w)$ be a function returning the frequency of the keyword w, we can see that, for all $w_j^0 \in \Delta_t^0$, where $j \in |\Delta_t^0|$, there are at least one another w_j^i in Δ_t^i (i.e., $\forall i \neq 0$) such that $f_{w_i^0} = f_{w_i^i}$. This turns out that the Setup needs to groups at least α keywords and pad them to be the same length such that, for

a given q_i^b in Search, under the chosen b, the transcript τ_i^b can be hardened by at least $(\alpha - 1)$ choices.

Now, we adapt the Theorem 2 in [75] to prove the extended constrained indistinguishability (i.e., Definition 12) by using regular leakage indistinguishability and extended acceptability of constraint set $\mathfrak{C}^{\mathcal{DB}_t}$ as follows.

Theorem 1. *Let Σ_{NP} = (Setup,Search) be our SE scheme, and $\mathfrak{C}^{\mathcal{DB}_t}$ a set of knowledge constraints. If Σ_{NP} is $\mathcal{L}_{NP}$-constrained-adaptively-indistinguishable secure, and $\mathfrak{C}^{\mathcal{DB}_t}$ is $(\mathcal{L}_{NP}, \alpha)$-acceptable, then Σ_{NP} is $(\mathcal{L}_{NP}$,$\mathfrak{C}^{\mathcal{DB}_t}, \alpha)$-constrained-adaptively-indistinguishability secure.*

Proof. Let $\mathcal{A}$ be an adversary in the $\mathrm{Ind}_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}$ game. We construct an adversary $\mathcal{B}$ against the game. $\mathcal{B}$ first randomly picks two integer $\alpha_0, \alpha_1 \in \{0, \alpha - 1\}$. Then, $\mathcal{B}$ starts $\mathcal{A}$ and receives α databases $(\mathrm{DB}_t^0, \ldots, \mathrm{DB}_t^{\alpha-1})$. Upon giving the pair $(\mathrm{DB}_t^{\alpha_0}, \mathrm{DB}_t^{\alpha_1})$ to the challenger, where the challenger holds a random secret bit b, $\mathcal{B}$ receives the challenge encrypted database EDB_t^* which she forwards to $\mathcal{A}$. Then, $\mathcal{A}$ repeatedly outputs α queries $(q_i^0, \ldots, q_i^{\alpha-1})$ and gives to $\mathcal{B}$. To respond, $\mathcal{B}$ outputs $(q_i^{\alpha_0}, q_i^{\alpha_1})$ to the game, and receives back the transcript τ_i^* and forwards it to $\mathcal{A}$. Then, $\mathcal{A}$ outputs the integer α'. If $\alpha' = \alpha_0$, $\mathcal{B}$ outputs $b' = 0$, else if $\alpha' = \alpha_1$, $\mathcal{B}$ outputs $b' = 1$, and otherwise outputs the probability $1/2$ for the output 0 and the probability $1/2$ for the output 1.

We first make an observation: for the pair $(H_t^{\alpha_0}, H_t^{\alpha_1})$, the views of the adversary $\mathcal{B}$ are indistinguishable due to $\mathcal{L}_{NP}(H_t^{\alpha_0}) = \mathcal{L}_{NP}(H_t^{\alpha_1})$, presenting both satisfying $\mathfrak{C}^{\mathcal{DB}_t}$. Then we can formalise $\mathcal{B}$ as follows:

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t}}(\lambda) = \left|\mathbb{P}[b = b'] - \frac{1}{2}\right| \leq negl(\lambda) \tag{4.3}$$

Now, we evaluate $\mathbb{P}[b = b']$ as follows.

$$\begin{aligned}
\mathbb{P}[b = b'] = \\
&\mathbb{P}[b = b' | \alpha' \in \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] \\
&+ \mathbb{P}[b = b' | \alpha' \notin \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \notin \{\alpha_0, \alpha_1\}] \\
&= \mathbb{P}[b = b' \cap \alpha' \in \{\alpha_0, \alpha_1\}] \\
&+ \mathbb{P}[b = b' | \alpha' \notin \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \notin \{\alpha_0, \alpha_1\}] \\
&= \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha} \text{ game}] \\
&+ \frac{1}{2}\left(1 - \mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}]\right)
\end{aligned} \tag{4.4}$$

Now, we evaluate $\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}]$ as follows.

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \mathbb{P}[\alpha' = \alpha_0] + \mathbb{P}[\alpha' = \alpha_1]$$

Since we have

$$\mathbb{P}[\alpha' = \alpha_0] + \mathbb{P}[\alpha' = \alpha_1] = \mathbb{P}[\alpha' = \alpha_b | b = 0] + \mathbb{P}[\alpha' = \alpha_b | b = 1]$$

then,

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \frac{1}{2}\left(\mathbb{P}[\alpha' = \alpha_b | b = 0] + \mathbb{P}[\alpha' = \alpha_0]\right) + \frac{1}{2}\left(\mathbb{P}[\alpha' = \alpha_b | b = 1] + \mathbb{P}[\alpha' = \alpha_1]\right)$$

We note that $\mathbb{P}[\alpha' = \alpha_b]$ is the probability $\mathcal{A}$ wins the 1-out-of-α indistinguishability game, and α_0 and α_1 are uniformly selected from $\{0, \alpha - 1\}$, then we have

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha} \text{ game}] + \frac{1}{\alpha} \tag{4.5}$$

Applying Eq. 4.5 to Eq. 4.4, we have

$$\mathbb{P}[b = b'] = \frac{1}{2} \cdot \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha} \text{ game}] + \frac{1}{2} - \frac{1}{2\alpha}$$

Then, from Equation 4.3, we can derive

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t}}(\lambda) = \frac{1}{2}\left(\mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha} \text{ game}] - \frac{1}{\alpha}\right) \tag{4.6}$$

Applying Equation 4.6 to Equation 4.2, finally, we can have

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t}}(\lambda) = \frac{1}{2}\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{A},\mathcal{L}_{NP},\mathfrak{C}^{\mathcal{DB}_t},\alpha}(\lambda) \tag{4.7}$$

□

4.4.4 Security IND Game against Persistent adversary

Prior knowledge of the databases We start to generalise the knowledge of the non-persistent adversary over the time to be the persistent adversary's knowledge of databases. Namely, we denote by $(\mathrm{DB}_{t=0}, \ldots, \mathrm{DB}_{t=T})$ such knowledge,

where T denotes the streaming period. We also make use of the constraint set $\mathfrak{C}^{\mathcal{DB}_t} = \{C^{\mathrm{DB}_t}, \mathrm{DB}_t \in \mathcal{DB}_t\}$, defined in Section 4.4.3, to formulate $\mathbf{C}^{[1,\ldots,T]} = \{\mathfrak{C}^{\mathcal{DB}_0}, \ldots, \mathfrak{C}^{\mathcal{DB}_T}\}$ the generalisation of constraint sets over the period such that we know every $\mathfrak{C}^{\mathcal{DB}_t}(\mathcal{L}_{NP}, \alpha)$-acceptable, $\forall t \in [0, T]$.

Let δ be a stateless function that outputs the keyword set difference in a pairwise different inputs. Then, $\delta(\mathrm{DB}_t, \mathrm{DB}_{t'}) = W^{t,t'}$, where $W^{t,t'} = \{w_i | w_i \in \mathrm{DB}_t, w_i \notin \mathrm{DB}_{t'}\}$. We consider the leakage function only depends on the query itself (*i.e.*, $q(.)$) and on the state of database at the querying time: $\mathcal{L}_{NP}(q)_t$ and $\mathcal{L}_{NP}(q)_{t'}$ be presented as stateless functions of $f_{\mathcal{L}_{NP}}(q, \mathrm{DB}_t)$, and $f_{\mathcal{L}_{NP}}(q, \mathrm{DB}_{t'})$, respectively.

Let $Q = \{q_1, \ldots, q_{n-1}\}$ be a query set, and C^{DB_t} and $C^{\mathrm{DB}_{t'}}$ be constraints applied on the $H_t = (\mathrm{DB}_t, Q)$ and $H_{t'} = (\mathrm{DB}_{t'}, Q)$, respectively. From Proposition 1, we denote by $\Gamma_{\mathcal{L}_{NP}, C^{\mathrm{DB}_t}}(H_t) = \{G_{1,t}, \ldots, G_{m,t}\}$ the clustering $\Gamma_t = \{G_{1,t}, \ldots, G_{m,t}\}$ of queries induced by the leakage $\mathcal{L}_{NP}$ after history H_t is a partition of Q, for which, in every cluster, queries share the same leakage after running the history H_t. Similarly, we also derive the clustering $\Gamma_{t'} = \{G_{1,t'}, \ldots, G_{m',t'}\}$ of queries Q_t induced by the leakage $\mathcal{L}_{NP}$ after history H'_t. We make an observation on EDB_t and $\mathrm{EDB}_{t'}$ that: let $q(w)$ (resp. $q(w')$) be the query of w (resp. w'), where $w \notin W^{t,t'}, w' \in W^{t,t'}$, and $q(w), q(w') \in G_{v,t}$, $(\exists v \in [1, m])$, such that:

$$(\widetilde{H}_t^w = H_t || q(w), \widetilde{H}_t^{w'} = H_t || q(w'))$$

and

$$(\widetilde{H}_{t'}^w = H_{t'} || q(w), \widetilde{H}_{t'}^{w'} = H_{t'} || q(w'))$$

We can see that $f_{\mathcal{L}_{NP}}(q(w), \mathrm{DB}_t) = f_{\mathcal{L}_{NP}}(q(w'), \mathrm{DB}_t)$, but $f_{\mathcal{L}_{NP}}(q(w), \mathrm{DB}_{t'}) \neq f_{\mathcal{L}_{NP}}(q(w'), \mathrm{DB}_{t'})$ due to $w \in \mathrm{DB}_{t'}$ while $w' \notin \mathrm{DB}_{t'}$, causing $\mathrm{EDB}_{t'}(w) \neq \mathrm{EDB}_{t'}(w')$. We can see that: if both $q(w), q(w') \in G_{v,t'}$, then $f_{\mathcal{L}_{NP}}(q(w), \mathrm{DB}_{t'}) = f_{\mathcal{L}_{NP}}(q(w'), \mathrm{DB}_{t'})$. This observation can be iterated for all other pairwise different queries in G_{v_t} and $G_{v_{t'}}$. More generally, we need to have $|G_{v,t}| = |G_{v,t'}|$, $\forall t, t \in [0, T]$ such that $\forall q(w), q(w') \in G_{v,t}$, they always have the same leakage at different $t' \in [1, T]$. Formally, given m the number of clusters, we define

$$\forall G_{v,t}, G_{v,t'} \neq \emptyset, F_{t,t',v}(G_{v,t}, G_{v,t'}) = (|G_{v,t}| \stackrel{?}{=} |G_{v,t'}|)$$

Now, we also use *constrained* security to formalise that leakage over EDB_t and EDB_{t-1}, $\forall t \geq 1$ as in.

Definition 13. *A constraint* $\mathbb{F}^t = (F_{t,t-1,1}, \ldots, F_{t,t-1,m})$ *over two clusters* $\Gamma_t = \{G_{1,t}, \ldots, G_{m,t}\}$ *and* $\Gamma'_t = \{G_{1,t'}, \ldots, G_{m,t'}\}$ *induced by the leakage* $\mathcal{L}_{NP}$ *after histories* H_t *(resp.* H_{t-1}*) over* Q_t *(resp.* Q_{t-1}*), is a sequence of algorithms such that* $F(t, t-1, i) = flag_i$*, where* $flag_i$ *is* `true` *or* `false`*. The constraint is consistent if* $(., \texttt{false}, .) = (\texttt{false}, .)$*, then* $\mathbb{F}^t = \texttt{false}$ *(the constraint remains* `false` *if it once evaluates to* `false`*).*

Let a constraint set $\mathcal{F} = (\mathbb{F}^1, \ldots, \mathbb{F}^T)$ is a sequence of algorithms evaluated at every $t \in [1, T]$. The set is consistent if $(., \texttt{false}, .) = (\texttt{false}, .)$, then $\mathcal{F} = \texttt{false}$ (the constraint remains `false` *if it once evaluates to* `false`*). In a short form, we write $\mathcal{F}^T$ to present the condition $\mathcal{F} = \texttt{true}$.*

The security of a scheme $\Sigma_P = (\mathsf{Setup}, \mathsf{Stream}, \mathsf{Search})$ against the persistent adversary over a streaming period T. We start adding a padding mechanism against the persistent adversary in Algorithm 1 (i.e., Padding Strategies) to Σ_P such that, $\forall G_{i,t}$ in $\Gamma_t = \{G_{1,t}, \ldots, G_{m,t}\}$ induced by Search_t (i.e., searching at time t) against EDB_t, $G_{i,t}$ always has the same size $|G_{i,t}| = |G_{i,t'}|$, where $G_{i,t'}$ in $\Gamma_{t'} = \{G_{1,t'}, \ldots, G_{m,t'}\}$, $\forall t' \neq t$. Let $\mathcal{L}^{Stream}_{[1,\ldots,T]} = \{L^{Stream}_1, \ldots, L^{Stream}_T\}$, and $\mathcal{L}^{Search}_{[1,\ldots,T]} = \{\mathcal{L}^{Srch}_{NP,i}\}$, for $i \in [1, \ldots, T]$, where $\mathcal{L}^{Srch}_{NP,i} = (\mathcal{L}^{Srch}_i, \mathcal{L}^{\alpha-pad}_i)$ is the search leakage at time i, where $\mathcal{L}^{\alpha-pad}_i$) reveals the sizes of clusters induced by $\mathcal{L}^{\alpha-pad}_i$. Then, the leakage function of Σ_P can be quantified via the leakage function $\mathcal{L}_P = (\mathcal{L}^{Stp}, \mathcal{L}^{Stream}_{[1,\ldots,T]}, \mathcal{L}^{Search}_{[1,\ldots,T]})$.

By using Definition 13, we can show that $\mathbf{C}^{[1,\ldots,T]} = \{\mathfrak{C}^{\mathcal{DB}_0}, \ldots, \mathfrak{C}^{\mathcal{DB}_T}\}$ is an $(\mathcal{L}_P, \alpha, \mathcal{F})$-acceptable set of constraint sets, where α denotes the minimum cluster size (over all constructable databases) and $\mathcal{F}^T$ denotes the condition $\mathcal{F} =$ `true`. The reason is that, for every time $t \in [0, T]$, $\mathfrak{C}^{\mathcal{DB}_t}$ is $(\mathcal{L}_{NP}, \alpha)$-acceptable, and $\forall G_{i,t}$ in $\Gamma_t = \{G_{1,t}, \ldots, G_{m,t}\}$ induced by Search_t (i.e., searching at time t) against EDB_t, $G_{i,t}$ always has the same size $|G_{i,t}| = |G_{i,t'}|$, where $G_{i,t'}$ in $\Gamma_{t'} = \{G_{1,t'}, \ldots, G_{m,t'}\}$, $\forall t' \neq t$

Now, we are ready to define the notion of constraned adaptive indistinguishability for Σ_P given $\mathbf{C}^{[1,\ldots,T]}$. This security game is formalised in Definition 14.

Definition 14. *Let $\Sigma =$ (*Setup,Streaming,Search*) be the DSSE scheme of ShieldDB, λ be the security parameter, and $\mathcal{A}$ be a persistent adversary. Let $\mathbf{C}^{[1,\ldots,T]}$ be a set of $(\mathcal{L}_P, \alpha, \mathcal{F})$-acceptable constraint sets. Let u_t (streaming) (resp. Q_t (search)) be an update (resp. query set, i.e., $Q_t = \{q_{t,1}, \ldots, q_{t,n}\}$) at time t, and $\mathcal{L}^{Stream}$ (resp. $\mathcal{L}^{Srch}$) be the leakage after the query u(resp. q), respectively. Let $\mathrm{Ind}_{DSSE,\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}}$ be the following game:*

$\mathrm{Ind}_{DSSE,\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}}(\lambda)$ *Game:*

$b \xleftarrow{\$} \{0, \ldots, \alpha - 1\}$
$(\mathfrak{C}^{\mathcal{DB}_0}, \mathrm{DB}^0_t, \ldots, \mathrm{DB}^{\alpha-1}_t) \leftarrow \mathcal{A}(1^\lambda)$
$(K, \mathrm{EDB}^b_0) \leftarrow \mathsf{Setup}(\mathrm{DB}^b_0)$
for $t = 1$ ***to*** T ***do***
  $\mathrm{EDB}^b_t \leftarrow \mathsf{Streaming}(u_t)$
  $(\mathfrak{C}^{\mathcal{DB}_t}, Q_t) \leftarrow \mathcal{A}(\mathrm{EDB}^b_t)$
  $\{\tau^b_{t,1}, \ldots, \tau^b_{t,n}\} \leftarrow \mathsf{Search}(Q_t, \mathrm{EDB}^b_t)$
end for
$b' \leftarrow \mathcal{A}(\tau^b_{T,1}, \ldots, \tau^b_{T,n})$
if $b = b'$ ***return*** *1, otherwise return 0*

where $\tau^b_{t,i}$ *presents the transcript of the query* $q_{t,i}$*, and with the restriction that, given* $\mathbf{C}^{[1,\dots,T]} = \{\mathfrak{C}^{\mathcal{DB}_0}, \dots, \mathfrak{C}^{\mathcal{DB}_T}\}$*, for* $\mathfrak{C}^{\mathcal{DB}_t} = \{C^{\mathrm{DB}_t}, \mathrm{DB}_t \in \mathcal{DB}_t\}$*, for all the* $H^i_t = (\mathrm{DB}^i_t, q_{t,1}, \dots, q_{t,n})$*,*

- $C^{\mathrm{DB}_t} \in \mathfrak{C}^{\mathcal{DB}_t}$*, and* $\forall 0 \leq i \leq (\alpha - 1), C^{\mathrm{DB}_t}(H^i_t) = \texttt{true}$
- $\mathcal{L}(H^0_t) = \cdots = \mathcal{L}(H^{\alpha-1}_t)$
- $\mathcal{F} = (\mathbb{F}^1, \dots, \mathbb{F}^T) = \texttt{true}$

We say that Σ *is* $(\mathcal{L}_P, \mathbf{C}^{[1,\dots,T]}, \alpha, \mathcal{F})$*-constrained-adaptively-indistinguishable if for all probabilistic polynomial time adversary* $\mathcal{A}$*,*

$$\begin{aligned} &\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\dots,T]},\alpha,\mathcal{F}}(\lambda) = \\ &\big|\mathbb{P}[\mathrm{Ind}_{\mathrm{DSSE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\dots,T]},\alpha,\mathcal{F}}(\lambda) = 1] - \frac{1}{\alpha}\big| \leq negl(\lambda). \end{aligned} \tag{4.8}$$

We can see that Σ_P offers at least $log(\alpha)$ bits of security. Given $\mathbf{C}^{[1,\dots,T]}$ is an $(\mathcal{L}_P, \alpha, \mathcal{F})$-acceptable set of constraint sets, we can analysing every transcript in the set $\{\tau^b_{t,1}, \dots, \tau^b_{t,n}\}$ under the choice of α. Let u_t simply contains a pair of (w_i, id), we can make an observation in ShieldDB as follows.

If $ST[w_i].c = 0$, presenting that w_i appears in the first time, then *Padding Controller* checks the cache cluster that expects to have w_i against the *first batch* condition. We recall that the condition ensures the existence of all keywords in the cluster before padding. If the condition is fail, we see that both EDB^0_i and EDB^1_i are indistinguishable under the choice of $b \xleftarrow{\$} \{0, \dots, \alpha - 1\}$. The reason is because the challenger does not send any batch to the server. In contrast, if the condition is passed, *Padding Controller* pads all the keywords in the cluster to be the same length and encrypt them before sending a batch to the server. Meanwhile, $ST[w_i].c$ gets updated. Accordingly, EDB^b_t is indistinguishable under the choice of because these databases receive the batch of keywords that have the same length. The *first batch* condition is crucial when ShieldDB is against the persistent adversary. It ensures there is no new keyword in the cluster appears in subsequent batches. Hence, the adversary cannot distinguish when a new keyword is added in EDB^b_i.

If $ST[w_i].c > 0$ and *first batch* condition has been met, *Padding Controller* performs padding similarly with the padding strategy against the non-persistent adversary, presented in Algorithm 1. We can also see that EDB^b_t is indistinguishable because *Padding Controller* guarantees all the keywords in a cluster have the same length.

Now, we start analysing the query q_t, i that queries the keyword w_i, with $ST[w_i].c = 0$ or $ST[w_i].c > 0$.

If $ST[w_i].c = 0$, the adversary cannot guess the picked database because $\tau^b_{t,i}$ return nothing.

If $ST[w_i].c > 0$, $\tau^b_{t,i}$ is indistinguishable to all other query keywords in the same cluster at time t.

Theorem 2. *Let* $\Sigma_P = (\mathsf{Setup}, \mathsf{Streaming}, \mathsf{Search})$ *be our DSSE scheme, and* $\mathbf{C}^{[1,\ldots,T]} = \{\mathfrak{C}^{\mathcal{DB}_0}, \ldots, \mathfrak{C}^{\mathcal{DB}_T}\}$ *is a set of constraint sets. If* Σ *is* $\mathcal{L}_P$*-constrained-adaptively-indistinguishable secure, and* $\mathbf{C}^{[1,\ldots,T]}$ *is* $(\mathcal{L}_P, \alpha, \mathcal{F})$*-acceptable, then* Σ *is* $(\mathcal{L}_P, \mathbf{C}^{[1,\ldots,T]}, \alpha, \mathcal{F})$*-constrained-adaptively-indistinguishability secure.*

Proof. Let $\mathcal{A}$ be an adversary in the $\mathrm{Ind}_{\mathrm{DSSE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}}$ game. We construct an adversary $\mathcal{B}$ against the game. $\mathcal{B}$ first randomly picks two integer $\alpha_0, \alpha_1 \in \{0, \alpha - 1\}$. Then, $\mathcal{B}$ starts $\mathcal{A}$ and receives α databases $(\mathrm{DB}_0^0, \ldots, \mathrm{DB}_0^{\alpha-1})$. Upon giving the pair $(\mathrm{DB}_0^{\alpha_0}, \mathrm{DB}_0^{\alpha_1})$ to the challenger, where the challenger holds a random secret bit b, $\mathcal{B}$ receives the challenge encrypted database EDB_0^* which she forwards to $\mathcal{A}$. Then, for every $t \in [1, \ldots, T]$, we have:

- $\mathcal{A}$ sends u_t (stream queries) to challenger and receives EDB_t^*.
- $\mathcal{A}$ outputs $((q_{t,1}^0, \ldots, q_{t,1}^{\alpha-1}), \ldots, (q_{t,n}^0, \ldots, q_{t,n}^{\alpha-1}))$ and gives to $\mathcal{B}$.
- To respond, $\mathcal{B}$ outputs $((q_{t,1}^{\alpha_0}, q_{t,1}^{\alpha_1}), \ldots, (q_{t,n}^{\alpha_0}, q_{t,n}^{\alpha_1}))$ to the game and receives back the transcripts $(\tau_{t,1}^*, \ldots, \tau_{t,n}^*)$, and forwards them to $\mathcal{A}$.

After executing all $t \in [1, \ldots, T]$, $\mathcal{A}$ outputs the integer α'. If $\alpha' = \alpha_0$, $\mathcal{B}$ outputs $b' = 0$, else if $\alpha' = \alpha_1$, $\mathcal{B}$ outputs $b' = 1$, and otherwise outputs the probability $1/2$ for the output 0 and the probability $1/2$ for the output 1.

We first make an observation: for the pair $(H_t^{\alpha_0}, H_t^{\alpha_1})$ at $\forall t \in [1, T]$, the views of the adversary $\mathcal{B}$ are indistinguishable due to $\mathcal{L}_P(H_t^{\alpha_0}) = \mathcal{L}_P(H_t^{\alpha_1})$, presenting both satisfying $\mathbf{C}^{[1,\ldots,T]}$. Then we can formalise $\mathcal{B}$ as follows:

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\mathcal{F}}(\lambda) = \left|\mathbb{P}[b = b'] - \frac{1}{2}\right| \leq negl(\lambda) \tag{4.9}$$

Now, we evaluate $\mathbb{P}[b = b']$ as follows.

$$\begin{aligned}
\mathbb{P}[b = b'] = \\
&\mathbb{P}[b = b' | \alpha' \in \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] \\
&+ \mathbb{P}[b = b' | \alpha' \notin \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \notin \{\alpha_0, \alpha_1\}] \\
&= \mathbb{P}[b = b' \cap \alpha' \in \{\alpha_0, \alpha_1\}] \\
&+ \mathbb{P}[b = b' | \alpha' \notin \{\alpha_0, \alpha_1\}] \cdot \mathbb{P}[\alpha' \notin \{\alpha_0, \alpha_1\}] \\
&= \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{DSSE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}} \text{ game}] \\
&+ \frac{1}{2}\left(1 - \mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}]\right)
\end{aligned} \tag{4.10}$$

Now, we evaluate $\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}]$ as follows.

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \mathbb{P}[\alpha' = \alpha_0] + \mathbb{P}[\alpha' = \alpha_1]$$

Since we have

$$\mathbb{P}[\alpha' = \alpha_0] + \mathbb{P}[\alpha' = \alpha_1] = \mathbb{P}[\alpha' = \alpha_b | b = 0] + \mathbb{P}[\alpha' = \alpha_b | b = 1]$$

then,

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \frac{1}{2}\left(\mathbb{P}[\alpha' = \alpha_b | b = 0] + \mathbb{P}[\alpha' = \alpha_0]\right) + \frac{1}{2}\left(\mathbb{P}[\alpha' = \alpha_b | b = 1] + \mathbb{P}[\alpha' = \alpha_1]\right)$$

We note that $\mathbb{P}[\alpha' = \alpha_b]$ is the probability $\mathcal{A}$ wins the 1-out-of-α indistinguishability game, and α_0 and α_1 are uniformly selected from $\{0, \alpha - 1\}$, then we have

$$\mathbb{P}[\alpha' \in \{\alpha_0, \alpha_1\}] = \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{DSSE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}} \text{ game}] + \frac{1}{\alpha} \tag{4.11}$$

Applying Eq. 4.11 to Eq. 4.10, we have

$$\mathbb{P}[b = b'] = \frac{1}{2} \cdot \mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}} \text{ game}] + \frac{1}{2} - \frac{1}{2\alpha}$$

Then, from Equation 4.9, we can derive

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\mathcal{F}}(\lambda) = \frac{1}{2}\left(\mathbb{P}[\mathcal{A} \text{ wins the } Ind_{\mathrm{SE},\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}} \text{ game}] - \frac{1}{\alpha}\right) \tag{4.12}$$

Applying Equation 4.12 to Equation 4.8, finally, we can conclude

$$\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{B},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\mathcal{F}}(\lambda) = \frac{1}{2}\mathbf{Adv}^{\mathrm{Ind}}_{\mathcal{A},\mathcal{L}_P,\mathbf{C}^{[1,\ldots,T]},\alpha,\mathcal{F}}(\lambda) \tag{4.13}$$

□

4.5 Implementation and Evaluation

A simple way to implement the padding service P of ShieldDB is that *Cache Controller* and *Padding Controller* are maintained synchronously in a single

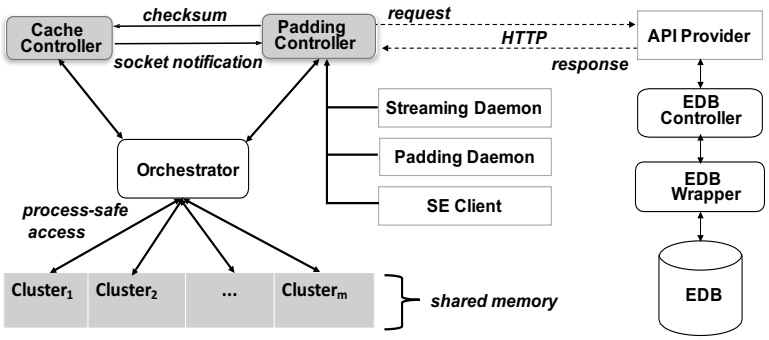


Figure 4.6: Implementation of ShieldDB

process. That is, *Cache Controller* is idle while *Padding Controller* performs padding and encryption, and vice versa. Then, encrypted batches are uploaded to the server S. We observe that this single process becomes extremely slow in the long run because *Cache Controller* and *Padding Controller* cannot make use of CPU cores in parallel. As a result, there are a very few batches uploaded to S.

To address the above bottleneck, we propose *Orchestrator*, a component bridging data flow between *Cache Controller* and *Padding Controller*. The design of *Orchestrator* is as follows.

4.5.1 System Implementation

Orchestrator enables ShieldDB to maximise the usage of CPU cores by splitting two controllers to process in parallel. Figure 4.6 depicts the implementation of the system. In details, *Cache Controller* and *Padding Controller* are spawned as separate system processes during setup. *Orchestrator* acts as an independent proxy manager managing the cache clusters in P's shared memory. It provides process-safe access methods of collecting, clearing, and updating data in a given cluster.

The communication between *Cache Controller* and *Padding Controller* is performed by sockets during the streaming operations. *Cache Controller* acts as a client socket, and notifies *Padding Controller* in the order of clusters that are ready for padding as per padding strategy. Then, *Cache Controller* awaits a checksum notified by *Padding Controller*. The checksum reports the number of keyword and document id pairs in the cached cluster. Note that *Padding Controller* only collects necessarily cached data for padding upon the *high* or *low* padding mode.

Apart from these components, ShieldDB contains *Padding Daemon*, *Streaming Daemon*, and *SE Client*. They are activated by *App Controller* during setup. *Padding Daemon* provides *Padding Controller* with the access to a bogus dataset, and maintains the track of remaining bogus entries for each keyword. It will generate a new bogus dataset if it is run out. *Streaming Daemon* allows *App*

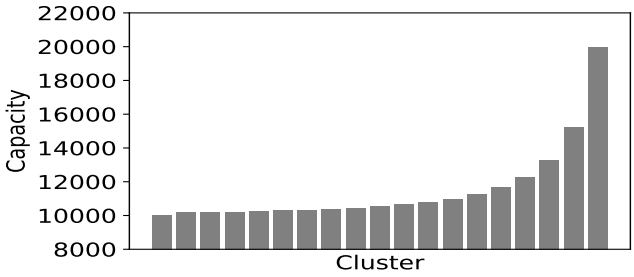


Figure 4.7: Cache capacities for $\alpha = 256$

Controller to setup HTTP request/response methods to S's address. *SE Client* deploys our encryption protocols at C. This service is separated so that later protocol updates are compatible to other components in the system.

At S, *API Provider* provides RESTful APIs to serve C's HTTP requests. By calling streaming API requests, *API Provider* then passes collected batches in streaming to the *EDB Controller*. This component executes the insertion protocol as presented in Algorithm 4. ShieldDB introduces a component called *EDB Wrapper*, which separates *EDB Controller*'s protocols from any database storage technology.

4.5.2 Experimental Setup

We evaluated ShieldDB to understand the applicability of the padding strategies by investigating (1) the streaming throughput, padding overhead, and local cache load of the *Padding Service* when using different padding strategies against the non-persistent and persistent adversaries, (2) the corresponding EDB size and search latency relating to the padding overhead, and (3) the efficiency of flushing in reducing the cache load and the padding overhead reduction when re-encryption is applied.

ShieldDB is developed using Python and the code is published online[2]. We use standard packages of Pycrypto (2.6.1) to implement cryptographic primitives (SHA256 for cryptographic hash functions and AES-128 cipher for pseudo-random functions) and NLTK (3.3) for textual processing. We deploy ShieldDB in Azure Cloud and run on an isolated DS15 v2 instance (Intel Xeon E5-2673 2.4GHz CPU with 20 cores and 140G RAM), where Ubuntu Server 17.1 is installed. The controllers of the padding service are implemented by using Python multiprocessing package. For simplicity, we co-locate the *Client* and *Padding Service* at the same instance. At the server side. *API Controller* works on top of the Flask-a micro web framework, while EDB is realised by RocksDB, a key-value storage.

We select the Enron data set[3], and extract 2,418,270 keyword/document id

[2] `https://github.com/MonashCybersecurityLab/ShieldDB`

[3] Enron email dataset: `https://www.cs.cmu.edu/~./enron/`

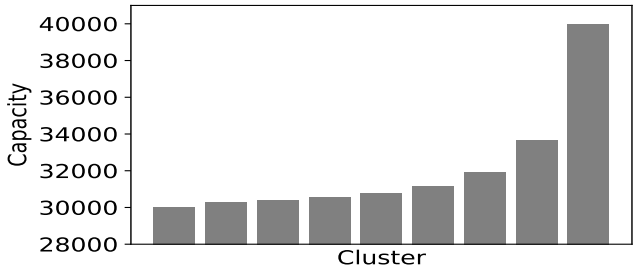


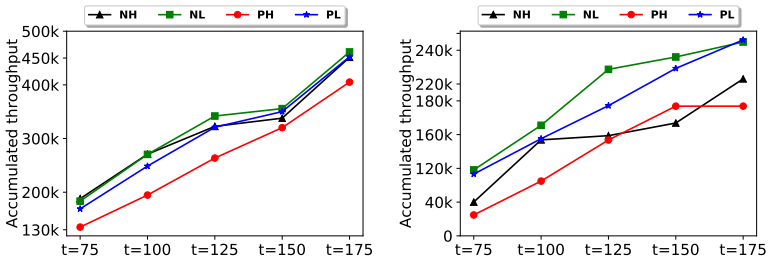
Figure 4.8: Cache capacities for $\alpha = 512$

pairs from the top 5,000 most frequent keywords in the dataset as the keyword space in our experiment. We group them and allocate the cache capacity for each keyword cluster based on their frequencies, as introduced in Section 4.3.1. Figures 4.7 and 4.8 presents the normalised cache capacities of these clusters at different values of α. Recalled that α indicates the minimum number of keywords in each cluster (see Section 4.3.1). During the setup, ShieldDB generates a padding dataset for the keyword set. In our experiments, the dataset is estimated empirically enough to be used in streaming data up to 175 seconds for both $\alpha = 256$ and $\alpha = 512$. In details, the dataset contains 1,859,877 bogus pairs ($\approx$ 389 Kb).

To create the streaming scenario, the *Client* groups every 10 documents in the Enron data set as a batch (approx. 560 stemmed keyword/id pairs) and continuously inputs batches to the system. Note that we do not limit the processing capability of the *Padding Service* P and we let it continuously handle batches in a queue sent by the *Client*. In every batch, P performs the padding and encryption, and then continuously streams encrypted batches to the *Server*. To faithfully understand the performance of padding, we deploy the client and server to the same dedicated instance so that the impact of network I/O is minimised. Note that we begin to record the performance of ShieldDB after the cold start period of 75 seconds.

We experiment ShieldDB with different combinatorial settings of padding strategies and modes. They are denoted as *NH* (strategy against non-persistent adversary via *high* mode), *NL* (non-persistent padding strategy via *low* mode), *PH* (strategy against persistent adversary via *high* mode, and *PL* (strategy against persistent adversary via *low* mode). The performance of ShieldDB is evaluated via a set of measurements such as system throughput, local cache size, used bogus pairs, EDB size, and search time. Here, the system throughput represents the total accumulated number of real (w, id) pairs that have been encrypted and inserted to EDB.

Remark: Our focus is to analyse the system performance with different settings of padding strategies and modes related to the security as mentioned in Section 4.3.2. The empirical settings of 175 second streaming period and 10 doc-



(a) $\alpha = 256$ (b) $\alpha = 512$

Figure 4.9: Accumulated throughput

uments per batch are used for the evaluation under a stable workload, not causing performance bottleneck, when the *Client* and *Padding Service* are co-located at the same Azure instance. Other parameters of a streaming period expects to result in the same observation as we obtained. The batch size can be adjusted empirically based on the application and client's resources.

4.5.3 Evaluation

We measure the performance of ShieldDB at both *Padding Service* and the untrusted server S over a 175-second streaming period. In details, we evaluate the performance of *Padding Service* with the three different metrics of accumulated throughput, local cache size, and padding overhead when setting $\alpha = 256$ and $\alpha = 512$. Then, we study the performance of S by observing EDB size, search time, and the average result length of query keywords.

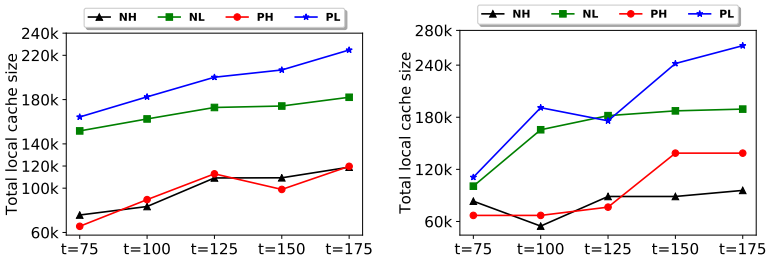
System throughput: We first measure the accumulated throughput over time when ShieldDB is deployed with different padding modes of *NH* (non-persistent using high padding mode), *NL* (non-persistent using low padding mode), *PH* (persistent using high padding mode), and *PL* (persistent using low padding mode) (see Figure 4.9). We also monitor the number of batch insertions and the average batch processing time of *Padding Controller* to evaluate the throughput difference between these padding strategies (see Table 4.2).

Fig. 4.9(a) shows that these padding modes have similar throughput at a lower $\alpha = 256$. However, the overall throughput reduces nearly a half when setting $\alpha = 512$ (see Fig 4.9(b)). It is explained that padding overhead and encryption cost are higher when more keywords are allocated in each cluster. Consequently, the throughput will be decreased.

Table 4.2 also supports that finding when fewer batches are inserted to S and the average processing time per batch takes a longer time when setting $\alpha = 512$. Furthermore, when setting $\alpha = 512$, Fig. 4.9(b) shows that *low* mode promotes more real keyword/id pairs to be inserted to EDB than *high* mode. In details,

Table 4.2: Batch processing results

Setting	**Batch Insertions**		**Avg. time/batch** (ms)	
	$\alpha = 256$	$\alpha = 512$	$\alpha = 256$	$\alpha = 512$
NH	30	5	7047.2	51384.41
NL	1919	497	113.94	456.87
PH	45	6	5280.58	45734.16
PL	1916	549	138.34	465.09

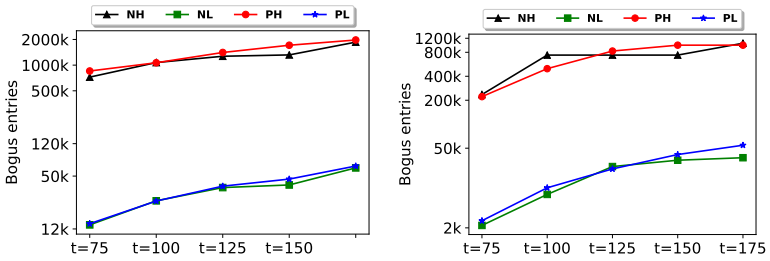


(a) $\alpha = 256$ (b) $\alpha = 512$

Figure 4.10: Local cache size

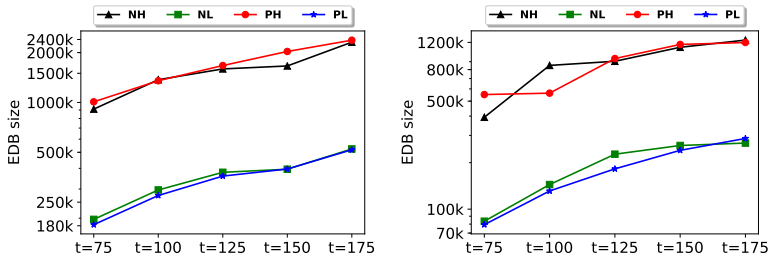
the throughput of *NL* is 1.23 times higher than the throughput of *NH*, and *PL*'s is about 1.51 timer higher than *PH*'s. Table 4.2 also supports this finding when it reports that *low* mode creates more batch insertions than *high* mode, while its average batch processing time is completely negligible compared to that value of the latter. This observation shows the efficiency of *low* mode since it only performs necessarily minimum padding for keywords in every batch. In contrast, *Padding Controller* takes longer time under *high* padding mode due to higher padding overhead and the longer encryption time taken by the large number of bogus pairs.

Local cache size: To investigate the overhead at the *Padding Service*, we monitor the local cache as shown in Fig. 4.10. In general, *low* mode results



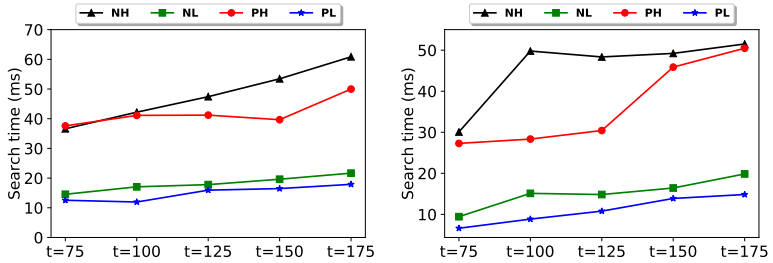
(a) $\alpha = 256$ (b) $\alpha = 512$

Figure 4.11: Bogus entries



(a) $\alpha = 256$ (b) $\alpha = 512$

Figure 4.12: EDB Size



(a) $\alpha = 256$ (b) $\alpha = 512$

Figure 4.13: EDB Size

in a larger number of cached pairs in cache clusters than *high* mode, regardless of padding constraints. The cache in *NL* consumes 150%∼197% larger space than the cache in *NH*. The load of cache in *PL* is 1.8∼2.5× higher than the load of the cache in *PH*.

Padding Overhead: We rely on the number of used bogus entries reported in Fig. 4.11 to compute the padding overhead of different combinatorial settings of padding strategies and modes. The padding overhead is estimated as the ratio between the bogus and real (throughput) pairs. We see that although *high* padding mode achieves a lower load of cache than *low* mode, it utilises more bogus pairs from the generated padding dataset than the latter. In details, the padding overhead of *NH* ranges from 3.8∼4.1 and from 5.6∼5.8 for $\alpha = 256$ and 512, respectively. In contrast, the padding overhead of *NL* ranges is marginal, varying from 0.07∼0.13 and 0.06∼0.16 for $\alpha = 256$ and 512, respectively. The reason is that a portion of streamed keyword/id pairs are still cached at the padding service. It also demonstrates that when α is large, *PH* generates a larger padding overhead than *NH* does. Specifically, the padding overhead of *PH* is in the range of 6.4∼8.9 for $\alpha = 512$. The reason is that *PH* will add bogus pairs for keywords that do not appear in the current time interval, while *NH* will not if the keywords have not existed.

Table 4.3: Result length with $\alpha = 256$

Setting	**Time intervals**				
	$t = 75$	$t = 100$	$t = 125$	$t = 150$	$t = 175$
NH	593.78	669.94	778.45	811.25	903.53
NL	109.856	144.66	164.40	171.56	186.04
PH	562.86	660.22	593.18	579.30	714.12
PL	89.25	82.38	107.92	110.57	126.43

Table 4.4: Result length with $\alpha = 512$

Setting	**Time intervals**				
	$t = 75$	$t = 100$	$t = 125$	$t = 150$	$t = 175$
NH	668.02	843.22	837.01	840.85	861.44
NL	81.64	140.95	147.73	168.17	174.40
PH	577.06	610.02	614.34	857.73	879.63
PL	51.66	85.33	89.21	93.82	112.32

EDB size: We report the number of real and bogus pairs in EDB over the time in Fig. 4.12. It demonstrates that *high* mode generates more data in EDB than *low* mode due to the selection of all cached pairs in clusters for padding and the large number of used bogus pairs.

Search time: To demonstrate the search performance, we configure the client to query 10% randomly selected keywords in EDB at timestamps, i.e, $t = 75$, 100 , 125, 150, and 175. Fig. 4.13 shows that *high* mode makes querying a keyword take a longer time, because S decrypts more bogus pairs. In contrast, the search time in *low* mode is shorter due to the fewer used bogus pairs. The search time in *NH* and *PH* is fluctuated due to the change of the result lengths of keywords in EDB as given in Table 4.3 and Table 4.4.

Flushing: We select two largest cache clusters to simulate the flushing operation. In particular, we set a small time window, 20 seconds, to trigger flushing. If these clusters do not exceed up to 75% of their original capacities, then the flushing operation is invoked. Figure 4.14 reports EDB size and cache size over the time with a scanning window of 20 seconds. The operation occurs at $t = 73$, 45, 80, 121, 144, 189, 222, 272, and 331 seconds. We observe that the cache size drops significantly at these timestamps since *Cache Controller* flushes the cached pairs to *Padding Controller*. Note that the EDB and cache sizes are flat while *Padding Controller* performs padding and encryption. Empirically, we observe that flushing operation averagely reduced the cache load efficiently up to

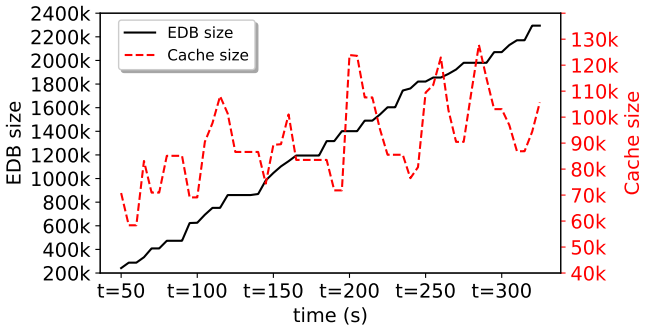


Figure 4.14: Flushing operation with $\alpha = 256$

Table 4.5: Re-encryption on the largest cluster

	Before	**During**	**After**
Bogus entries used	643, 131	230, 715	230, 715
Search time (ms)	379.37	0.03	210.18

Table 4.6: Overall performance of ShieldDB throughout a 175-second streaming period

Setting	**Throughput per second**		**Avg. cache load**		**Padding overhead**	
	$\alpha = 256$	$\alpha = 512$	$\alpha = 256$	$\alpha = 512$	$\alpha = 256$	$\alpha = 512$
NH	2, 634.27	1, 459.62	99, 347.8	82, 267.8	$3.8 \sim 4.12$	$5.6 \sim 5.8$
NL	2, 779.77	1, 515.74	168, 681.4	164, 960	$0.07 \sim 0.13$	$0.06 \sim 0.16$
PH	2, 702.05	1, 289.64	97, 351.6	97, 557.6	$4.8 \sim 6.3$	$6.4 \sim 8.9$
PL	2, 833.46	1, 590.46	195, 702.2	196, 413.6	$0.08 \sim 0.14$	$0.08 \sim 0.23$

$1.9 \sim 2.8\times$ across the padding strategies in the same streaming period.

Re-encryption: To investigate the performance of re-encryption, we experiment ShieldDB after 175 seconds operated with *NH* at $\alpha = 256$. We select the keyword cluster that has the most entries stored in EDB for the re-encryption. This keyword set is also re-used as the query set to benchmark the query performance before, during, and after re-encryption. There are 180,677 real entries associating with 256 keywords of this cluster. Table 4.5 demonstrates the performance of the re-encryption. This operation takes 131.3 seconds for fetching process, and 103.11 seconds for padding and re-insertion. During the operation, the average query time per keyword is the smallest due to the deletion of all entries in the selected cluster. Note that this query time takes into account the search over local cache clusters if the keyword is not available in EDB. After re-encryption, the number of bogus entries used for the cluster is nearly reduced by 64.1%, making the average search time shorter.

Overall performance: Table 4.6 summarises the performance of *Padding Service* regarding three critical measurements of throughput per second, average cache size at every second, and the overall padding overhead. As seen, there are

Table 4.7: Overall performance of the insecure streaming system and the forward-private SE streaming system

	insecure system	Forward-private system				
		$t = 75$	$t = 100$	$t = 125$	$t = 150$	$t = 162$
Throughput	4.3×10^4	1.73×10^4	1.71×10^4	1.71×10^4	1.73×10^4	1.74×10^4
Avg. result length	483.66	281.5	384.8	476.53	513.61	483.61
Search latency	5.12	20.9	24.62	33.75	40.95	43.59
Storage overhead	2.41×10^6	1.3×10^6	1.7×10^6	2.14×10^6	2.17×10^6	2.41×10^6

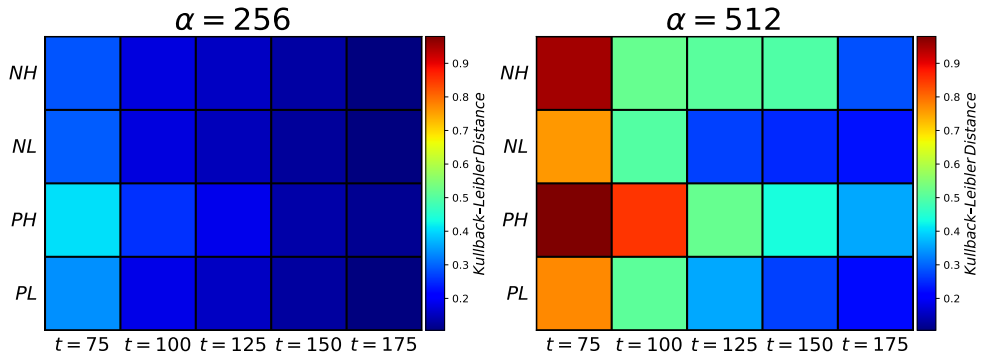


Figure 4.15: The difference in streaming distribution

no perfect padding strategies that can achieve a great balance. *Low* padding mode makes a higher throughput value and lightens padding overhead, but it incurs a significant cache load. In contrast, *high* padding mode makes the cache load lightweight, but it introduces a higher overhead.

Note that the padding strategies against the persistent adversary are also applicable to the non-persistent adversary. The *firstBatch* condition can theoretically make some clusters might be not achieved in a long time if some keywords never appear. However, this is not the case in our current experiments. Therefore, the throughput for *PH* and *NH*, and *PL* and *NL* is close, respectively.

The value α relates to the number of keywords in clusters. A higher value indicates that more keywords are co-located in the same cluster. Hence, they all will have the same result length after padded. From the results, ShieldDB shows the tradeoff when selecting a higher value of α. That is, the throughput is declined nearly double while padding overhead increases almost twice (see Table 4.6).

Comparison with baselines: We further investigate the security and performance trade-off between ShieldDB and two baselines. The first baseline (aka Baseline-I) is an insecure system for which batches of un-encrypted keyword/id pairs without padding are streamed to the server's storage. We define the batch size as 256 pairs. The second one (aka Baseline-II) is also the streaming system without padding, but it realises our searchable encryption scheme with forward privacy, to encrypt the pairs of every batch insertion.

We measure the overall performance of these baselines by using the same streaming database and the measurement metrics as evaluated for ShieldDB. It

is clear that Baseline-II brings 2.5× overhead in addition throughput compared to Baseline-I. The reason is because that the encrypted entries of keywords in the same batch indeed are generated from the ephemeral key of the batch and keyword's extracted state. We note that Baseline-I maintained a constant throughput and completed streaming within 55 seconds, while Baseline-II finished in 162 seconds (Table 4.7). The throughput overhead ShieldDB brought forward is about $6.04 \sim 6.5\times$ (resp. $10.7 \sim 11.74\times$) lower than performance of Baseline-II when setting $\alpha = 256$ (resp. 512). We note that overhead is caused by the encryption of additional bogus pairs introduced in every batch insertion. The average result length for keywords streamed to the EDB of ShieldDB is about $1.8 \sim 2.3\times$ (*high* padding mode used) greater than that value if Baseline-II is deployed. The search latency of ShieldDB is almost double ($1.7 \sim 2.1\times$), slower than Baseline-II. We observe that the security enhanced by the padding and forward privacy would overall bring the streaming throughput per second $\sim 16.3\times$ (resp. $\sim 27\times$) slower than Baseline-I when setting $\alpha = 256$ (resp. 512).
Empirical analysis of streaming distribution: Next, we investigate how the streaming distribution of real data outsourced by ShieldDB to EDB changes over the time. To do this, we consider the training distribution used to generate the padding dataset and cluster's caches extracted from the training dataset in the Setup as the baseline distribution. Note that the training distribution was different with respecting to α (i.e., the minimal number of keywords in every cache cluster) (see Equation 1). Then, we monitor the streaming distribution of real data at different times t when ShieldDB employs different combinatorial settings of padding strategies and modes. In particular, we use the Kullback–Leibler (KL) distance [122] to measure the difference between such streaming distributions and the baseline distribution (Figure 4.15).

Our observation shows that the streaming distribution was different compared to the baseline at the early streaming time (i.e., $t = 75 - 150$). Then, it tended to converge to the baseline when the streaming dataset was almost outsourced completely ($t \geq 175$). At this time, the padding dataset was also almost used. The reason for that is because the completed streaming dataset shares the same distribution with the training dataset. However, at the earlier time, the difference was large because there was some keywords in the training distribution that did not appear yet in these early streaming batches. We note that, with $\alpha = 256$, the persistent padding settings (i.e., *PH* and *PL*) have the largest distribution difference since it requires the existence of all keywords in the cluster at the *firstBatch* (in Algorithm 6). With $\alpha = 512$ (i.e., more keywords required in clusters), the distribution difference was double (i.e., $0.8 \sim 1.0$ KL unit) since the setting causes a longer waiting period before the padding strategies meet, under the same streaming rate.

4.5.4 Discussion on the deployment of ShieldDB

We note that the above experiments consider the keyword frequency distribution in setup is similar to the one in a period of streaming operation. We deem that the assumption and the corresponding setting of ShieldDB for deployment can be practically held in practice.

First, this setting is applicable to streaming applications for which the underlying distribution does not change much over time or it is known in advance, like the known data range of the IoT sensors [123, 124]. Second, in our observation (Figure 4.15), we show that the streaming distribution changes towards the training distribution over a streaming period, not requiring the exact matching between them for any particular interval. Therefore, the assumption can at least be hold for that duration, and better than assuming that the streaming and training distribution are close for any particular time interval. In addition, we note that the setup operation can be re-invoked again to re-cluster keywords based on up-to-date streaming data if the streaming distribution is different from the training distribution. In that way, the re-clustering can use that up-to-date streaming distribution as the training distribution.

We are aware that the keyword distribution difference can cause a long tail effect when applying the proposed padding strategies to low frequent keywords. For instance, if a keyword only occurs in the first batch of that keyword's cluster and disappears for all subsequent batches, *Padding Controller* still pads that keyword during subsequent batches when padding strategies NH and NL are used. As seen in the above experiments, such different distribution can happen at the early streaming stages. We also note that, when the streaming distribution differs from the training distributions, it may cause some "cold" clusters and/or the intensive usage of the padding datasets in some "hot" clusters. As a results, the overall streaming throughput can be slowed down.

To mitigate the above issues, ShieldDB offers flushing and re-encryption operations regarding the highly varied frequency of streaming keywords. As experimented, flushing could quickly reduce $1.9 \sim 2.8\times$ the cache load to boost the "cold" clusters. In addition, re-encryption could lowered 64% the amount of bogus pairs used by re-padding all keywords in the "hot" cluster. The above result is obtained when we checked and applied the operations for every fixed time window. Nevertheless, it is non-trivial to optimise the padding overhead in the streaming setting and we leave it as future work.

4.6 Discussion

ShieldDB employs SSE [65] as an underlying building block to enable single-keyword encrypted search. Curtmola et al. [66] and Kamara et al. [69] formalise the security of SE for static and dynamic databases respectively, and devise concrete

constructions with sublinear search time. A line of schemes [87, 95, 121, 70, 2, 3] (just to list a few) are proposed to improve performance and expressiveness of SE [91, 93]. Driven by leakage-abuse attacks [74, 76, 125], new schemes [78, 82, 81] with less leakage in search and update are proposed to achieve forward and backward security. Note that, although oblivious RAM [25, 26] provides the highest protection for the *Server*'s memory access pattern, we do not consider it for ShieldDB due its inefficient capability in the streaming setting. In details, the approach requires more computation and storage at the *Client*, and the communication between the *Client* and the *Server*. Also, ORAM does not hide the size of the query results, unless there is non-trivial padding.

The investigation of the practical streaming ShieldDB in this chapter helps us to understand the capability of leakage-abuse attacks in the dynamic (addition) SE. For example, the non-persistent adversary can guess the query keyword at any time interval if naive padding approach is applied. Even more importantly, the persistent adversary who monitors the change of the EDB can easily guess newly added keywords into the DB over the time. Using our application scenario as an example, we note that the *Padding Service P* in ShieldDB is deployed at the enterprise gateway of a private network, and it aggregates plaintext documents uploaded by employees of the enterprise company (represented as the *Client C*). We stress that the service adapts padding countermeasures during encryption and then streams encrypted batches to the encrypted database hosted by the server S. Note that, the global knowledge of the adversary reflects the plaintext documents received by P, which is the data before padding and encryption. Our attack's assumption is in line with the leakage-abuse attacks. We do not consider that the server can distinguish query tokens from different clients. In our scenario, all query tokens from clients are sent from the enterprise gateway.

We also note that all prior leakage-abuse attacks are focused on the single keyword search. We are not aware of any of the literature that has proposed leakage-abuse attacks based on boolean queries. Therefore, we still focus on mitigating the leakage-abuse attacks on the single keyword query model. But note that the padding countermeasure has been demonstrated to protect the co-relations between keywords and documents; namely, the leakage across multi-keyword search can also be protected.

Another important aspect of the chapter is how to design effective padding strategies, in particular, to mitigate the non-persistent and persistent adversaries. Our approach relies on the dynamic clustering solutions that carefully track dynamic keyword states over the time. In our design, we always ensure that there are at least $(\alpha - 1)$ other keywords such that all of them have the same query result length because of padding, where α is the size of the cluster. We note that our approach is completely different from k-anonymity since k-anonymity would not address the leakage-abuse attacks in our scenarios. First, it might not be possible to directly find k keywords with the same query lengths for all keywords. In the count attack, a portion of keywords have unique result lengths. Second,

leakage-abuse attacks also exploits the relationships between keywords for query recovery, and this has been shown very powerful when queries have the same query lengths. Therefore, only group keywords with same result lengths together without padding still suffers from leakage-abuse attacks, not to mention the more powerful adversaries that can monitor the database over the time intervals.

It is not too hard to see the limitation in this chapter: the system requires the training dataset in the `setup` of ShieldDB. That is the existence of a sample dataset in which the keyword frequencies are close to the real ones. We deem that the assumption can be practically held because of the following reasons:

- The assumption is feasible in practical streaming applications for which the underlying distribution does not change much over time, or it is already known in advance, like the known data range and frequencies of the IoT environment.

- In our observation, we show that the streaming distribution changes towards the training distribution in the long run over a streaming period, not requiring the exact matching between them for any particular interval. Therefore, the assumption can at least be held for that duration, and even the assumption is more practical than assuming that the streaming and training distribution are close for any particular time interval.

- We note that the setup operation can be re-invoked again to re-cluster keywords based on up-to-date streaming data if the streaming distribution is heavily different from the training distribution after a certain period of system deployment. In that way, the re-clustering can use that up-to-date streaming distribution as the training distribution. With the above mechanism, we believe that our assumption regarding the streaming distribution can be held for a certain foreseen period of the upcoming streaming events.

Notwithstanding the training data set requirement, if the distribution of the training and real datasets is not similar, the keyword distribution difference can cause a long tail effect when applying the proposed padding strategies to low frequent keywords. For instance, if a keyword only occurs in the first batch of that keyword's cluster and disappears for all subsequent batches, the *Padding Controller* still pads that keyword during subsequent batches when padding strategies *NH* and *NL* are used. As seen in the above experiments, such different distribution can happen at the early streaming stages. We also note that, when the streaming distribution differs from the training distributions, it may cause some *cold* clusters and/or the intensive usage of the padding datasets in some *hot* clusters. As a result, the overall streaming throughput can be slowed down.

We note that recent works [111, 112] proposed volume-hiding encryption schemes to mitigate the leakage-abuse attacks. We note that those schemes are focused on the static setting, as they resort to specialised data structures and

constructions. First, they are not dynamic friendly. Specifically, multi-hashing and cuckoo hashing techniques are adopted as the underlying data structures. It is not easy to insert new data into those data structures, and we are not aware any existing volume-hiding schemes support efficient updates. Second, volume hiding schemes may hide the size of the query result, but it is not clear whether they can protect the relationships between different query keywords when applying them into the context of keyword search.

ShieldDB can also be fit into a line of research on designing encrypted database systems. Most of existing encrypted databases [33, 35, 37, 38, 36, 126] focus on supporting rich queries over encrypted data in SQL and NoSQL databases. They mainly target on query functionality and normally integrate different primitives together to achieve the goal. Like the issues in SE, inference attacks against encrypted databases [53, 127] are designed to compromise their claimed protection. To address this issue, one approach is to use advanced cryptographic tools such as secure multi-party computation [35, 37]. Note that padding can also be adapted to mitigate inference attacks. A system called Seabed [36] proposes a schema for RDBMS that introduces redundant data values in each attribute of data records to hide the frequency of the underlying data values. Compared with the above systems, ShieldDB focuses on the document-oriented data model and supports keyword search over encrypted documents.

Chapter 5

Accelerating Forward and Backward SSE schemes

In [107], Cash et al. introduced the concept of active attacks against SSE; the leakage in data update operations can be exploited to compromise the claimed security of SSE. After that, Zhang et al. [76] proposed the first instantiation of active attacks called file-injection attacks through the exploitation of the leakage in data addition. This work raises a natural question: whether a DSSE scheme with less leakage can be designed to mitigate existing and even prevent prospective active attacks. To address this question, *forward* and *backward-private* DSSE schemes [78, 82, 81, 80] have drawn much attention recently.

In DSSE, the notion of *forward privacy* means that the linkability between newly added data and previously issued search queries should be hidden against the server, and the notion of *backward privacy* means that the linkability between deleted data and search queries after deletion should be hidden. To achieve higher security for DSSE, the efficiency of DSSE is compromised. Existing *forward* and *backward* private DSSE schemes [82, 81, 80] introduce large overhead in storage and computation at both client and server, and/or increase the client-server interaction. Therefore, in this chapter, we investigate how to design *forward* and *backward-private* DSSE schemes that efficiently support large addition/deletion. This research question is important to make DSSE more practical deployment[1].

5.1 Existing SGX-supported Backward-private Constructions

Recently, hardware assistance, like Intel SGX, has demonstrated as an effective solution to maintain the efficiency of DSSE, where native code and data can be executed in a trusted and isolated execution environment. For example, recent

[1]This chapter is partly based on [2, 3]

Table 5.1: Comparison with previous SGX-supported Type-II backward-private schemes. N, D, and W denote the total number of keyword/document pairs, total number of documents, and total number of keywords, respectively. d presents the number of deleted documents. n_w is the number of (current, non-deleted) documents containing w, a_w is the total number of entries (including addition and deletion updates) performed on w, d_w denotes the number of deletions performed on w. r is the predefined number of necessary dummy entries to be inserted in oblivious operations.

SGX Schemes	Communication between enclave and server				Enclave Computation		Client Storage	Enclave Storage	BP Type
	#Search rounds (*ecall* + *ocall*)	Search	#Update *ocalls*	Update	Search	Update			
Bunker-B [83]	a_w	$O(n_w)$	a_w	$O(1)$	$O(a_w)$	$O(1)$	$O(W \log D)$	–	II
SGX-SE1	$(d + d_w)\star$	$O(n_w)$	$n_w\dagger$	$O(1)$	$O(n_w + d)\ddagger$	$O(1)$	–	$O(W \log D + d)$	II
SGX-SE2	$(d_w)^\star$	$O(n_w)$	$n_w\dagger$	$O(1)$	$O(n_w)$	$O(1)$	–	$O(W \log D)+$ $O(a_w W)$	II
Bunker-A [83]	a_w	$O(n_w)$	a_w	$O(1)$	$O(a_w)$	$O(1)$	$O(W \log D)$	–	III

$\star$: The complexity also requires n_w *ocall*s (one-way trip) when sending query tokens to the server.

†: We note that the number of update *ocall*s is n_w if the update is addition. Otherwise, deletion updates do not take any *ocall*s.

‡: If there is no deletion updates between two searches on different w, d is cancelled. Then, the complexity is only $O(n_w)$.

work in ORAM powered by SGX [118] demonstrates that SGX can be treated as a delegate of clients, so as to ease the overhead of client storage and computation, and reduce the communication cost between the client and server. Therefore, we first start to investigate the limitations of the existing SGX-supported DSSE schemes that support *forward* and *backward privacy*. Then, we propose new efficient schemes that support these advanced security notions.

5.1.1 Type-II Backward privacy with Bunker-B

We note that Amjad et al. [83] recently proposed three *backward private* SGX-supported schemes: the Type-I scheme Fort, Type-II scheme Bunker-B, and Type-III scheme Bunker-A. The performance and security overview of these schemes can be found in Table 5.1. The table demonstrates the computation and communication cost for update and search among SGX-supported backward-private schemes. As shown, Bunker-B has $O(1)$ update computation complexity and a_w update *ocall*s. However, it causes high computation complexity $O(a_w)$ and involves a large number of roundtrips (i.e., a_w) during the search. Bunker-A does not perform re-encryption and re-insertion after search and thus only achieves Type-III backward privacy. However, it still treats deletion as insertion, just like Bunker-B. Therefore, we only analyse the limitations of Bunker-B as follows.

The Update and Search protocols of Bunker-B are summarily presented in Algorithm 8. As shown, Bunker-B only requires $O(1)$ update computation complexity and a_w update *ocall*s. For each (w, id), Bunker-B lets the enclave follow the same routine to generate tokens for addition and deletion and uses the generated tokens to update M_I on the server (line 5 in Algorithm 8). However, it causes high computation complexity $O(a_w)$ and involves a large number of roundtrips (i.e., a_w) during the search. In the Search protocol, the core idea of Bunker-B is to let the enclave read all records (associated with *add* or *del*) in M_I corresponding to the keyword. Then, the enclave decrypts them and filters deleted *id*s based on the operation. After query, the enclave re-encrypts non-deleted ids and sends the newly generated tokens to the server for updates. These steps are summarised in lines 21-26 in Algorithm 8. We have implemented Bunker-B (see Section 5.5.2) and found that the scheme also has other limitations in practice as follows:

Intensive Ecall/Ocall Usage: Giving a document doc with an identifier id and M unique keywords to the server, Bunker-B repeatedly performs the Update protocol by using M *ecall*s and then the same number of *ocall*s to insert tokens to the index map M_I. It indicates that the number of *ecall/ocall* for Bunker-B is linear to the keyword-document pairs for updates. In practice, a dataset can include a large number of keyword-document pairs ($> 10^7$). As a result, Bunker-B takes 12μs to insert one (w, id) pair, and 2.36×10^7 *ecall/ocalls* to insert 10^6 documents to the database. Similarly, deleting a doc in Bunker-B is the same as the addition, with the exception that the tokens contain $op = del$. Experimentally, Bunker-B takes 1.98×10^8 *ecall*/*ocall*s to delete 2.5×10^5

Algorithm 8 Bunker-B [83]: Update and Search protocols

Update(op, in) : // $op \in \{add, del\},\ in = (w, id)$

1: Client retrieves $st_w = (version, count)$ from st;
2: Send $(w, version, count, op, id)$ to enclave;
3: Client updates $st_w = (version, count + 1)$ to st;
4: Enclave generates an update token $u_{tk} = (u, v)$:
5: $u := F_{K_1}(w||version||count + 1)$
6: $v := Enc(K_2, id||op)$
7: Enclave sends u_{tk} to the server;
8: Server receives $u_{tk} = (u, v)$ from the enclave;
9: Server updates the map $M_I[u] = v$

Search(w) :

1: Client retrieves $st_w = (version, count)$ from st;
2: Client outputs $(w, version, count)$ to enclave st;
3: Client updates $st_w = (version + 1, count)$ to st;
4: Enclave receives $(w, version, count)$ from client;
5: Enclave generates query tokens $q_{tk} = (u_1, \ldots, u_i, \ldots, u_{count})$, where :
6: $u_i := F_{K_1}(w||version||i)$
7: Enclave sends q_{tk} to the server;
8: Server returns to the enclave with the list $L = \{(u_1, v_1), \ldots, (u_c, v_c)\}$;
9: Server deletes all pairs in the L from M_I;
10: Enclave filters non-deleted ids with $R = \{id : \nexists(id, op = del) \in L\}$;
11: Enclave returns R to the client;
12: Enclave resets $count = 1$ and re-encrypts R with
13: **for each** $id \in R$ **do**:
14: Generate a new token with
15: $u := F_{K_1}(w||version + 1||count)$
16: $v := Enc(K_2, id||op = add)$
17: Send (u, v) to the server to update M_I;
18: Enclave increase $count+ = 1$;
19: **end for**

documents. The practical performance of Bunker-B can be found in Section 5.5.2. We also note that Bunker-B only supports deletion updates on the index map M_I without considering deleting real documents [83].

Search Latency: The re-encryption on non-deleted ids per search makes Bunker-B inefficient. In particular, when the number of those ids is large and the deleted ones is a small portion (adding 10^6 documents and deleting 25% documents), Bunker-B takes 3.2s to query a keyword (see Section 5.5.2).

Table 5.2: Comparison with previous SGX-supported Type-I *backward-private* schemes

Type-I Scheme	Communication Enclave-Server			Enclave Computation			Enclave Storage
	Add	Del	Search	Add	Del	Search	
Orion*	$\mathcal{O}(log^2 N)$	$\mathcal{O}(log^2 N)$	$\mathcal{O}(n_w log^2 N)$	$\mathcal{O}(log^2 N)$	$\mathcal{O}(log^2 N)$	$\mathcal{O}(n_w log^2 N)$	$\mathcal{O}(1)$
Fort[83]	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(log^2 N)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)+$ $\mathcal{O}(\Sigma_{\forall w} d_w)$	$\Sigma_{\forall w} d_w$
Maiden	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(WlogD)$ $+\mathcal{O}(a_w W)$ $+\mathcal{O}(N)$

In this table, N denotes the total number of keyword/document pairs. a_w presents the total number of entries of addition updates performed on w. n_w is the number of (current, non-deleted) documents containing w. Let d_w denote the number of deletions performed on w. D and W denote the total number of documents, and the total number of keywords, respectively. Orion* presents the scheme of porting the *Client* in Orion [80] to TEE.

5.1.2 Type-I Backward privacy with Orion* and Fort

In this section, we firstly review the existing attempts on designing a TEE-based Type-I backward-private scheme and indicate why they still fall short under the TEE setting (see Table 5.2).

Orion*: A basic attempt to build a Type-I *backward-private* scheme is to utilise ORAM. In particular, the latest construction (Orion [80]) leverages two oblivious map OMAPs to store the database index and states. These two OMAPs ensure the update and query operations are oblivious, and thus Orion can achieve Type-I backward privacy. Since the existing work demonstrates how to use TEE (Intel SGX) to accelerate the oblivious data structure [118], one can directly employ SGX to fulfil the *Client*'s role in non-TEE supported scheme. Then, the *Server* in the scheme (Orion [80]) can be executed in untrusted memory area outside the *Enclave*. We name this ported scheme as **Orion***. However, this solution still maintains the high communication overhead between the *Enclave* and the *Server* during Update *addition/deletion* and Search due to the use of multiple oblivious maps at the *Server* (see Table I for porting the *Client* of Orion [80] to the *Enclave*).

Fort: The second approach (Fort) was proposed by Amjad et al. [83]. Fort is also the most secure while still relying on ORAM and thus we exclude it in this work due to its overhead. Fort reduces the communication overhead between the *Enclave* and the *Server* via two solutions. First, it asks the *Server* to only maintain one oblivious map OMAP. The map stores the pair $(F(w, id), label)$ during Update, where *label* is the token used to insert (w, id) pair into the index map M_I. Secondly, the *Enclave* in Fort stores a $Stash_{del} = \Sigma_{\forall w} d_w$ that maintains the deleted *label*s d_w of every keyword w. The *Client* in Fort holds keyword state

$st_w = (version, count)$ where $version$ increases after every Search, and $count$ gets updated for every Update op $\in \{add, del\}$ on w. During Update, the *Enclave* generates an update token $(label, value)$, where $label := F_{K_1}(w||version||count)$ and $value := Enc(K_2, id||\mathsf{op})$, to insert into M_I. Whenever the *label* is inserted into M_I, the newly generated pair $(F(w, id), label)$ is obliviously added to the OMAP. If the op of Update is *deletion*, the *Enclave* obliviously retrieves the corresponding *label* from the OMAP and then appends it to $Stash_{del}$. The *Enclave* will execute dummy operations on OMAP to hide whether the Update is for *addition* or *deletion*. The Search operation of Fort is Type-I *backward-private* because the *Enclave* only sends (reveals) n_w currently matching *label*s to the *Server* after locally discarding deleted *label*s found in $Stash_{del}$. The complexity of Fort can be found in Table 5.2.

In update, Fort requires $(a_w + r)$ *ocall*s and $O(log^2 N)$ computation complexity. The search operation of Fort requires a_w roundtrips between the enclave and the server since the enclave needs to retrieve all the labels associated with w before discarding deleted labels retrieved previously in deletion updates.

Amjad et al. [83] acknowledged that the cost of identifying and discarding the deleted *label*s of the query keyword w in $Stash_{del}$ of Fort could slow down the *search latency*. However, that cost was not investigated thoroughly in their work [83]; only a theoretical scheme was proposed. Therefore, we had re-implemented the *Enclave*'s computation of Fort and found that the scanning could take up to 8.02×10^6 ms just to scan 10^4 tokens when $Stash_{del} = 10^7$. That insufficient cost is added to the *search latency* upon Search operation of Fort.

Remarks on Fort's optimisation. Amjad et al. [83] note that Fort can be optimised by replacing the usage of $Stash_{del}$ in the *Enclave* by an OMAP to be stored in the untrusted *Server*. In this way, the *Enclave* does not need to perform the linear scanning of identifying and discarding deleted labels of the query keywords. Instead, the *Enclave* obliviously retrieves them from the OMAP during Search. However, this access will downgrade the security of the scheme. The reason is that it additionally leaks the number of deletions of the query keyword during Search, i.e., the number of ORAM accesses can be exposed.

5.2 System Overview

We present the high-level overview of our proposed schemes, as shown in Fig 5.1. The design involves three entities: the client (who is the data owner and therefore trusted), the untrusted server, and the trusted SGX enclave within the server. The system flow involves 9 steps.

At step 1, the client uses the SGX attestation feature to authenticate the enclave and establish a secure channel with the enclave. The client then provisions a secret key K to the enclave through this channel. This completes the Setup protocol of our proposed protocol. Note that this operation does not deploy any

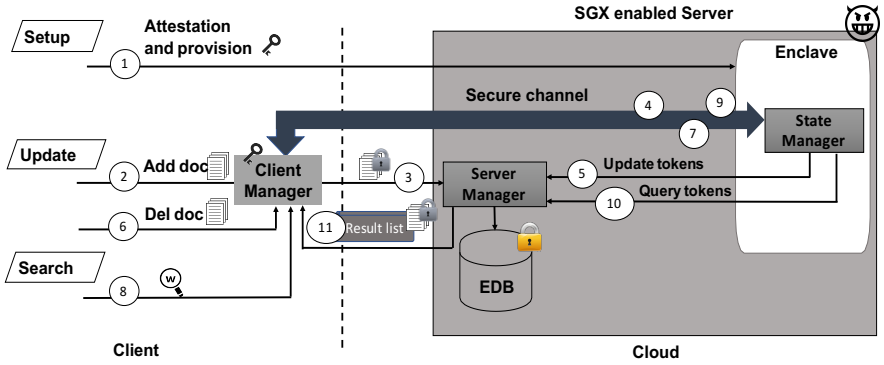


Figure 5.1: High level design

EDB to the server as in DSSE schemes [82]. Instead, we consider that the client outsources documents to the server via Update operations later.

At step 2, giving a document with a unique identifier *id*, the *Client Manager* encrypts the document with the key K and sends the encrypted version of the document to the *Server Manager* (see step 3). The encrypted version with its *id* is then inserted to EDB. After that, the *Client Manager* sends the original document to the *State Manager* located in the enclave via the secure channel (see step 4). At this step, the *State Manager* performs cryptographic operations to generate update tokens that will be sent to *Server Manager* (see step 5). The tokens are used to update the encrypted index of dynamic SE located in the *Server Manager*. Note that traditional DSSE schemes [70, 82, 81] often consider EDB as the underlying encrypted index of DSSE, and omit the data structure storing encrypted documents. Here, we locate them separately to avoid that ambiguity, i.e., the index of DSSE M_I is located in *Server Manager*, and encrypted documents reside in EDB as an encrypted document repository, respectively. To delete a document with a given *id* (step 6), the *Client Manager* directly sends the document *id* to the *State Manager* (see step 7).

At step 8, the client wants to search documents matching a given query keyword w. The *Client Manager* will send the keyword w to the *State Manager* (see step 9). Then, the *State Manager* computes query tokens and excludes the tokens for deleted documents according to the deletion information from step 6. Later, the *State Manager* sends them to the *Server Manager* (in step 10). The *Server Manager* will search over the received tokens and return the list of encrypted matching documents back to the *Client Manager*. At that stage, the encrypted documents are decrypted with K.

5.3 Assumptions and Threat Models

Our Assumptions with Intel SGX: We assume that SGX behaves correctly, (i.e., there are no hardware bugs or backdoors), and the preset code and data

inside the enclave are protected. Also, the communication between the client and the enclave relies on the secure channel created during SGX attestation. Like many other SGX applications [128, 118], side-channel attacks [129, 130, 131] against SGX are out of our scope. Denial-of-service (DoS) attacks are also out of our focus, i.e., the enclave is always available whenever the client invokes or queries. Finally, we assume that all the used cryptographic primitives and libraries of SGX are trusted.

Threat Models: Like existing work [119, 83], we consider a semi-honest but powerful attacker at the server-side. Although the attacker will not deviate from the protocol, he/she can gain full access over software stack outside of the enclave, OS and hypervisor, as well as hardware components in the server except for the processor package. In particular, the attacker can observe memory addresses and (encrypted) data on the memory bus, in memory, or in EDB to generate data access patterns. Additionally, the attacker can log the time when these memory manipulations happen. The goal of the attacker is to learn extra information about the encrypted database from the leakage both revealed by hardware and the leakage function defined in sections 5.4.3 and 5.6.3.

5.4 Design for SGX-supported Type-II

Technical Highlights: Motivated by the limitations of Bunker-B, we design SGX-SE1 and SGX-SE2 that are Type-II backward private schemes with: (1) reduced number of *ecall/ocall* when the client wants to add/delete a document, (2) reduced search roundtrips, and (3) accelerated enclave's computation in search.

We achieve (1) by allowing the client to transfer the document to the enclave for document addition, instead of transferring (w, id) pairs. This design reduces the number of *ecall*s to 1. We then use the enclave to store the latest states ST of all keywords, where the state of a keyword w is $ST[w] = count$. As a result, the enclave is able to generate addition tokens based on ST. Our experiments (see Sections 5.5.2 and 5.5.3) show that this design improves $2\times$ the addition throughput compared to Bunker-B. We note that it is negligible to store ST in the enclave since it costs less than 6 MB to store the states of all keywords in the American dictionary of English[2] (assuming each keyword state item can take up 18 bytes in a dictionary map). Additionally, our scheme only requires 1 *ecall* if the client deletes a document, by transferring that document id to the enclave.

W.r.t. (2), the SGX-SE1 scheme reduces the search roundtrips between the enclave and the server to $(d + d_w)$. The basic idea behind SGX-SE1 is to let the enclave cache the mapping between w and the deleted document ids. In particular, the enclave loads and decrypts d deleted documents to extract the mapping (w, id). It cleans the memory after loading each deleted document to

[2]The dictionary contains about 300,000 common and obsolete keywords

Algorithm 9 The setup protocol in SGX-SE1

$\mathsf{Setup}(1^\lambda)$

Client

1: $k_\Sigma, k_f \xleftarrow{\$} \{0,1\}^\lambda$
2: Launch a remote attestation and establish a secure channel
3: Send $K = (k_\Sigma, k_f)$ to *Enclave*

Enclave

1: Initialise maps ST and D, and a list d
2: Initialise tuples T_1 and T_2
3: Receive $K = (k_\Sigma, k_f)$

Server

1: Initialise maps M_I and M_c and a repository R

avoid the memory bottleneck. After that, the enclave needs d_w roundtrips to retrieve the *counter*s when the enclave filters those deleted *id*s. SGX-SE2 is more optimal by requiring only d_w roundtrips without the need for loading d deleted documents. To do this, SGX-SE2 uses a Bloom filter BF to store the mapping (w, id) within the enclave. Note that the BF can track 1.18×10^7 (w, id) pairs with the storage cost of 34 MB enclave memory[3] with the false positive probability $P_e = 10^{-4}$. Our experiments (see Sections 5.5.2 and 5.5.3) show that the search latency of SGX-SE1 is 30% faster than Bunker-B after inserting 10^6 documents and caching 2.5×10^5 deleted documents. Moreover, SGX-SE2 is 2× faster than Bunker-B for the query after deleting 25% documents.

W.r.t. (3), the proposed SGX-SE1 scheme improves the search computation complexity to $O(n_w + d)$. We note that the complexity is even amortised if there is no deletion updates between a sequence of queries. The reason is that the enclave only loads d document for the first query to update the mapping of all keywords in ST with the deleted documents. Furthermore, the search computation complexity of SGX-SE2 is only $O(n_w)$. We note that testing the membership of d documents in the BF is v_d where v is the vector of BF. Our experiments (see Sections 5.5.2 and 5.5.3) show that Bunker-B takes 3.2s for queries after inserting 10^6 documents and deleting 25% documents while SGX-SE1 only takes 2.4s after caching those deleted documents. In addition, SGX-SE2 spends the least time 1.4s, i.e., 2× faster than Bunker-B.

5.4.1 SGX-SE1

The basic idea behind SGX-SE1 is to let the enclave store the latest states ST of keywords and keeps the list d of deleted document *id*s, in order to facilitate searches. Then, the enclave only loads the deleted documents for the first search

[3] 1.18×10^7 pairs $\approx 386\times$ `Hamlet` tragedy written by William Shakespeare

Algorithm 10 The update protocol in SGX-SE1

Update(op,in)

Client

1: **if** op $= add$ **then**
2: $f \leftarrow \mathsf{Enc}(k_f,\mathsf{doc})$
3: send send (id, f) to *Server*
4: **end if**
5: send (op,id) to *Enclave*

Enclave

1: **if** op $= add$ **then**
2: $f \leftarrow R[id]$
3: $\{(w, id)\} \leftarrow Parse(\mathsf{Dec}(k_f, f))$
4: **for each** (w, id) **do** $k_w \parallel k_c \leftarrow F(k_\Sigma, w)$ $c \leftarrow ST[w]$
5: **if** $c = \perp$ **then**
6: $c = -1$
7: **end if** $c \leftarrow c + 1$
8: $k_{id} \leftarrow H_1(k_w, c)$
9: $(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id)$
10: add (u, v) to T_1
11: $(u', v') \leftarrow (H_3(k_w, id), \mathsf{Enc}(k_c, c)$
12: add (u', v') to T_2
13: $ST[w] \leftarrow c$
14: **end for**
15: send (T_1, T_2) to *Server*
16: reset T_1 and T_2
17: **else**
18: add id to d
19: **end if**

Server

1: // if op $= add$
2: receive (id, f) from *Client*
3: $R[id] \leftarrow f$
4: receive (T_1,T_2) from *Enclave*
5: **for each** (u, v) **in** T_1 **do**
6: $M_I[u] \leftarrow v$
7: **end for**
8: **for each** (u', v') **in** T_2 **do**
9: $M_c[u'] \leftarrow v'$
10: **end for**
11: // if op $= del$ then do nothing

Algorithm 11 The search protocol in SGX-SE1

Search(w)

Client

1: send w to *Enclave*

Enclave

1: $st_{w_c} \leftarrow \{\emptyset\}, Q_w \leftarrow \{\emptyset\}$;
2: $k_w \parallel k_c \leftarrow F(k_\Sigma, w)$
3: **for each** id_i **in** d **do**
4: $f_i \leftarrow R[id_i]$; $\mathsf{doc}_i \leftarrow \mathsf{Dec}(k_f, f_i)$
5: **if** w **in** doc_i **then**
6: $D[w] \leftarrow id_i \cup D[w]$
7: delete $R[id_i]$
8: **end if**
9: **end for**
10: **for each** id **in** $D[w]$ **do**
11: $u' \leftarrow H_3(k_w, id)$
12: $v' \leftarrow M_c[u']$
13: $c \leftarrow \mathsf{Dec}(k_c, v')$
14: $st_{w_c} \leftarrow \{c\} \cup st_{w_c}$
15: delete $M_c[u']$
16: **end for**
17: $st_{w_c} \leftarrow \{0, \ldots, ST[w]\} \setminus st_{w_c}$
18: **for each** c **in** st_{w_c} **do**
19: $u \leftarrow H_2(k_w, c)$
20: $k_{id} \leftarrow H_1(k_w, c)$
21: $Q_w \leftarrow \{(u, k_{id})\} \cup Q_w$
22: **end for**
23: send Q_w to *Server*
24: delete $D[w]$

Server

1: receive Q_w from *Enclave*
2: $Res \leftarrow \emptyset$
3: **for each** (u_i, k_{id_i}) **in** Q_w **do**
4: $id_i \leftarrow \mathsf{Dec}(k_{id_i}, M_I[u_i])$
5: $\mathsf{doc}_i \leftarrow R[id_i]$
6: add doc_i to Res
7: **end for**
8: send Res to *Client*

between two deletion updates to update the mapping between deleted *id*s and tracked keywords. Subsequent searches between the two deletion updates do not

require loading the deleted documents again. We note that the enclave clearly needs to remove d after retrieving them in the first query to save the enclave's storage. Once the enclave knows the mapping between the query keyword and deleted documents, it infers the mapping of the query keyword with the rest non-deleted documents, in order to generate query tokens. After that, the server retrieves documents based on the received tokens and returns the document result list to the client. We explain the protocols further as follows:

In setup (see Algorithm 9, client communicates with enclave upon an established secure channel to provision $K = (k_\Sigma, k_f)$ where k_Σ enables enclave to generate update/query tokens and k_f is the symmetric key for document encryption/decryption. The enclave maintains the maps ST and D, and the list d, where ST stores the states of keywords, D presents the mapping between keywords and deleted documents, and d is the array of deleted ids. The server holds an encrypted index M_I, the map of encrypted state M_c, and the repository R with $R[id]$ stores the encrypted document of document identifier id.

In update (see Algorithm 10), the client receives a tuple $(\mathsf{op}, \mathsf{in})$, where it could be $(\mathsf{op} = add, \mathsf{in} = (\mathsf{doc}, id))$ or $(\mathsf{op} = del, \mathsf{in} = id)$. If the update is addition, the client encrypts doc by using k_f and sends that encrypted document to server. After that, the client sends $(\mathsf{op}, \mathsf{in})$ to the enclave. The enclave will then parse doc to retrieve the list L of $\{(w, id)\}$. For each w, the enclave generates k_w and k_c from k_Σ, and retrieves the latest state $c \leftarrow ST[w]$. The enclave will then generate k_{id} from c by using $H_1(k_w, c)$ with H_1 is a hash function. After that, the enclave uses k_w, k_c, and k_{id} to generate encrypted entries (u, v) and (u', v') for w. In particular, the first encrypted entry, with $(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$, holds the mapping between c and id to allows the server retrieves id based on given u and k_{id}. The second encrypted entry, with $(u', v') \leftarrow (H_3(k_w, id), \mathsf{Enc}(k_c, c))$, hides the state c of documents. In this way, the client can retrieve the state c of deleted documents upon sending u' in search operation. In our protocols, H_1 and H_2 are hash functions, and Enc is a symmetric encryption cipher. We note that enclave only sends a batch of (T_1, T_2) to the server within one *ocall* per a document addition, where $T1 = \{(u_{w_1}, v_{w_1}), \ldots, (u_{w_{|L|}}, v_{w_{|L|}})\}$ and $T2 = \{(u'_{w_1}, v'_{w_1}), \ldots, (u'_{w_{|L|}}, v'_{w_{|L|}})\}\}$. Then, the server will update T_1 and T_2 to M_I and M_c, respectively. If the update is deletion, the enclave simply updates d by the deleted id without further computation or communication to the server.

In search (see Algorithm 11), the client sends a query q containing w to the enclave via the secure channel and expects to receive all the *current* (non-deleted) documents matching w from the server. The enclave begins loading deleted encrypted documents in d from the server in a sequential manner. By using k_f, the enclave decrypts those documents for checking the existence of w, and updating $D[w]$ if applicable. By leveraging $D[w]$, the enclave can retrieve the state list $st_{w_c} = \{c_{id}^{del}\}$, where c_{id}^{del} is the state used when the enclave added the deleted document id for w. After that, the enclave simply infers the states of non-deleted

documents by excluding st_{w_c} from the set of $\{0, \ldots, ST[w]\}$. Finally, the enclave will compute the query token u and k_{id} for these non-deleted documents, and send the list $Q_w = \{(u, k_{id})\}$ to the server. At the server, upon receiving Q_w, it can retrieve id_i when decrypting $M_I[u_i]$ with k_{id_i}. Finally, the server returns the encrypted documents $Res = \{R[id_i]\}$ to the client.

Efficiency of SGX-SE1: In update, SGX-SE1 only takes n_w *ocalls* to add all n documents containing w to the server, and no *ocall* for deletion due to the caching of deleted documents within the enclave. That efficiency outperforms Bunker-B since the latter requires an additional *ocall* per a deletion. However, we note that the asymptotic performance of SGX-SE1 is affected by $(d + d_w)$ search roundtrips. In particular, the enclave needs to load and decrypt deleted documents within the enclave. Thus, the search performance really depends on how large the number of deleted documents is at the query time. We will later compare our search latency with Bunker-B in Sections 5.5.2 and 5.5.3.

5.4.2 SGX-SE2

We see that SGX-SE1 has $(d + d_w)$ search roundtrips and non-trivial $O(n_w + d)$ computation. One downside is that the enclave needs to spend time on decrypting deleted documents. Here, we present SGX-SE2, an advanced version of SGX-SE1, that reduces search roundtrips to d_w and achieves better asymptotic and concrete search time $O(n_w + v_d)$. The main solution we make to SGX-SE2 is that we use a Bloom filter BF within the enclave to verify the mapping between query keyword w and deleted document ids. In this way, SGX-SE2 avoids loading them from the server. Since BF is a probabilistic data structure, we can configure it to achieve a negligible false positive rate P_e (see Section 5.5.3). We summarily introduce SGX-SE2 in Algorithm 12 as follows:

In setup, SGX-SE2 is almost the same with that one in SGX-SE1 with the exception that the client also requires to initialise the parameters of BF. They are, k_{BF}, b and h, where k_{BF} is the key for computing the hashed value of $(w||id)$, and b is the number of bits in the BF vector (i.e, vector size), and h is the number of hash functions. Upon receiving the BF setting, the enclave initialises the BF vector and the set of hash functions $\{H'_j\}_{j\in[h]}$. In SGX-SE2, the mapping D between keywords and deleted ids is no longer needed within the enclave like that one in SGX-SE1.

In update, SGX-SE2 is also similar with SGX-SE1. However, if the update is addition, the enclave computes a new member $H'_j(k_{BF}, w \parallel id)$ to update BF.

In search, SGX-SE2 verifies the mapping between query keyword w and deleted ids by checking the membership of $(w||id)$ with BF. If the mapping is valid, SGX-SE2 performs the same as SGX-SE1 to retrieve the state list $st_{w_c} = \{c_{id}^{del}\}$, where c_{id}^{del} is the state used for deleted ids. After that, the enclave infers the states of non-deleted documents and computes query tokens to send to the server.

Efficiency of SGX-SE2: The scheme clearly outperforms SGX-SE1 in terms of

Algorithm 12 The protocols in SGX-SE2

Setup(1^λ)

1: Performs the same Setup in SGX-SE1
2: Client inits $k_{BF} \xleftarrow{\$} \{0,1\}^\lambda$
3: Client sets integers b, h and provisions (k_{BF}, b, h) to $Enclave$
4: Enclave selects $\{H'_j\}_{j\in[h]}$ for BF
5: Enclave does not maintain D

Update(op,in)

1: Performs the same Update in SGX-SE1
2: **if** op $= add$ **then**
3: **for each**(w, id) **do**
4: **for** $j = 1 : h$ **do**
5: $h'_j(w, id) \triangleq H'_j(k_{BF}, w \parallel id)$
6: $BF[h'_j(w, id)] \leftarrow 1$
7: **end for**
8: **end for**
9: **end if**

Search(w)

1: Replacing lines 3-16 in Search in SGX-SE1 with:
2: **for** id**in**d **do**
3: **if** $BF[H'_j(k_{BF}, w \parallel id)]_{j\in[h]} = 1$ **then**
4: $u' \leftarrow H_3(k_w, id)$
5: $v' \leftarrow M_c[u']$
6: $c_{id}^{del} \leftarrow \mathsf{Dec}(k_c, v')$
7: $st_{w_c} \leftarrow \{c_{id}^{del}\} \cup st_{w_c}$
8: delete $M_c[u']$ and delete $R[id]$
9: **end if**
10: **end for**

search computation and communication roundtrips due to the usage of the Bloom filter. It avoids loading d deleted documents into the enclave, making the search roundtrip only d_w. The scheme is even more efficient when $|d|$ is large. The reason is that the cost of verifying a membership $(w||id)$ is always $O(1)$ under the fixed BF setting. We note that checking d members in the BF is still more efficient than loading/decrypting their real documents. BF is also memory-efficiently; therefore, one can configure its size to balance the enclave memory with the demand of large datasets.

Remark: Note that deleting a document doc with identifier id in Bunker-B requires deletion entries of all keywords in that doc with $(w_i, id, \mathsf{op} = del)$ have been inserted in the encrypted index M_I beforehand. That would require M

*ocall*s for the doc of M keywords. Then, Bunker-B takes extra one *ocall* to physically delete the doc. This physical deletion cost is the same with SGX-SE1 and SGX-SE2 (i.e., one *ocall*) except that these two schemes do not require any deletion entries to be inserted in M_I. Clearly, Bunker-B, SGX-SE1, and SGX-SE2 can do batch processing to delete d documents in one *ocall*. With SGX-SE1, deleting a doc can be done right after all keywords in the deleted document have been cached in $D[w]$. With SGX-SE2, a doc can be deleted at the earliest time when any keyword in the doc is being searched.

5.4.3 Security Analysis

The only difference between SGX-SE1 and SGX-SE2 in term of security is that SGX-SE1 requires to load encrypted deleted documents to the enclave during the search. Therefore, our following analysis is almost identical to both schemes. We will state the difference between them wherever is necessary.

We denote $\mathcal{D}$ as our general scheme that could be SGX-SE1 or SGX-SE2. The security of $\mathcal{D}$ can be quantified via a stateful leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch}, \mathcal{L}^{hw})$. The first three components define the information exposed in Setup, Update, and Search, respectively. The latter one, $\mathcal{L}^{hw}$, defines the inherent leakage of the used SGX enclave with the outputs from the enclave to the server. We now define $\mathcal{L}$ and then formalise our security with analysis.

In Setup, $\mathcal{D}$ leaks nothing to the server except the data structure of M_I (i.e., the encrypted index), M_c (i.e., the encrypted map of keyword states), R (i.e., the empty repository of encrypted documents).

In Update($\mathsf{op} = add, \mathsf{in}$), $\mathcal{D}$ leaks the data access pattern of encrypted entries to be inserted in M_I, M_c, and R. Otherwise, if $\mathsf{op} = del$, $\mathcal{D}$ leaks nothing under the secure channel established in Setup. Hence,

$$\mathcal{L}^{Updt}(\{(\mathsf{op}, \mathsf{in})\}) = \{(T_1, T_2, R[id_i])\}$$

where $T_1 = \{(u, v)\}$ and $T_2 = \{(u', v')\}$ present the collections of entries to be inserted in M_I and M_c respectively, and $R[id_i]$ denotes an encrypted document to be inserted in R with label id_i.

In Search(w), $\mathcal{D}$ leaks 1) the access pattern on M_c when the enclave queries the deleted states of w, named $\mathsf{ap}_{M_c}(w)$, 2) the access pattern on M_c when the enclave queries non-deleted *id*s, named $\mathsf{ap}_{M_I}(w)$, if $\mathcal{D}$ is SGX-SE1, and 3) the pattern on deleted documents d_w, named $\mathsf{ap}_R(d_w)$. Then, formally

$$\mathcal{L}^{Srch}(w) = \mathsf{ap}_{M_c}(w) + \mathsf{ap}_{M_I}(w) + [\mathsf{ap}_R(d_w)]$$

We define $\mathcal{L}^{hw}(M_I, M_c, R)$ as the hardware leakage during Update and Search. That includes memory access and location, the time log, and the size of the

manipulated memory area.

$$\mathcal{L}^{hw}(M_I, M_c, R) = (M_I, M_c, R)^{Updt} + (M_I, M_c, R)^{Srch}$$

This function outputs the trace τ of (l, T, v, t), where l is the label input, T is a map data structure that could be M_I, M_c, and R, v is the value at $T[l]$, and t is the time access of op. W.r.t. SGX-SE1, if l is an id, the function will output the encrypted document e and the document size $|e|$.

Definition 15. *Let $\mathcal{D}$ denote our scheme that consists of three protocols Setup, Update, and Search. Consider the probabilistic experiments $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, whereas $\mathcal{A}$ is a stateful adversary, and $\mathcal{S}$ is a stateful simulator that gets the leakage function $\mathcal{L}$.*

$\boldsymbol{Real}_{\mathcal{A}}(\lambda)$: The challenger runs $\mathsf{Setup}(1^{\lambda})$ that involves the client, the enclave, and the server to initialise necessary data structures. $\mathcal{A}$ chooses a database $\mathsf{DB} = \{\mathsf{doc}_i\}_{i \in Z}$ and makes a polynomial number of updates (addition/deletion) with $(\mathsf{op}, \mathsf{in})$, where Z is a natural number of documents, and $(\mathsf{op} = add, in = \mathsf{doc}_i)$ or $(\mathsf{op} = del, \mathsf{in} = id_i)$. Accordingly, the challenger runs those updates with $\mathsf{Update}(\mathsf{op}, \mathsf{in})$ and eventually returns the tuple $(M_I, M_c, R)^{Updt}$ to $\mathcal{A}$. After that, $\mathcal{A}$ adaptively chooses the keyword w (resp., $(\mathsf{op}, \mathsf{in})$) to search (resp., update). In response, the challenger runs $\mathsf{Search}(w)$ (resp., $\mathsf{Update}(\mathsf{op},\mathsf{in})$) and returns the transcript of each operation. The challenger also returns $(M_I, M_c, R)^{Srch}$ to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b.

$\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ chooses a $\mathsf{DB} = \{\mathsf{doc}_i\}_{i \in Z}$. By using $\mathcal{L}^{Updt}$ and $(M_I, M_c, R)^{Updt}$, $\mathcal{S}$ creates a tuple of (M_I, M_c, R) and passes it to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses the keyword w (resp., $(\mathsf{op}, \mathsf{in})$) to search (resp., update). The challenger returns the transcript simulated by $\mathcal{S}(\mathcal{L}^{Srch}(w))$ (resp., $\mathcal{S}(\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}))$) with $(M_I, M_c, R)^{Srch}$. Finally, $\mathcal{A}$ returns a bit b.

We say $\mathcal{D}$ is $\mathcal{L}$-secure against adaptive chosen-keyword attacks if for all probabilistic polynomial-time algorithms $\mathcal{A}$, there exist a PPT simulator $\mathcal{S}$ such that

$$|Pr[\boldsymbol{Real}_{\mathcal{A}}(\lambda) = 1] - Pr[\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda)$$

Theorem 3. *The scheme $\mathcal{D}$ presented above is $\mathcal{L}$-secure according to Def 16.*

We now prove Theorem 4 by describing a PPT simulator $\mathcal{S}$ for which a PPT adversary $\mathcal{A}$ can distinguish $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.

Proof. $\mathcal{S}$ first generates a random key $\tilde{K} = (\tilde{k}_{\Sigma}, \tilde{k}_f)$ to simulate the key components that the enclave contains. Then, $\mathcal{A}$ executes $\mathsf{Search}(\mathrm{w})$ with w, which is a random keyword, in order to obtain a query token q sent by the enclave. Then, $\mathcal{A}$ simulates addition tokens a for w based on $\tilde{K}$ and $\mathcal{L}^{hw}(M_I, M_c, R)$, and sends them to the enclave to receive the new update of (M_I, M_c, R). However, $\mathcal{A}$ cannot map which update token in a relates to q. The reason is that the enclave keeps

Table 5.3: Statistics of the datasets used in the evaluation.

Name	# of keywords	# of docs	# of keyword-doc pairs
Synthesis	$1,000$	$1,000,000$	$11,879,100$
Enron	$29,627$	$517,401$	$37,219,800$

increasing the state $ST[w]$. Hence, $\mathcal{A}$ cannot distinguish between the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and the simulated output in Update and Search (*forward privacy*).

During Search, if there were delete updates made in the past on deleted documents d with identifier list $\{id_i\}$, $\mathcal{A}$ cannot know which keywords are inside the encrypted doc $R[id_i]$. Also, $\mathcal{A}$ does not know when delete updates made since the enclave only requests d during Search. The $\mathsf{ap}_{M_c}(w)$ does not reveal id_i (see Search. However, $\mathcal{A}$ knows the time when the entry relating id_i added to ap_{M_c} via $\mathcal{L}^{hw}$, and how many id_i in d. Clearly, at the end of the protocol $\mathcal{A}$ knows how many current (non-deleted) id accessed. Hence, $\mathcal{D}$ is type-II *backward privacy*. □

5.5 Evaluation of Type-II Backward privacy

In this section, we summarise the implementation of SGX-SE1 and SGX-SE2 as well as the theoritically proposed Bunker-B. Then, we investigate the performance of those schemes.

5.5.1 Experiment Setup and Implementation

We build the prototype of SGX-SE1 and SGX-SE2 using C++ and the Intel SGX SDK. In addition, we implement the prototype of Bunker-B as the baseline for comparisons, since its implementation is not publicly available. The prototype leverages the built-in cryptographic primitives in the SGX SDK to support the required cryptographic operations. It also uses the settings and APIs from the SDK to create, manage and access the application (enclave) designed for SGX. Recall that the SGX can only handle 96 MB memory within the enclave. Access to the extra memory space triggers the paging mechanism of the SGX, which brings an extra cost to the system (average 5× as reported in [132]). To avoid paging in our prototype, our prototypes are implemented with batch processing to tackle with the keyword-document pairs, which splits a huge memory demand into multiple batches with smaller resource requests. The batch processing enables our prototypes to handle queries with large memory demands. On the other hand, the prototype should avoid too many *ecall*s/*ocall*s as it incurs the I/O communication cost between the untrusted and the trusted application (enclave).

For evaluation, we choose two datasets: One is a synthesis dataset (3.2 GB) generated from the English keyword frequency data based on the Zipf's law

Table 5.4: Average time (μs) for adding a keyword-doc pair under different schemes.

# of docs	# of keyword-doc pairs	BunkerB	SGX-SE1	SGX-SE2
2.5×10^5	2.5×10^5	21	23	26
5×10^5	6.5×10^5	19	19	21
7.5×10^5	1.9×10^6	15	12	14
1×10^6	1.18×10^7	12	7	8

Table 5.5: Number of *ecall*/*ocall* for adding 1×10^6 documents for different schemes.

# of calls	BunkerB	SGX1	SGX2
ecall	1.18×10^7	1×10^6	1×10^6
ocall	1.18×10^7	1×10^6	1×10^6

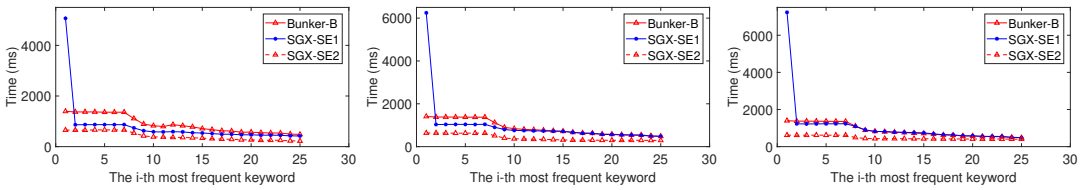
distribution, and the other one is the Enron email dataset (1.4 GB). A summary of the statistical features of the datasets is given in Table 5.3.

The prototypes are deployed in a workstation equipped with SGX-enabled CPU (Intel Core i7-8850H 2.6 GHz) and 32 GB RAM.

5.5.2 Performance evaluation on synthesis dataset

Insertion and deletion: First, we evaluate the time for insertion and deletion under three different schemes. In this evaluation, we follow a reversed Zipf's law distribution to generate the encrypted database of our synthesis dataset, and we measure the runtime for adding one keyword-document pair into the encrypted database of different schemes. As shown in Table 5.4, Bunker-B takes 21 μs to insert one pair, which is faster than our schemes (23 μs and 26 μs) when the number of keyword-document pairs equals the number of documents. The reason is that the insertion time of the above three schemes is bounded by the I/O (*ecall*/*ocall*) between the untrusted application and the enclave. For Bunker-B, the I/O cost is linear to the number of keyword-document pairs, while the one for our schemes is linear to the number of documents. Also, our schemes involve more computations (PRF, Hash) and maintain more data structures (Bloom filter), which require more time to be processed. Nonetheless, when inserting 1×10^6 documents, our schemes only require 7 μs and 8 μs respectively to insert one keyword-document pair, which is 2× faster than Bunker-B (12 μs). In the above case, the number of keyword-document pairs is 10× larger than the number of documents, which implies that Bunker-B needs 10× more I/O operations (*ecall*/*ocall*) to insert the whole dataset comparing to our schemes (see Table 5.5 for details). Note that the real-world document typically consists of more than one keyword. Hence, our schemes are more efficient than Bunker-B when dealing with a real-world dataset (see Section 5.5.3).

For deletion, the performance of Bunker-B is identical to that for insertion



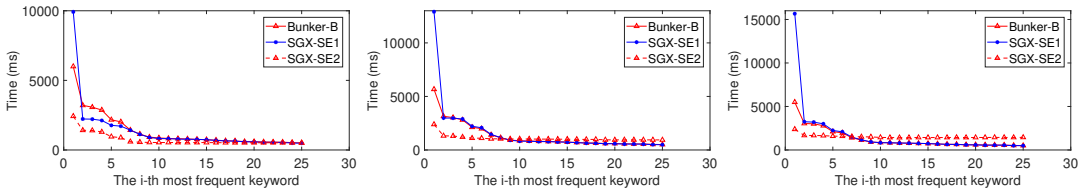
(a) 25% deletion (b) 50% deletion (c) 75% deletion

Figure 5.2: The query delay of querying the i-th most frequent keyword in the synthesis dataset under different schemes (insert 2.5×10^5 documents and delete a portion of them).

(12 μs), because deletion runs the same algorithm with different operations. For our schemes, the deletion process only inserts the document *id* into a list, and the deletion operation is executed by excluding the deleted *id* during the query phase. Thus, our schemes only need 4 μs to process one document in the deletion phase.

Query delay: Next, we report the query delay comparison between Bunker-B and our schemes to show the advantage of using SGX-SE1 and SGX-SE2. To measure the query delay introduced by keyword frequency and the deletion operation, we choose to query the top-25 keywords after deleting a portion of documents. In our first evaluation, we insert 2.5×10^5 documents and delete 25%, 50% and 75% of the documents, respectively. Fig. 5.2 illustrates the query delays when deleting 25% of documents: For the most frequent keyword, Bunker-B needs 1.3 s to query while SGX-SE2 only needs 654 ms. Although SGX-SE1 takes 5 s to perform the first search, it also caches the deleted keyword-document pairs inside the enclave and performs deletion on documents during the first query. As a result, the rest of the queries are much faster, as the number of *ocall*s is significantly reduced (900 μs if we query the most frequent keyword again). Even for the 25-th most frequent keyword, SGX-SE1 (159 ms) and SGX-SE2 (155 ms) are still 40% faster than Bunker-B (221 ms). Bunker-B is always slower than SGX-SE1 and SGX-SE2 in the above case as it requires to re-encrypt the remaining 75% documents after each query. Compared to Bunker-B, SGX-SE1 and SGX-SE2 only access the deleted 25% files and exclude the corresponding token of deleted files before sending the token list. With the increase of the deletion portion, the difference of the query delay between our schemes and Bunker-B becomes smaller as Bunker-B has fewer documents to be re-encrypted after queries. When 75% of the documents are deleted, our schemes still outperform Bunker-B when querying the keywords with a higher occurrence rate (see Fig. 5.2(c)). However, their performances are almost the same when querying the 25-th most frequent keyword, i.e., about 400 ms for three schemes, because Bunker-B only re-encrypts a tiny amount of document id (almost 0).

The second evaluation shows the query delay when inserting all 1×10^6



(a) 25% deletion (b) 50% deletion (c) 75% deletion

Figure 5.3: The query delay of querying the i-th most frequent keyword in the synthesis dataset under different schemes (insert 1×10^6 documents and delete a portion of them).

Table 5.6: Number of *ecall*/*ocall* for deleting a portion of documents after adding 1×10^6 documents.

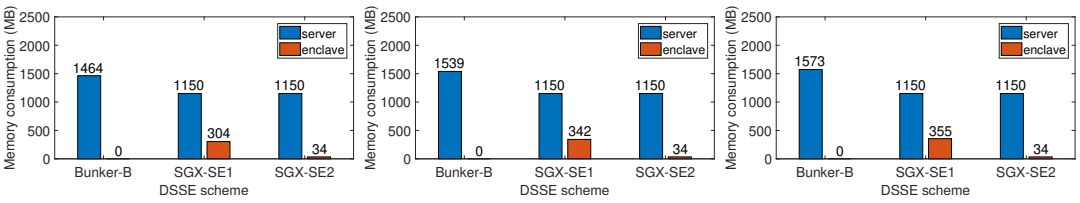
Deletion %	BunkerB		SGX1		SGX2	
	ecall	*ocall*	*ecall*	*ocall*	*ecall*	*ocall*
25%	9.9×10^6	9.9×10^6	2.5×10^5	0	2.5×10^5	0
50%	1.12×10^7	1.1×10^7	5×10^5	0	5×10^5	0
75%	1.16×10^7	1.16×10^7	7.5×10^5	0	7.5×10^5	0

Table 5.7: Number of *ecall*/*ocall* when querying the most frequent keyword after adding 1×10^6 documents and deleting a portion of them.

Deletion %	BunkerB		SGX1		SGX2	
	ecall	*ocall*	*ecall*	*ocall*	*ecall*	*ocall*
25%	1	21	1	$250,011\star/11$	1	11
50%	1	20	1	$500,010\star/10$	1	10
75%	1	21	1	$750,011\star/11$	1	11

$\star$: It includes the *ocall* for caching and deleting the encrypted documents.

documents into the encrypted database. The major difference between this experiment and the previous one is that the SGX-SE1 scheme requires more than 128 MB to cache the deleted documents, which triggers paging. As shown in Fig. 5.4, SGX-SE1 needs 10 s to cache the deleted documents. When processing the query that contains a large number of documents (e.g., the second most frequent keyword), SGX-SE1 (2.4 s) is almost 2× slower than SGX-SE2 (1.4 s). Nonetheless, their query performance is still better than Bunker-B, which takes 3.2 s to answer the above query. When our schemes delete a larger portion of documents (see Fig 5.3(b) and Fig.5.3(c)), the query delay of SGX-SE1 and SGX-SE2 is very close, since SGX-SE1 only refers to the small deletion information cached in the enclave while SGX-SE2 requires to check the Bloom filter for each deleted document.



(a) 25% deletion (b) 50% deletion (c) 75% deletion

Figure 5.4: The enclave's memory after inserting 1×10^6 documents and deleting a portion of them).

Communication cost: The next evaluation demonstrates the impact of I/O operation (*ecall*/*ocall*) on the performance of different schemes. As shown in Table 5.5, Bunker-B needs 10× more *ecall*/*ocall* operations than our schemes. Consequently, although both Bunker-B and our schemes generate and store the encrypted keyword-document pairs at the end, our schemes can achieve a better performance for insertion, because our schemes rely on less I/O operations. This result is consistent with the average insertion time reported in the insertion and deletion part.

In terms of the deletion operation, Bunker-B needs almost 30× more I/O operation than ours (see Table 5.6). Moreover, the deletion in our schemes only requires to insert the deleted id, which does not involve any cryptographic operation, whereas Bunker-B executes the same procedure as insertion. This indicates that our schemes also have less communication cost than Bunker-B.

We further present the number of *ecall*/*ocall* involved during the query process in Table 5.7. Note that we implement batch processing for all schemes, so each *ocall* can process 10^5 query tokens at the same time. The result shows that Bunker-B has more *ocall* during the query process because it needs to issue tokens to query all document id as well as the deleted document. After that, it should issue additional tokens to re-encrypt the undeleted documents. On the other hand, our schemes keep the state map within the enclave, which indicates that our schemes do not require to retrieve all the document id via *ocall*. In most of the case, Bunker-B has 2× more I/O operations than our schemes except for the cache stage of SGX-SE1. Despite the fact that SGX-SE1 takes more than 10^5 *ocall*s to perform caching, we stress that this is a one-time cost; it also enables our scheme to remove the document physically, whereas Bunker-B only can delete the document from the encrypted index.

Memory consumption: Finally, we present the memory consumption of three different schemes. Since the memory consumption on the client is negligible comparing to that for the server and enclave (i.e., less than 1 MB). As shown in Fig. 5.4, the encrypted database on the server always keeps unchanged for SGX-SE1 and SGX-SE2 because they keep the same keyword-document pairs after adding 1×10^6 documents. On the other hand, the memory usage of Bunker-B

Table 5.8: Average time (μs) for adding a keyword-doc pair from Enron dataset and removing 25% documents under different schemes.

Operation	BunkerB	SGX-SE1	SGX-SE2
Insertion	12	7	8
Deletion (25%, 129,305 documents)	12	4	4

keeps increasing when we delete more documents as it should maintain the deleted keyword-document pairs on the server. Within the enclave, Bunker-B does not maintain any persistent data structure while SGX-SE1 and SGX-SE2 need to store the necessary information for deletion. For SGX-SE1, it caches all the document *id* in the enclave, which leads to notably high memory usage (e.g., 304 MB when deleting 25% documents, and 355 MB when deleting 75%). The memory resource requests in SGX-SE1 triggers the paging mechanism of the SGX, resulting in a larger query delay as presented above. SGX-SE2 successfully prevents the paging by using the Bloom filter. After applying a Bloom filter with the false positive rate 10^{-4}, SGX-SE2 only needs 34 MB to store all keyword-document pairs (1.18×10^7 pairs) and maintains a low query delay over the large dataset.

5.5.3 Performance evaluation on Enron dataset

We use a real world dataset to illustrate the practicality of the proposed scheme. Since the bulk deletion (e.g. delete 50%) is rare in the real world, we only focus on the setting with a small deletion portion. Therefore, in the following experiments, we insert the whole Enron dataset and test the average runtime for insertion/deletion as well as the query delay with a small deletion portion (25%).
Insertion and deletion: As described in Section 5.5.2, our schemes are more efficient for the insertion and deletion if the number of keyword-document pairs is larger than the number of documents. The evaluation result on the Enron dataset further verifies our observation: as shown in Table 5.8, our schemes only need 7 μs and 8 μs respectively to insert one keyword-document pair while Bunker-B needs 12 μs to do that. Besides, both of SGX-SE1 and SGX-SE2 only takes 4 μs to delete one document, but Bunker-B still requires 12 μs to execute the same algorithm as the insertion.
Query delay: Finally, we present the query delay when using the Enron dataset. As the Enron dataset has more keyword-document pairs than our synthesis dataset, deleting 25% documents still triggers paging, as it includes more keyword-document pairs than the whole synthesis dataset. In Fig. 5.5, we present the query delay when querying the top-25 frequent keywords in the Enron dataset. The result shows that SGX-SE2 maintains a relative low query delay (530 ms to 900 ms) while SGX-SE1 needs 580 ms to 2.6 s and Bunker-B requires 645 ms to 1.5 s. This above result further illustrates that SGX-SE2 can both prevent the paging within the SGX enclave and eliminate the cost of re-encryption.

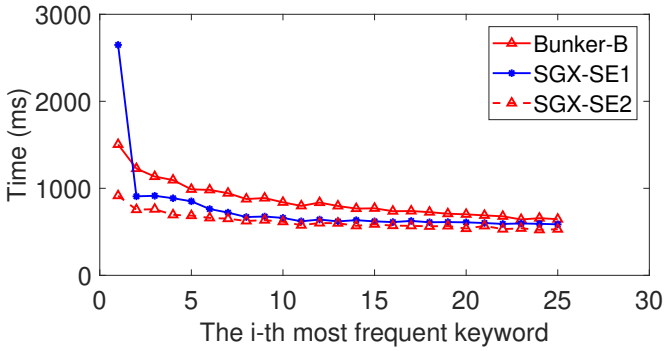


Figure 5.5: The query delay of querying the i-th most frequent keyword in the Enron dataset under different schemes (insert all documents and delete 25% of them).

5.5.4 Discussion

Our key idea to design SGX-supported Type-II backward private schemes is to leverage SGX to take over the most tasks of the client, i.e., tracking keyword states along with data addition and caching deleted data. However, handling large datasets is non-trivial due to the I/O and memory constraints of the SGX enclave. Thus, we further develop batch data processing and state compression technique to reduce the communication overhead between the SGX and untrusted server, and minimise the memory footprint in the enclave. We conduct a comprehensive set of evaluations on both synthetic and real-world datasets, which confirm that our designs outperform the prior art-Bunker-B. We also note that our experimental evaluation (including Bunker-B) is independent of the SGX side-channel attacks since we only focus on the leakage mitigation of dynamic SSE. We further discuss how to independently mitigate those side-channel attacks in Section 5.8.

5.6 Maiden: SGX-supported Type-I scheme

In this section, we first highlight our design and detail the protocol of Maiden, which achieves Type-I *backward privacy* and the scheme achieve optimal search's computation and communication (i.e., $\mathcal{O}(n_w)$) between the client and the server. After that, we analyse the security of the proposed scheme in section 5.6.3. In section 5.7, we evaluate Maiden and compare it with the baseline schemes Orion* and Fort.

5.6.1 Design Intuition

As analysed, Fort relies on TEE (a hardware *Enclave*) to protect the supporting information for Search, which is the deletion information in $Stash_{del}$. The deleted labels in $Stash_{del}$ need to be retrieved via ORAM accesses to the *Server* during Update *deletion* before Search happens. But, this causes the intensive linear

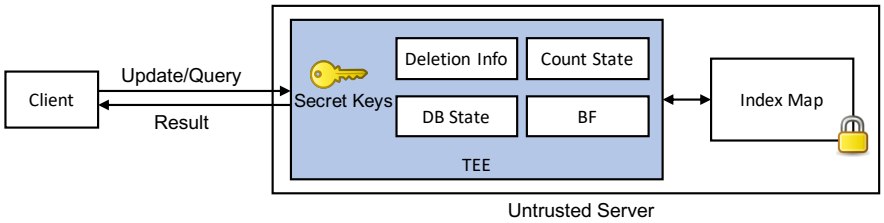


Figure 5.6: High-level illustration of Maiden

scaning operation during Search if the $Stash_{del}$ or the number of generated undeleted/deleted tokens of query keywords is large.

Similar to Fort, we also rely on TEE to protect the supporting information for Search. But, we let the *Enclave* store a normal state map M_c of all (w, id) pairs received during Update *addition*. Based on our assumption, the *Enclave* can protect code and data inside the *Enclave*, migrating M_c to the *Enclave* does not affect the security of our protocol while it fully eliminates ORAM operations for Type-I DSSE schemes. By doing so, our design neither does require the *Server* to store any OMAP data structure in Setup, nor access to that in Update *deletion*. Instead, we simply track the deleted *id* within the *Enclave*. In addition, Maiden also employs a sketch addition BF, i.e., a Bloom filter, to compress all (w, id) pairs added during Update *addition*. With the latest states of tracked keywords, BF, and deleted *id* list, the *Enclave* is able to generate the query tokens for currently matching documents. This helps the scheme to achieve Type-I backward-privacy (i.e., leaking only TimeDB(w)), without exposing historical deletion information to the *Server*. Yet, storing M_c in the *Enclave* may cause the paging overhead in SGX *Enclave Page Cache* (EPC). Nonetheless, we observe that the access with the EPC paging is still one to two orders of magnitude faster than the linear scan in Fort and the ORAM accesses from the *Enclave* to the *Server* in Orion* (see Section 5.7).

5.6.2 The Detailed Protocol

Figure 5.6 presents the design overview of Maiden. The design contains three participants: the trusted *Client*, the TEE denoted as the *Enclave* within the *Server* and the untrusted *Server*. Maiden equips with a lightweight *Client*, which does not maintain any data structure locally. On the other hand, the untrusted *Server* only maintains a normal *index map* M_I to store the mapping between *label* and *value*. The *Enclave* keeps the deletion information. To accelerate the query process, the *Enclave* also has three state maps: The first one is the database state map ST which stores the update counter for each keyword. It indicates the number of updates regarding the keyword. We migrate it from the *Client* to the *Enclave* to reduce the workload for generating query/update tokens. The second one is a count state map M_c maintaining the mapping between (w, id) and the

Algorithm 13 The Setup protocol in Maiden with *storage-free Client*

$\mathsf{Setup}(1^\lambda)$

Client

1: Initialise $k_\Sigma, k_{BF} \xleftarrow{\$} \{0,1\}^\lambda$, and integers l, h
2: Attest and establish a secure channel to *Enclave*
3: Send (k_Σ, k_{BF}, l, h) to *Enclave*

Enclave

1: Initialise $R_k \xleftarrow{\$} \{0,1\}^\lambda$
2: Init maps ST, M_c, and a list d
3: Initialise $BF \leftarrow 0^l$ and $\{H'_j\}_{j\in[h]}$

Server

1: Initialise an index map M_I

corresponding count. The third one is a compressed state map BF stored as a Bloom filter. This indicates whether a given keyword is in a given document, which can be used to facilitate the query process.

To communicate with the *Server*, the *Client* leverages the remote attestation mechanism to establish a secure channel with the *Enclave*. Then, the *Client* can remotely access the database via Setup, Update (add/del documents), and Search operations. The *Enclave* receives the above operations and manipulates the encrypted database stored on the untrusted *Server* on behalf of the *Client*. The detailed procedures of Maiden are provided in Algorithms 13, 14 and 15.

Setup. During Setup (see Algorithm 13), the *Client* attests the *Enclave* and then establishes a secure channel for later communication. The *Enclave* maintains the latest keyword state ST, list d of deleted *id*s, a Bloom filter BF, and importantly a state map M_c that tracks the state c of (w, id). It also receives necessary keys (K_Σ, K_{BF}) provisioned by the *Client*. The *Server* maintains an encrypted map M_I to facilitate the index search.

Update (Algorithm 14): The *Client* directly provides a tuple $(\mathsf{op} = add, \mathsf{in} = \{\mathsf{doc}, id\})$ to the *Enclave* via the secure channel. Then, the *Enclave* generates update tokens $T = \{(u, v)\}$ for $\forall(w, id) \in \mathsf{doc}$ to update the index map M_I, where $(u_i, v_i) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$ with k_{id} generated from $ST[w]$. After that, the *Enclave* tracks the latest state c of $F(k_w, id)$ in the map M_c. This state tracking later enables retrieving the states of deleted doc with id containing w in Search. In addition, the *Enclave* updates the membership of $(w||id)$ to BF. If the Update is *deletion*, given a tuple of (doc', id) sent by the *Client*, the *Enclave* adds id to the list d. It also adds dummy token entries (u', v') generated from doc' to M_I to hide the *deletion* op.

Search (Algorithm 15). The *Client* sends a query keyword w to the *Enclave* for receiving documents matching the keyword. The *Enclave* first performs the membership testing for $(w, id_i), id_i \in d$. With the help of the internal map M_c,

Algorithm 14 The Update protocol in Maiden

Update(op, in

Client

1: **if** op $=$ *add* **then**
2: send (op, in) to *Enclave*
3: **else**
4: send (op, in $=$ (doc$'$, id))
5: **end if**

Enclave

if op $=$ *add* **then**
Parse doc to $D = \{(w, id)\}$
$T \leftarrow \{\emptyset\}$
for $(w, id) \in D$ **do**
$k_w \leftarrow F(k_\Sigma, w)$
$c \leftarrow ST[w]$; $c \leftarrow c + 1$
$k_{id} \leftarrow H_1(k_w, c)$
$(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$
add (u, v) to T
$M_c[F(k_w, id)] \leftarrow c$
$BF[H'_j(k_{BF}, w \parallel id)] \leftarrow 1$ for $j \in [1, h]$
$ST[w] \leftarrow c$
end for
send T to *Server* in batch
else
add id to d
set dummy entries (u', v') in T
end if

Server

$M_I[u] \leftarrow v$ for (u, v) in T

the *Enclave* can retrieve the state of (w, id_i) if id_i was deleted. Then, the *Enclave* can generate the query tokens $\{(u, k_{id}\})$, where $(u, k_{id} \leftarrow (H_2(k_w, c), H_1(k_w, c))$, for undeleted states based on the latest state $ST[w]$ after eliminating deleted ones. Upon receiving the query tokens, the *Server* returns the currently matching document *id_List* to the *Enclave*.

The efficiency of Maiden. The asymptotic search complexity of Maiden is $\mathcal{O}(n_w)$. The scheme relies on the interval map M_c to compute n_w query tokens. It does not need to communicate to the *Server* to find the states of deleted documents. As a trade-off, the scheme maintains a storage of $(\mathcal{O}(WlogD) + \mathcal{O}(a_w W) + \mathcal{O}(N))$, where the significant factor $\mathcal{O}(N)$ presents the size of M_c. The access pattern on M_c during Search is protected by the *Enclave*. Our experiments

Algorithm 15 The Search protocol in Maiden

Search(w)

Enclave

1: Receive w from $Client$
2: $k_w \leftarrow F(k_\Sigma, w)$
3: $st_{(w,c)} \leftarrow \{\emptyset\}, Q \leftarrow [\emptyset]$;
4: **for** id **in** d **do**
5: **if** $BF[H'_j(k_{BF}, w \parallel id)]_{j\in[h]} = 1$ **then**
6: $c \leftarrow M_c[F(k_w, id)]$
7: $st_{(w,c)} \leftarrow \{c\} \cup st_{(w,c)}$
8: **end if**
9: **end for**
10: $st_{(w,c)} \leftarrow \{0, \ldots, ST[w]\} \setminus st_{(w,c)}$
11: **for** c **in** $st_{(w,c)}$ **do**
12: $(u, k_{id}) \leftarrow (H_2(k_w, c), H_1(k_w, c))$
13: $Q \leftarrow \{(u, k_{id})\} \cup Q$
14: **end for**
15: send Q to $Server$ in batch

Server

1: $id\_List \leftarrow \{\emptyset\}$
2: **for** (u, k_{id}) **in** Q **do**
3: $id \leftarrow \mathsf{Dec}(k_{id}, M_I[u])$
4: $id\_List \leftarrow \{id\} \cup id\_List$
5: **end for**
6: send id_List to $Enclave$

show that Maiden is still more than two orders of magnitude faster than the *linear scanning* cost in Fort even when Maiden suffers large memory overhead.

Remarks: Maiden employs a BF for keeping track of *addition*, which facilitates the search token generation in Search. A false positive can be introduced when non-member (w, id) pairs map to set bit positions in the BF vector. This turns out w presumably presented in the deleted document id by the wrong testing. We note that this false match does not affect the correctness of search. The state $ST[w]$ only tracks the matching states for truly existing (w, id) pairs (see Algorithm 14), and no valid state can be found in M_c if w does not exist in the document id (see line 6 in Algorithm 15). Therefore, an invalid state cannot be used to generate query tokens.

Like many other SSE works [75, 80, 81, 83, 2] that focus on the search document index, Search protocol in Maiden only retrieves the document identifiers *id*s of currently matching documents docs containing the query keyword. We note that encrypted data blocks of the documents can be independently outsourced

to an oblivious data structure stored in the *Server*. The idea of using this data structure is to hide document update patterns for the document access. Once the *Enclave* obtains the currently matching *id*s, it can perform oblivious access to the *Server* to retrieve these data blocks and return them to the *Client* via the established secure channel.

5.6.3 Security Analysis

Maiden contains the leakage of Update and Search operations. We formulate the leakage and define $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and a $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ game for an adaptive adversary $\mathcal{A}$ and a polynomial time simulator $\mathcal{S}$ with the security parameter λ as follows.

Let $\mathcal{L}$ be a stateful leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch}, \mathcal{L}^{hw})$, where the first three functions are inherited from DSSE *Server*. They define the information exposed to the *Server* in Setup, Update and Search, respectively. Besides, $\mathcal{L}^{hw}$ defines the inherent leakage of the used SGX *Enclave* communicating with the *Server*. In Setup, Maiden only leaks the data structure of M_I (i.e., the encrypted index). We note that the state map M_c is protected by SGX *Enclave* and it is not exposed to the *Server*. In Update($\mathsf{op} = \{add, del\}, \mathsf{in}$), Maiden leaks the data access pattern T_{M_I} of encrypted entries to be inserted in M_I. Hence, $\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}) = \{T_{M_I}\}$. In Search($w$), Maiden leaks the access pattern on M_I when the *Enclave* queries n_w, named $\mathsf{ap}_{M_I}(w)$. Then, formally $\mathcal{L}^{Srch}(w) = \{\mathsf{ap}_{M_I}(w)\}$. We define $\mathcal{L}^{hw}(M_I)$ as the hardware leakage during Update and Search. That includes memory addresses, the time log, and the size of the manipulated memory area. We write $\mathcal{L}^{hw.Updt}(\mathsf{op}, \mathsf{in}) \leftarrow (M_I)^{Updt}$, which outputs the trace τ of $\{(v, s, t)\}$ on M_I, where v is the encrypted data inserted into M_I, s is the memory size of v, and t is the accessing timestamp of op. We note $\mathcal{L}^{hw.Srch}(w) \leftarrow (M_I)^{Srch}(w)$, which also leaks the trace τ of entries matching w in M_I. We let EDB_k be the state of EDB after updated by the k-th operation $(\mathsf{op}, \mathsf{in})_k$.

Definition 16. *Consider* Maiden *scheme that consists of three protocols* Setup, Update, *and* Search. *Consider the probabilistic experiments* $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ *and* $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, *whereas* $\mathcal{A}$ *is a stateful adversary, and* $\mathcal{S}$ *is a stateful simulator that gets the leakage function* $\mathcal{L}$.

$\mathbf{Real}_{\mathcal{A}}(\lambda)$: The challenger runs $\mathsf{Setup}(1^{\lambda})$. Then, $\mathcal{A}$ chooses a database $\mathsf{DB} = \{\mathsf{doc}_i\}_{i \in Z}$ and makes a polynomial number of Updates (addition/deletion) with $(\mathsf{op}, \mathsf{in})$, where Z is a natural number of documents, and $(\mathsf{op} = add, in = \{\mathsf{doc}_i, id_i\})$ or $(\mathsf{op} = del, \mathsf{in} = \{\mathsf{doc}', id_i\})$. Accordingly, the challenger runs those updates with $\mathsf{Update}(\mathsf{op}, \mathsf{in})$ and eventually returns the tuple $(M_I)^{Updt}$ to $\mathcal{A}$. After that, $\mathcal{A}$ adaptively chooses the keyword w (*resp.*, $(\mathsf{op}, \mathsf{in})$) to search (*resp.*, update). In response, the challenger runs $\mathsf{Search}(w)$ (*resp.*, $\mathsf{Update}(\mathsf{op},\mathsf{in})$) and returns the transcript of each operation. The challenger also returns $(M_I)^{Srch}$ to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b.

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: The challenger runs $\mathcal{S}(\mathcal{L}^{Stp}(1^\lambda))$. $\mathcal{A}$ chooses a $\mathsf{DB} = \{\mathsf{doc}_i\}_{i\in Z}$, and makes a polynomial number of $\mathsf{Updates}$ (addition/deletion) with $(\mathsf{op}, \mathsf{in})$ to the $\mathcal{S}$, where Z is a natural number of documents, and $(\mathsf{op} = add, in = \{\mathsf{doc}_i, id_i\})$ or $(\mathsf{op} = del, \mathsf{in} = \{\mathsf{doc}', id_i\})$ By using $\mathcal{L}^{Updt}$ and $\mathcal{L}^{hw.Updt}$, $\mathcal{S}$ creates a tuple of (M_I) and send them to the *Server*. Then, $\mathcal{A}$ adaptively chooses the keyword w (*resp.*, $(\mathsf{op}, \mathsf{in})$) to search (*resp.*, update). The challenger returns the transcript simulated by $\mathcal{S}(\mathcal{L}^{Srch}(w), \mathcal{L}^{hw.Srch}(w))$ (*resp.*, $\mathcal{S}(\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}), \mathcal{L}^{hw.Updt}(\mathsf{op}, \mathsf{in}))$). Finally, $\mathcal{A}$ returns a bit b.

We say Maiden is $\mathcal{L}$-secure against adaptive chosen-keyword attacks if for all probabilistic polynomial-time algorithms $\mathcal{A}$, there exist a PPT simulator $\mathcal{S}$ such that

$$|Pr[\mathbf{Real}_{\mathcal{A}}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda)$$

Theorem 4. *Assuming the map M_c is secure and protected by SGX Enclave, and the communication between the Client and the Enclave is secure,* Maiden *is an adaptively-secure SSE scheme with ($\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}) = \mathsf{op}$, $\mathcal{L}^{hw.Updt}(\mathsf{op}, \mathsf{in}) = (M_I)^{Updt}$), and ($\mathcal{L}^{Srch}(w) = \mathsf{TimeDB}(w)$, $\mathcal{L}^{hw.Srch}(w) = (M_I)^{Srch}$).*

Proof. We now prove Theorem 4 by describing a PPT simulator $\mathcal{S}$ for which a PPT adversary $\mathcal{A}$ can distinguish $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability. We now describe $\mathcal{S}$ as follows:

- $\mathcal{S}.Init(1^\lambda)$. It generates a random key $\tilde{K} = (\tilde{k}_\Sigma, \tilde{k}_{BF})$ to simulate the key components that the enclave contains (see Figure 13). $\mathcal{S}$ also creates an empty M_I. It then sets $\mathsf{EDB}_0 \leftarrow M_I$ and sends it to the *Server*, and set $st_{\mathcal{S}}$ to null.

- $\mathcal{S}.Update(st_{\mathcal{S}}, \mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in})_k, \mathcal{L}^{hw.Updt}(\mathsf{op}, \mathsf{in})_k, \mathsf{EDB}_{k-1})$. Recall that $\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in})_k = \{T_{M_I}\}_k$, and $\mathcal{L}^{hw.Updt}(\mathsf{op}, \mathsf{in})_k = \tau_k$. $\mathcal{A}$ selects a doc with id and send a tuple of $(\mathsf{op} = add, in = \{\mathsf{doc}, id\})$ or $(\mathsf{op} = del, \mathsf{in} = \{\mathsf{doc}', id\})$ to $\mathcal{S}$, where doc' is a dummy doc. Upon receiving doc, $\mathcal{S}$ computes new entries and sends them to the *Server* for the insertion to M_I. We note that $\mathcal{S}$ computes these new entries by simulating the output of the secure hardware (i.e., TEE). To do so, the simulator first takes encrypted data in $\{T_{M_I}\}_k$ and decrypts them using $\tilde{k}_\Sigma$. Based on the timestamps and data sizes revealed in τ_k, $\mathcal{S}$ tries to locally updates $st_{\mathcal{S}}$, and generates new tokens for (w, id) pairs in doc. It then sends these new tokens to the *Server*.

- $\mathcal{S}.Search(st_{\mathcal{S}}, \mathcal{L}^{Srch}(w)_k, \mathcal{L}^{hw.Srch}(w)_k, \mathsf{EDB}_{k-1})$. $\mathcal{A}$ choose a keyword w and sends it to $\mathcal{S}$. Recall that $\mathcal{L}^{Srch}(w)_k = \mathsf{TimeDB}(w)$. Then, with $\mathcal{L}^{hw.Srch}(w)$ and $st_{\mathcal{S}}$, $\mathcal{S}$ simulates the outputs of the secure hardware and sends them to the *Server*. Finally, let R_w be the set of document identifiers corresponding to the queried keyword, as derived from $\mathsf{TimeDB}(w)$. $\mathcal{S}$ sends R_w to $\mathcal{A}$.

Consider the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ game with the described simulator $\mathcal{S}$, the produced transcript is indistinguishable from the one produced during $\mathbf{Real}_{\mathcal{A}}(\lambda)$ as the map M_I get entries inserted in the same document addition manner, the state protected by secure TEE, and the document identifiers of the query keyword are also the same.

We note that $\mathcal{A}$ knows the timestamps when encrypted entries are inserted into the index map M_I in both *addition/deletion* Updates, but $\mathcal{A}$ cannot distinguish the Update is *addition* or *deletion*. The reason is the map M_I always get entries inserted during the doc *addition/deletion* under $\mathcal{A}$'s view. During Search, Maiden only reveals n_w during the query on M_I. The rest information of d_w and M_c are within the *Enclave*. Therefore, $\mathcal{A}$ cannot match the accessed positions in Search to any previous document Update on particular w. This ensures that Maiden only reveals $\mathsf{TimeDB}(w)$. □

5.7 Evaluation of Type-I Backward privacy

SGX-supported schemes for evaluation. We develop Maiden and two baseline schemes Orion* and Fort for comparison by using Intel SGX SDK and C++[4].

The prototype of Maiden contains three components of *Client*, the *Enclave*, and the *Server*. We leverage standard *ecalls/ocalls* interfaces provided by SGX SDK to implement the communication between these components. In all experiments, we set a batch size to 1×10^4 when the *Enclave* sends query tokens to the *Server* during Search via the *ocall* interface. Note that, we use the same Bloom filter's configuration for all the following used datasets, with the false positive rate 10^{-4} and it can store up to 1.5×10^7 pairs.

For baseline schemes, we first choose Orion [80] since it is publicly known as the most optimal non-TEE supported Type-I *backward-private* scheme with $\mathcal{O}(n_w log^2 N)$ *search latency*. We migrate the *Client* of Orion to the *Enclave*, and name this ported version as Orion*. The *Enclave* in Orion* stores the map $LastInd[w]$ that maintains the most recently inserted file identifier matching w, and the map $UpdtCnt[w]$ tracking the total number of currently matched documents of w. The *Server* in Orion* maintains two oblivious maps (OMAPs) to facilitate the Update and Search operations, as presented in the original scheme. They are OMAP_{upd} and OMAP_{src}, respectively. We carefully port the implementation of Orion to the *Enclave* and also construct the OMAP_{upd} and OMAP_{src} in the *Server* by using oblivious data structures initiated by AVL trees [80], as introduced in the original Orion scheme. We refer readers to the original work [80] for the detailed protocols of the scheme.

In addition, we also implement the *Enclave* component of Fort during Search for comparison since the implementation of Fort is not publicly available. In

[4]Source code: `https://github.com/MonashCybersecurityLab/SGXSSE`

Table 5.9: Statistics of the datasets used in the evaluation of Type-I.

Name	# of keywords	# of docs	# of keyword-doc pairs
DS1	500	10,000	119,286
DS2	1,000	1,000,000	8,281,451
Enron	23,355	85,000	8,895,865

Table 5.10: Avg. (μs) for adding/deleting a (w, id) pair when adding/deleting a portion of DS1 and DS2.

Scheme	Add 100% docs		Del 25% docs		Del 50% docs		Del 75% docs	
	DS1	DS2	DS1	DS2	DS1	DS2	DS1	DS2
Maiden	19	43	1.24	1.4	2.09	2.4	3.01	3.3
Orion*	361	601	575	5,059	820	8,564.1	1,021.3	11,495.1

Table 5.11: Number of *ocalls* for data communication between *Enclave* and *Server* in adding/deleting a portion of documents

Scheme	Add 100% docs		Del 25% docs		Del 50% docs		Del 75% docs	
	DS1	DS2	DS1	DS2	DS1	DS2	DS1	DS2
Maiden	12	829	$1^\star$	$1^\star$	$1^\star$	$1^\star$	$1^\star$	$1^\star$
Orion*	8.9×10^4	6.2×10^6	7.86×10^5	1.34×10^7	9.2×10^5	1.7×10^7	3.2×10^6	3.6×10^7

*: Maiden performs 1 *ocall* per doc in non-batch setting to add dummy entries to M_I

details, for a given sampled $Stash_{del}$ cached in the *Enclave*, we ask the *Enclave* to generate deleted/undeleted query tokens for a query keyword w. Then, the *Enclave* linearly scans $Stash_{del}$ to identify and discard a portion of the deleted tokens existing in the $Stash_{del}$.We only measure the scanning time and consider it as the *search latency* for Fort.

For both three schemes, we leverage built-in cryptographic primitives in `sgx_tcrypto` library to implement required cryptographic operations. The prototypes of these schemes are deployed into an Intel SGX-equipped station with Intel core i7 2.6 GHz and 32 GB RAM.

Experimental datasets: We use two synthesis datasets (a small DS1: 70 MB, and a large DS2: 4 GB), and a portion of public Enron email dataset[5] (895 MB). The synthesis datasets are generated from the American English keyword frequency data and sampled by using the Zipf's law distribution. With DS1, the keyword's state map M_c of Maiden can fit in the limited memory protected by SGX *Enclave* (i.e, 98 MB), while the map causes *paging overhead* when DS2 is used. The *paging overhead* is essential to enable *Enclave Page Cache* (EPC) perform page swaps of Intel SGX [84]. Table 5.9 summarises these used datasets.

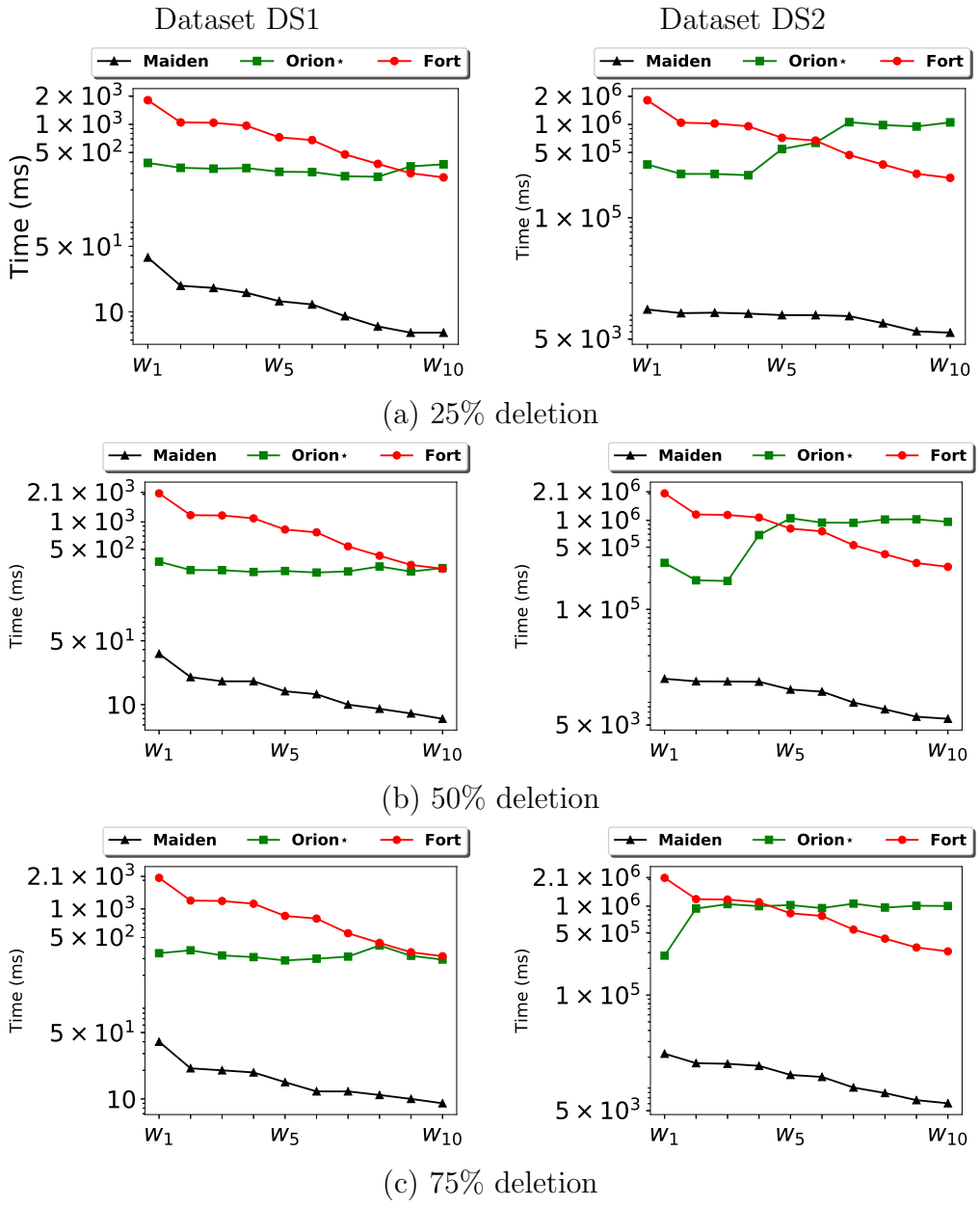
[5]Enron email dataset: `https://www.cs.cmu.edu/~./enron/`

5.7.1 The performance on the Synthesis Datasets

Insertion and Deletion. We first evaluate the time for insertion and deletion an (w, id) pair under different schemes when using datasets DS1 and DS2. As shown in Table 5.10, Orion* takes 361 and 601 μs to insert a pair to DS1 and DS2, respectively. That latency is about $(13 \sim 36)\times$ significantly higher than Maiden. The reason is because the *Enclave* in Orion* needs to update/traverse the AVL tree structures of both OMAP_{upd} and OMAP_{src} stored in the *Server*. Table 5.11 confirms that the communication (i.e.,*ocalls*) in Orion* during *addition* is about $(4.6 \times 10^3 \sim 1.4 \times 10^5)\times$ more than Maiden. It is clearly that Maiden is more efficient because it only updates the local state map M_c within the *Enclave*. With Maiden, the number of *ocalls* contacting to the *Server* is negligible (12 *ocalls* with DS1, 829 *ocalls* with DS2). This communication is purely made when the *Enclave* inserts encrypted entries to the index map M_I.

Similar with the *addition*, Orion* operates on both OMAP_{upd} and OMAP_{src} to retrieve/update new state for every (w, id) pairs with the recently inserted document identifiers. Therefore, the time to delete a document with Orion* scales to the number of keywords in that document. Averagely, Orion* takes $6,325$ ms to delete a document containing $8 \sim 14$ keywords. Table 5.11 reports the latency when deleting a portion of documents in DS1 and DS2. In contrast, Maiden takes a negligible time cost to delete a document. The main reason behind it is because Maiden only tracks the identifiers of those deleted documents within the *Enclave*. It only takes 1 *ocall* per a deleted document to insert dummy entries into the index map M_I to hide the operation.

Query Delay. Next, we monitor the search latency between Fort, Orion*, and Maiden when using datasets DS1 and DS2. We choose to query the top-10 keywords in the datasets after deleting a portion of documents. With DS1, we insert 1×10^4 documents, then delete 25%, 50%, and 75% of the documents, respectively. Similarly, with DS2, we insert 1×10^6 documents, and also delete these portions of the documents. Figure 5.7 reports the search latency under different schemes. The result shows that Fort has the downward trend when querying less-frequent keywords. The reason is because those keywords have fewer number of undeleted/deleted tokens to be scanned against the map $Stash_{del}$ stored in the *Enclave*. Averagely, scanning a token (w, id) of the most frequent keyword in DS1 takes 18 μs when $Stash_{del} = 1 \times 10^5$. In the larger dataset DS2, this cost is averagely 284.2 μs to scan just an (w, id) pair for the most frequent keyword when $Stash_{del} = 6.3 \times 10^6$. The more documents deleted, the longer time Fort takes to scan the tokens of query keywords. Querying the most frequent keyword in DS2 after deleting 75% documents takes more than 2×10^6 ms. With Orion*, the scheme takes 1.4×10^3 ms less than Fort to query the most frequent keyword w (with the frequency of 1×10^4) in the small dataset DS1 after deleting 25% documents. The reason is because Orion* only computes the tokens of currently documents matching the query keyword w. However, the latency to



(a) 25% deletion

(b) 50% deletion

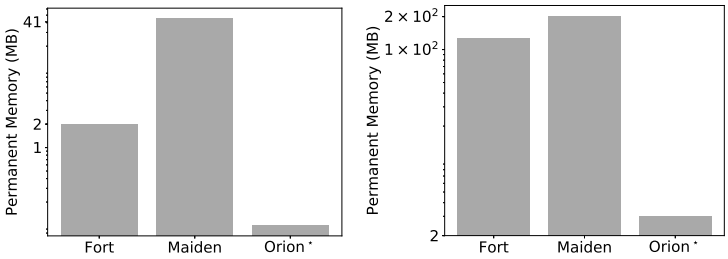
(c) 75% deletion

Figure 5.7: The query delay of querying the i-th most frequent keyword in the DS1 and DS2 datasets after deleting a portion of documents

query documents matching a keyword in the larger dataset DS2 is non-trivial. It takes about 9.2×10^5 ms to query a keyword in the top-10 frequent keywords in DS2 after deleting 75% documents. The reason for it is because the *Enclave* in Orion* needs to retrieve matching nodes from a large AVL tree (with 2^{23} AVL nodes in DS2) of OMAP_{src}, where the tree's nodes are stored in the random positions of the underlying ORAM structure stored in the *Server*. In addition, the oblivious accesses in Orion* also include the cost of mapping visited AVL nodes to new ORAM positions, and encrypting/writing them back to the *Server*.

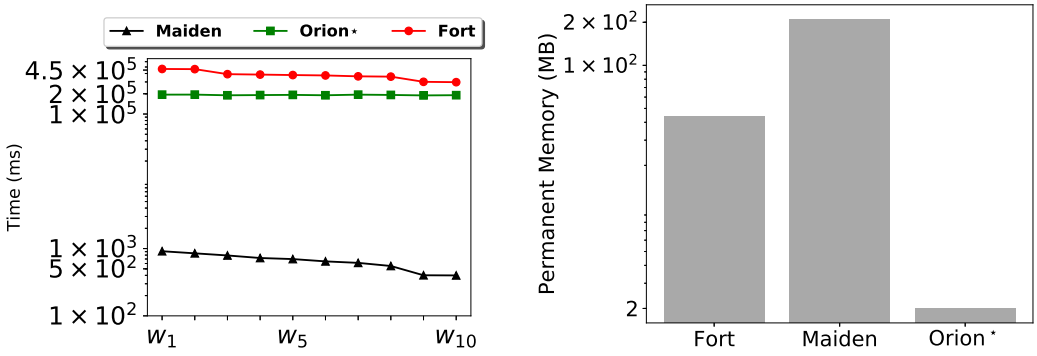
Figure 5.7 shows that Maiden completely outperforms Fort and Orion* in both DS1 and DS2. With the small dataset DS1, querying the most frequency keyword with Maiden is 10× and 47× faster than Orion* and Fort, respectively, after deleting 25% documents in DS1. With the large dataset DS2, the difference is about 35× and 174× faster than Orion* and Fort, respectively. When deleting 75% documents in DS2, Maiden is more efficient than Orion* and Fort about 12× and 95× when querying the most frequent keyword. Even when querying the 10-th frequent keyword, that difference varies from $45 \sim 175\times$. Note that, the main difference in the search of Maiden compared to others is how it generates the query tokens of currently matching documents for query keywords. Unlike Fort, Maiden does not require intensive computation (i.e. linear scanning undeleted/deleted tokens of query keywords against the large $Stash_{del}$), neither does Maiden perform oblivious accesses to the *Server* to identify the state of currently matching identifiers like Orion*. We note that membership testing with Bloom filter in Search of Maiden is $\mathcal{O}(1)$. With tracked deleted identifiers in the list d, the *Enclave* in Maiden can directly retrieve the deleted state of deleted documents matching the query keyword. The difference between DS1 and DS2 is that the size of the state map M_c in Maiden triggers paging overhead in SGX *Enclave*. We monitor that M_c in DS1 takes 2.27 MB, while the latter exceeds 157 MB. The paging cost is added to the search latency of Maiden when the EPC swaps pages to access the states of deleted (w, id) from the map. Nonetheless, with paging access, we observe that Maiden is averagely 75× and 90× faster than Orion* and Fort when querying the top-10 frequent keywords in DS2 after deleting 25% documents of the dataset. With 75% documents deleted in DS2, the difference is in the range $70 \sim 72\times$ faster than Orion* and Fort.

Memory Storage. Finally, we present the memory storage in the *Enclave* of the three schemes. As shown in Figure 5.8, Maiden takes the largest memory, about 41 MB, among others when using dataset DS1. The main reason is because the storage of M_c (i.e., 2.27 MB) and the configured Bloom filter 38 MB (i.e, $P_e = 10^{-4}$) . When using DS2, the memory storage is about 200 MB due to the large state map M_c (i.e., 157 MB). We note that, deleting more documents, i.e., 75% documents in DS1 and DS2 does not affect significantly the memory consumption in the *Enclave* of Maiden. The reason is because only the identifiers of these deleted documents are appended in the list d of the scheme. Note that, the size of d is only about 30 KB and 3 MB when deleting 75% documents



(a) DS1 (b) DS2

Figure 5.8: The permanent memory in the *Enclave* in Type-I evaluation.



(a) Query latency (b) Memory Storage

Figure 5.9: Query latency and memory storage between schemes in Type-I evaluation

in DS1 and DS2, respectively. With Orion*, the memory consumption in the *Enclave* is negligible because the scheme only maintains the number of current documents and the most recently inserted document identifiers matching every keyword in the maps $UpdtCnt[w]$ and $LastInd[w]$, respectively. With Fort, the scheme maintains $Stash_{del} = \Sigma_{\forall w} d_w$ in the *Enclave*. Hence, with the DS2, deleting 25% documents requires about 121 MB to store 6.4×10^6 deleted tokens.

5.7.2 The performance on the Enron Dataset

Query Latency. We use a portion of real world Enron email dataset to demonstrate the efficient of Maiden when the paging overhead in SGX Enclave occurs. We insert 85,000 email documents and test the average query delay with a small deletion portion 25%. With this deletion portion, there is no paging overhead in Fort. Figure 5.9(a) reports the query delay when querying the top-10 frequent keyword in Enron dataset. The result shows that Maiden is averagely 291× and 575× faster than Orion* and Fort, respectively. We obtain that Maiden is more efficient than Fort and Orion* during Search with the used Enron dataset. The reason is because Enron actually has more keywords in the same deletion portion

compared to the used dataset DS2 (see Table 5.9). With DS2, the rate of cache hit in Fort is 1.56×10^{-1}, an order of magnitude higher than that rate (i.e., 1.52×10^{-2}) when Enron dataset is used. We note that reducing the false positive rate P_e of the Bloom filter used in Maiden does not change much its search performance. Changing $P_e = 10^{-6}$ from $P_e = 10^{-4}$, it only incurs averagely an additional $120ms$ latency to search the top-10 frequent keywords.

Memory Storage. Clearly, Maiden needs the largest SGX Enclave memory to store the state map M_c of the all (w, id) pairs in the dataset ($\sim$ 170 MB). Fort consumes a minimal storage $Stash_{del}$ = 44 MB to store 2.3×10^6 deleted tokens. The limitation of Maiden is the memory bottleneck in the SGX Enclave. In the future work, we will improve the memory efficient of the scheme. In the meantime, we expect the new version Intel SGXv2 to increase the size of the *enclave page cache* more greatly and support dynamic page allocation [133].

5.8 SGX-Related Attacks and Defence

With Intel SGX's security guarantee, CPU is the only trustworthy component where enclave's code and data are handled in plain-text format, all other components including operating system (OS), memory, hypervisor, memory bus, etc. are treated as untrusted. Whenever the code/data are moved out of the CPU, i.e., into untrusted DRAM memory space, they are encrypted and integrity protected. However, there have been many side-channel attacks showing that it is not impossible to infer/steal the secrets protected by the SGX enclaves. Those attacks leverage the side information revealed by cache [130, 134, 135], page table [131, 136, 137], transient execution [30, 138] and others [139, 140, 141]. In this section, we discuss significant SGX-related attacks and existing defences, and consider how they can be applied to our proposed scheme.

5.8.1 Cache Side-channel Attacks and Defence

While enclave's code and data are encrypted and authenticated by the CPU, they are still stored unencrypted in CPU's caches and registers to facilitate the execution. Therefore, by monitoring the cache channels, an adversary can learn fine-grained data leakage of the enclave. These cache-based attacks have been investigated at L1/L2 caches (on the same shared CPU core with hyper-threading) and L3 cache (cross-CPU core attacks). With shared L1/L2 cache channels, an adversarial process and a victim enclave process interleaved on the same physical CPU core, sharing both L1 cache that stores code and data, and L2 cache that unifies code and data at fine granularity level. Therefore, the adversary can infer the memory content of the victim enclave via the cache data access pattern. This is also known as time-sliced cache attacks [130, 134, 142]. With L3 cache channel, i.e., the last level cache (LLC) shared between CPU-cores, Schwartz et

al. [135] developed an unprivileged program injected in a malicious enclave to conceal the secret key of a co-located victim enclave running on the same host machine. The simplest way to prevent the adversarial hyperthread from accessing to the shared L1/L2 cache channels of the victim enclave's process is by disabling hyper-threading [143]. However, this solution is not highly recommended since it obstructs other applications' performance and restricts CPUID instruction access from the victim enclave. Alternatively, preferred solutions to mitigate these cache channel attacks are transaction memory randomisation [144, 145] and oblivious execution approaches [146, 147, 148] to obfuscate the cache data access pattern, and/or using Varys-protected run-time environment [149].

Transaction memory randomisation: Dr. SGX [144] applies a hardening randomisation technique to all data locations in enclave's memory at cache-line granularity. By randomising every eight data blocks at once, it makes the cache tracing of enclave's data is harder. Cloak [145] is also another mitigation solution using memory transaction technique. It uses Intel Transactional Synchronization Extension (TSX) to construct atomic memory operations that obliviously hide the memory access of enclave's data. The idea is that the enclave is requested to touch all cache lines before it accesses to the real data. Therefore, an adversary, monitoring the cache channel, learns nothing about the enclave's data access. We note that Dr. SGX is built as a compiler tool and Cloak simply just requires annotating enclave's data. Therefore, they can be applied directly to current SGX-supported (backward-private) SE schemes, (i.e., Maiden, Fort [83], Orion*, and SGX-SE1 and SGX-SE2 [2]), without changing the schemes' design. As a trade-off, they require an increasing overhead of averagely 4.8× for Dr. SGX, and 2.48× for Cloak, respectively. The penalty overhead is added to the complexity of enclave's computation in Update and Search operations for all the schemes.

Oblivious memory execution : Oblivious execution is also another approach to hide all enclave's code and data access. For example, Raccoon [146] provides annotation guides to hide the data access regarding different data sizes in the enclave. For small-size secrets, the data access is hidden by using Path ORAM. Otherwise, Raccoon uses Advanced Vector Extensions (AVX) intrinsic operations provided by Intel to stream over large data structures. In addition, Racoon also obfuscates control flows by using oblivious operation primitives extended from CMOV x86 instructions. Applying Racoon to current backward-private SE schemes' operations will add about 16× penalty overhead. We also note that this solution can be plugged directly to the current implementation of the schemes, without changing the design. Alternatively, ZeroTrace [147] also proposes efficient oblivious memory primitives by using Circuit-ORAM. It runs on top of a software memory controller. Therefore, applying ZeroTrace to the implementation of the backward-private schemes requires a minor modification of using the memory controller interface for all enclave's accesses, without changing the (backward-private) SE schemes' implementation design. Other oblivious primitives of memory assignments and comparisons, and oblivious array ac-

cess [148] can be directly adapted to the schemes' implementation. Again, using oblivious data structures like ORAM will reduce the efficiency, and designing an TEE-based SSE schemes that can address memory access side-channels without using ORAM is still an open question.

Side-channel protected runtime environment: To mitigate the cache-channel attacks, Varys [149] provides a trusted core reservation technique that ensures the CPU-core only shares its caches to Varys's benign threads, preventing adversarial threads from using the same caches. In particular, for single-thread application like Maiden, Fort [83], Orion*, and SGX-SE1 and SGX-SE2 [2], we realise that Varys would simply pair that application thread with a service thread to reserve the complete core, and schedule it for runtime monitoring. Varys was reportedly to incur 15% penalty overhead in previous case studies [149]. Therefore, we assume that it would not impact much on the performance of these SE schemes. In addition, Varys is built as a Low Level Virtual Machine (LLVM)-based compiler; therefore, it also does not affect the schemes' code structure.

5.8.2 Page-Table Side-Channel Attacks and Defence

Apart from exploiting CPU caches, enclave's code and data are stored in Enclave Page Cache (EPC), which is the a subset of a contiguous Processor Reserved Memory (PRM) of DRAM. Every 4KB enclave page of code and data is allocated from the EPC (including paging). With SGX design, the page table is managed by the (untrusted) OS. Therefore, it reveals the page-level access patterns of the victim enclave. The malicious OS can trigger page faults from requested pages during the enclave execution to learn the enclave's control flow and memory access. That page fault channel is sufficiently informative to extract the rich text information [137], or secret key bits [131] of victim enclaves. While increasing high page fault rate, these attacks consequently trigger asynchronous enclave exit (AEX) to report the accessing address of the faulting page to the (malicious) OS (i.e., even up to 11000 exits per second [136]). Therefore, a common system-level solution to thwart page-table side-channel attacks is to monitor and detect AEXs due to interrupts of page faults, like T-SGX [150] and Déjá Vu [151]. This solution allows the enclave to stop its execution if the detection occurs. Using this detection solution is a separated configuration and it also does not affect the designs of the SE schemes. Alternatively, Varys [149], an LLVM-based compiler, also introduces a monitoring mechanism for enclave exits so that the application thread running in the enclave can be terminated, without revealing further faulty pages' addresses to the OS. Other studies [152, 153] also provide a self-verification mechanism to the enclave when page faults occur with an extra $1.2 \sim 2.4\times$ overhead. We note that these compiler-based tools do not cause any impact on the SE scheme's code structure and implementation.

5.8.3 Transient Execution Attacks and Defence

Recent works also exploit the CPU execution design to steal enclave's secrets. The execution of a program in Intel CPUs (i.e., Intel's Skylake microarchitecture) is facilitated by two parts including a *frontend* component and an *Execution Engine*. While the *frontend* performs speculative execution predicting branch instruction to speed up the program's execution, the *Execution Engine* can execute instructions in out-of-order fashion so that multiple instructions can be executed in parallel. It has been shown that both these two parts can be exploited. For instance, Chen et al. [30] demonstrated SgxPectre attack that poisons the branch prediction of a victim enclave so that malicious injected secret-leaking instructions can be executed when the victim enclave runs. Unlike SgxPectre attack, Foreshadow [138] exploits the out-of-order execution to access even pages where the victim enclave's memory lies in. It exploits OS's system calls to trigger page faults and then uses Meltdown-like technique to access enclave's data before the page fault is handled. After that, it uses caching side-channel attack (i.e., Flush+Reload [129]) to read enclave's secrets from CPU's cache. We note that these attacks cannot solely mitigated by software solutions. It would include updates to OS, hypervisors, and CPU microcode. We refer interested readers to [154, 155] for additional details about these hardware countermeasures.

5.8.4 Other Attacks and Defence

Apart from side-channel and transient execution attacks, recent studies found that SGX Enclave is also vulnerable to memory-corruption attacks [139, 140]. These attacks often assume that the adversary has knowledge of vulnerabilities in the enclave's legacy code (i.e., stack overflow, data type confusion, format string vulnerability, etc). Therefore, the untrusted code outside the enclave could pass parameters or invokes specific functions in the enclave, which subsequently perform sensitive computations. Since SGX instructions of *ecalls* (i.e., EENTER) and *ocalls* (i.e., EEXIT) do not clear CPU registers, thereby they allow the execution of (vulnerable) trusted code in the enclave to pass sensitive results/access to untrusted code (i.e., gaining access to CPU registers [139], or exfiltrating confidential code and data from enclave memory [140]). Mitigation solutions could be either 1) restricting the enclave's permission from accessing pages containing malicious code injection [156], 2) providing memory-safe access for variables/objects in SGX [157], 3) designing memory randomisation scheme for SGX enclave [158], or 4) static host-to-enclave code analysis tool [159]. We note that memory-corruption attacks are out of our focus since we consider that SE schemes used in our experiment and evaluation are memory-safe implementation.

Finally, adversaries can rely on power management software [141] to induce memory errors and cause overflows in the SGX runtime. Particularly, they can

change protected values in the EPC region and direct in-enclave pointers to untrusted memory via this attack. Unlike prior attacks, this attack does not require any knowledge on code/memory. Fortunately, the issue has been fixed by recent microcode and BIOS updates offered by Intel [160].

5.8.5 Discussion

In this section, we resort to secure enclaves to design Maiden, the first strong backward-private DSSE scheme without relying on ORAM. Our key idea is to keep track of the states of updates and the deletion information inside the secure enclave to prevent the leakage from the server. To speed up, we further leverage a compressed data structure to maintain a sketch of addition operations in the enclave to facilitate the fast generation of search tokens of non-deleted data. We perform comprehensive evaluations on both synthetic and real-world datasets. Our results confirm that Maiden outperforms the prior work.

Chapter 6

Conclusion

This thesis presented new efficient design of dynamic searchable encryption against attacks that exploit the inherent leakage of update (i.e., addition, deletion) and search operations. This last chapter summarises the results presented previously, and open new research directions.

6.1 Summary of the Results

In this thesis, we first explored new threat models against dynamic SE based on leakage-abuse attacks, and new efficient design to mitigate the attacks. Then, we presented new efficient design to achieve the advanced security notions of forward and backward privacy to reduce the leakage of addition and deletion updates during search.

In chapter 3, we saw that leakage-abuse attacks bring the new attack assumption of auxiliary knowledge to enable the adversary break the claimed security of SE in static setting. Thus, in chapter 4, we conceptualised the attacks in dynamic setting by presenting two new threat models of non-persistent and persistent adversaries. We defined new constraints to capture the knowledge of these adversaries and provided new security definitions for dynamic SE (section 4.4). Accordingly, we designed new padding countermeasures to mitigate them. Having not seen any practical encrypted database system able to mitigate leakage-abuse attacks, we employed the dynamic SE framework and developed ShieldDB, a streaming encrypted database, which supports keyword search over streaming encrypted data (section 4.5). We showed that our proposed padding strategy is practical and deployable to real-world streaming applications/systems that require the privacy preservation on data stream. Our proposed design can be easily applied to any existing dynamic SE scheme in the literature.

In chapter 5, we provided new efficient design of dynamic SE to support strong forward and Type-II and Type-I backward privacy. Namely, strong backward privacy enhances the security of a dynamic SE by not revealing the historal data

deletion to the server. We carefully analysed the limitations of the prior forward and backward-private constructions, with and without using trusted execution support (TEE). We find that non-TEE constructions suffer high communication overhead between the client and the server due to either multiple round-trip communication or ORAM bandwidth overhead. In addition, existing TEE-supported constructions have high search latency due to the computation bottleneck in the SGX enclave. Therefore, we proposed SGX-SE1 and SGX-SE2 for Type-II backward privacy (section 5.4). For Type-I, we also proposed Maiden without relying on ORAM (section 5.6). Our proposed designs reduce the overhead computation of the SGX enclave and also reduces the communication between the enclave and server. We implemented prior works and our schemes, and conducted a detailed evaluation on the performance under different schemes. The results show that our designs are more efficient in the update operation (addition/deletion) and query latency.

6.2 Future Research Directions

6.2.1 More theoritical directions

An interesting opening question for leakage-abuse attacks in dynamic SSE in general is finding efficient padding solution with low computation and storage overhead. If we recall from section 4.6, the padding dataset used for streaming operations is essentially generated using heuristic algorithm in setup operation. Another limitation is the storage overhead of the dataset until it is used completely via padding strategies. Therefore, future studies may investigate how to dynamically and adaptively generate the padding dataset upon the data has been streamed. This leads to the question of efficient design to achieve strong forward and backward privacy. Indeed, we believe that the asymptotic efficient of the addition and deletion for new entries in the padding dataset can leverage the design of Maiden. The reason is that Maiden does not rely on ORAM to do addition/deletion, and the search latency only depends on the query result. It is clear that the problem of SGX memory overhead caused by Maiden is no longer a matter for new generation of TEE.

Although named as “more theoretical questions”, the answers to these questions would pave the way for various real-world applications. Therefore, they should not be seen and investigated independently without the real application scenarios.

6.2.2 More application-oriented directions

Using TEE has been shown as a potential approach to provide and accelerate the secure computation environment in the untrusted server. On the other

hand, oblivious-RAM (ORAM) is an quintessential approach to enable encrypted search without exposing access pattern but it is shown as an expensive tool. Many privacy-preserving ORAM-base schemes and primitives have been proposed to hide the data access patterns in practical applications/settings, such as file sharing in peer-to-peer network or multi-server ORAM setting. Such applications often assume the peers/clients are trusted so that the data could be returned back for background operations, such as data routing or data re-encryption. Relaxing such security assumption could make the applications/settings more practical. One straightforward direction is considering the semi-honest peers/clients, where the data access pattern at those parties should be protected. Therefore, how could TEE be involved in such applications require further investigation and experiments. In addition, future studies can investigate how TEE and DSSE can be involved to accelerate user's privacy enhancement in existing network distributed systems [161, 162, 163, 164, 165, 166, 167, 168, 128, 59, 169] and learning systems [170, 171, 172, 173].

References

[1] Vo, V., Yuan, X., Sun, S.F., Liu, J.K., Nepal, S., Wang, C.: Shielddb: An encrypted document database with padding countermeasures. IEEE TKDE (2021)

[2] Vo, V., Lai, S., Yuan, X., Sun, S.F., Nepal, S., Liu, J.K.: Accelerating forward and backward private searchable encryption using trusted execution. In: ACNS. (2020)

[3] Vo, V., Lai, S., Yuan, X., Nepal, S., Liu, J.K.: Towards efficient and strong backward private searchable encryption with secure enclaves. In: ACNS'21. (2021)

[4] Bhargavan, K., Boureanu, I., Fouque, P.A., Onete, C., Richard, B.: Content delivery over tls: a cryptographic analysis of keyless ssl. In: 2017 IEEE European Symposium on Security and Privacy (EuroS P). (2017)

[5] Information is Beautiful: World's Biggest Data Breaches. Online at `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/` (2016)

[6] IBM: 2017 ponemon cost of data breach study - australia-specific report. Online at `https://www-03.ibm.com/security/au/en/data-breach/` (2019)

[7] Verizon: 2020 data breach investigations report. Online at `https://enterprise.verizon.com/en-au/resources/reports/dbir/` (2020)

[8] Australian Government: Australia's cyber security strategy. Online at `https://cybersecuritystrategy.pmc.gov.au/assets/img/PMC-Cyber-Strategy.pdf` (2016)

[9] Australian Government: Science and research priorities. Online at `http://www.science.gov.au/scienceGov/ScienceAndResearchPriorities/Pages/default.aspx` (2015)

[10] Liu, C., Chen, J., Yang, L.T., Zhang, X., Yang, C., Ranjan, R., Kotagiri, R.: Authorized public auditing of dynamic big data storage on cloud with efficient verifiable fine-grained updates. IEEE TPDS (2014)

[11] Fadolalkarim, D., Bertino, E., Sallam, A.: An anomaly detection system for the protection of relational database systems against data leakage by application programs. In: Proc. IEEE ICDE. (2020)

[12] Ji, Y., Xu, C., Xu, J., Hu, H.: vabs: Towards verifiable attribute-based search over shared cloud data. In: Proc. IEEE ICDE '19. (2019)

[13] Yi, X., Paulet, R., Bertino, E., Varadharajan, V.: Practical approximate k nearest neighbor queries with location and query privacy. IEEE TKDE (2016)

[14] Commission, F.T.: How to Comply with the Privacy of Consumer Financial Information Rule of the Gramm-Leach-Bliley Act. Online at https://www.ftc.gov/tips-advice/business-center/guidance/how-comply-privacy-consumer-financial-information-rule-gramm (2000)

[15] of The European Parliament, R.E., of the Council: Chapter 4 Controller and processor. Online at https://gdpr-info.eu/chapter-4/ (2006)

[16] of The European Parliament, R.E., of the Council: Australian Privacy Principles quick reference. Online at https://www.oaic.gov.au/privacy/australian-privacy-principles/australian-privacy-principles-quick-reference (1988)

[17] Benny Chor, Oded Goldreich, E.K., Sudan, M.: Private information retrieval. In: 36th FOCS. IEEE Computer Society Press'95. (1995)

[18] Benny Chor, Oded Goldreich, E.K., Sudan, M.: Private information retrieval. Journal of the ACM (1998) 965–982

[19] Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: 38th FOCS. IEEE Computer Society Press'97. (1997)

[20] Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: EUROCRYPT'99. (1999)

[21] Kiayias, A., Leonardos, N., Lipmaa, H., Pavlyk, K., Tang, Q.: Optimal rate private information retrieval from homomorphic encryption. In: PET'15. (2015)

[22] Carlos Aguilar-Melchor, Joris Barrier, L.F.M.O.K.: Xpir: Private information retrieval for everyone. In: PET'16. (2016)

[23] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3) (1996) 431–473

[24] Zahur, S., Wang, X., Raykova, M., Gascón, A., Doerner, J., Evans, D., Katz, J.: Revisiting square-root oram: Efficient random access in multi-party computation. In: 2016 IEEE Symposium on Security and Privacy (SP). (2016)

[25] Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. (2012)

[26] Stefanov, E., van Dijk, M., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., et al.: Path ORAM: An Extremely Simple Oblivious RAM Protocol. In: ACM CCS'13

[27] Wang, X., Chan, H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In: ACM CCS'15. (2015)

[28] Doerner, J., Shelat, A.: Scaling oram for secure computation. In: ACM CCS'17. (2017)

[29] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious ram. In: UNSENIX Security'15. (2015)

[30] Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In: Euro S&P'19. (2019)

[31] Chakraborti, A., Sion, R.: Concuroram: High-throughput stateless parallel multi-client oram. In: NDSS'19. (2019)

[32] Chakraborti, A., Aviv, A.J., Choi, S., Mayberry, T., Roche, D.S., Sion, R.: roram: Efficient range oram with o(log2 n) locality. In: NDSS'19. (2019)

[33] Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: protecting confidentiality with encrypted query processing. In: Proc. ACM SOSP. (2011)

[34] microsoft SQL Server 2016: Always Encrypted (Database Engine). Online at `https://msdn.microsoft.com/en-us/library/mt163865.aspx/` (2016)

[35] Pappas, V., Vo, B., Krell, F., Choi, S., Kolesnikov, V., Keromytis, A., Malkin, T.: Blind Seer: A Scalable Private DBMS. In: Proc. IEEE S&P. (2014)

[36] Papadimitriou, A., Bhagwan, R., Chandran, N., Ramjee, R., Haeberlen, A., Singh, H., Modi, A., Badrinarayanan, S.: Big Data Analytics over Encrypted Datasets with Seabed. In: Proc. USENIX OSDI. (2016)

[37] Poddar, R., Boelter, T., Popa, R.A.: Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591 (2016)

[38] Yuan, X., Guo, Y., Wang, X., Wang, C., Li, B., Jia, X.: Enckv: An encrypted key-value store with rich queries. In: Proc. ACM AsiaCCS. (2017)

[39] Lewi, K., Wu, D.J.: Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In: Proc. ACM CCS. (2016)

[40] Chenette, N., Lewi, K., Weis, S.A., Wu, D.J.: Practical Order-Revealing Encryption with Limited Leakage. In: Proc. ACM FSE. (2016)

[41] Wu, S., Li, Q., Li, G., Yuan, D., Yuan, X., Wang, C.: ServeDB: Secure, Verifiable, and Efficient Range Queries on Outsourced Database. In: IEEE ICDE'19

[42] Poddar, R., Boelter, T., Popa, R.A.: Arx: An encrypted database using semantically secure encryption. Proc. VLDB Endow. **12** (2019) 1664–1678

[43] Dai, C., Yuan, X., Wang, C.: Privacy-preserving ridesharing recommendation in geosocial networks. In: Proc. of International Conference on Computational Social Networks, Springer (2016) 193–205

[44] Pattuk, E., Kantarcioglu, M., Khadilkar, V., Ulusoy, H., Mehrotra, S.: BigSecret: A secure data management framework for key-value stores. In: Proc. IEEE International Conference on Cloud Computing. (2013)

[45] Ishai, Y., Kushilevitz, E., Lu, S., Ostrovsky, R.: Private large-scale databases with distributed searchable symmetric encryption. In: Cryptographers' Track at the RSA Conference, Springer (2016) 90–107

[46] Popa, R.A., Li, F.H., Zeldovich, N.: An ideal-security protocol for order-preserving encoding. In: Proc. IEEE S& P. (2013)

[47] Zhang, H., Liu, X., Andersen, D.G., Kaminsky, M., Keeton, K., Pavlo, A.: Order-preserving key compression for in-memory search trees. In: SIGMOD. (2020)

[48] Meng, X., Zhu, H., Kollios, G.: Top-k query processing on encrypted databases with strong security guarantees. In: ICDE. (2018)

[49] Roche, D.S., Apon, D., Choi, S.G., Yerukhimovich, A.: Pope: Partial order preserving encoding. Technical report (2015) `http://eprint.iacr.org/2015/1106`.

[50] Mavroforakis, C., Chenette, N., O'Neill, A., Kollios, G., Canetti, R.: Modular order-preserving encryption, revisited. In: Proc. ACM SIGMOD, ACM (2015) 763–777

[51] Sundaram, N., Turmukhametova, A., Satish, N., Mostak, T., Indyk, P., Madden, S., Dubey, P.: Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. VLDB Endowment **6**(14) (2013) 1930–1941

[52] Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich Queries on Encrypted Data: Beyond Exact Matches. In: European Symposium on Research in Computer Security, Springer (2015) 123–145

[53] Naveed, M., Kamara, S., Wright, C.V.: Inference Attacks on Property-Preserving Encrypted Databases. In: ACM CCS'15. (2015)

[54] Bindschaedler, V., Grubbs, P., Cash, D., Ristenpart, T., Shmatikov, V.: The tao of inference in privacy-protected databases. VLDB Endowment **11**(11) (2018)

[55] Kamara, S., Moataz, T.: SQL on Structurally-Encrypted Databases. Cryptology ePrint Archive, Report 2016/453 (2016) `http://eprint.iacr.org/2016/453`.

[56] Zhang, Y., Katz, J., Papamanthou, C.: Integridb: Verifiable SQL for Outsourced Databases. In: Proc. ACM CCS. (2015)

[57] Li, R., Liu, A.X., Wang, A.L., Bruhadeshwar, B.: Fast Range Query Processing with Strong Privacy Protection for Cloud Computing. Proc. VLDB Endow. **7**(14) (2014) 1953–1964

[58] Ren, K., Wang, C., Wang, Q., et al.: Security challenges for the public cloud. IEEE Internet Computing **16**(1) (2012) 69–73

[59] Tang, X., Wang, C., Yuan, X., Wang, Q.: Non-interactive privacy-preserving truth discovery in crowd sensing applications. In: Proc. IEEE INFOCOM. (2018)

[60] Yuan, X.: Private and versatile search over very large encrypted datasets. Online at `https://scholars.cityu.edu.hk/en/theses/private-and-versatile-search-over-very-large-encrypted-datasets.html` (2016)

[61] Yuan, X., Yuan, X., Wang, C., Li, B.: Secure multi-client data access with boolean queries in distributed key-value stores. In: Proc. of IEEE CNS. (2017)

[62] Zhu, J., Li, Q., Wang, C., Yuan, X., Wang, Q., Ren, K.: Enabling generic, verifiable, and secure data search in cloud services. IEEE TPDS (2018)

[63] Yuan, X., Weng, J., Wang, C., Ren, K.: Secure integrated circuit design via hybrid cloud. IEEE TPDS (2018)

[64] Yuan, X., Wang, X., Wang, C., Squicciarini, A.C., Ren, K.: Towards privacy-preserving and practical image-centric social discovery. IEEE TDSC (2016)

[65] Song, D., Wagner, D., Perrig, A.: Practical Techniques for Searches on Encrypted Data. In: IEEE S&P'00

[66] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable Symmetric Eencryption: Improved Definitions and Efficient Constructions. In: ACM CCS'06

[67] Kurosawa, K., Ohtaki, Y.: Uc-secure searchable symmetric encryption. In Keromytis, A.D., ed.: Financial Cryptography and Data Security. (2012)

[68] Asharov, G., Segev, G., Shahaf, I.: Tight tradeoffs in searchable symmetric encryption. J. Cryptol. **34**(2) (2021) 9

[69] Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM CCS'12. (2012)

[70] Cash, D., Jaeger, J., Jarecki, S., Jutla, C.: Dynamic Searchable Encryption in Very Large Databases: Data Structures and Implementation. In: NDSS'14

[71] Eskandarian, S., Zaharia, M.: Oblidb: Oblivious query processing for secure databases. VLDB Endowment **13**(2) (2019)

[72] Liu, Z., Li, B., Huang, Y., Li, J., Xiang, Y., Pedrycz, W.: Newmcos: Towards a practical multi-cloud oblivious storage scheme. IEEE TKDE **32**(4) (2020)

[73] Islam, M., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: Proc. Network and Distributed System Security Symposium. (2012)

[74] Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proc. of ACM CCS, ACM (2015) 668–679

[75] Bost, R., Fouque, P.A.: Thrawting leakage abuse attacks againts searchable encryption a formal approach and applications to database padding. Cryptology ePrint Archive, Report 2017/1060 (2017) https://eprint.iacr.org/2017/1060.

[76] Zhang, Y., Katz, J., Papamanthou, C.: All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In: USENIX Security'16

[77] Stefanov, E., Papamanthou, C., Shi, E.: Practical Dynamic Searchable Symmetric Encryption with Small Leakage. In: NDSS'14. (2014)

[78] Bost, R.: Sophos - Forward Secure Searchable Encryption. In: ACM CCS'16

[79] Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. Privacy Enhancing Technologies **2018**(1) (2018) 5–20

[80] Ghareh Chamani, J., Papadopoulos, D., Papamanthou, C., Jalili, R.: New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In: ACM CCS'18

[81] Sun, S.F., Yuan, X., Liu, J., Steinfeld, R., Sakzad, A., Vo, V., et al.: Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In: ACM CCS'18. (2018)

[82] Bost, R., Minaud, B., Ohrimenko, O.: Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In: ACM CCS'17. (2017)

[83] Amjad, G., Kamara, S., Moataz, T.: Forward and Backward Private Searchable Encryption with SGX. In: EuroSec'19

[84] Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptol. ePrint Arch. (2016)

[85] Garg, S., Mohassel, P., Papamanthou, C.: Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In: CRYPTO'16

[86] Chang, Y.C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. ACNS'05

[87] Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Proc. ASIACRYPT. (2010)

[88] Cash, D., Tessaro, S.: The Locality of Searchable Symmetric Encryption. In: EUROCRYPT'14. (2014)

[89] Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Proc. of ACM CCS, ACM (2014) 310–320

[90] Ian, M., Payman, M.: Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. In: Proc. of NDSS, The Internet Society (2017)

[91] Demertzis, I., Papamanthou, C.: Fast Searchable Encryption with Tunable Locality. In: ACM SIGMOD'17

[92] Asharov, G., Naor, M., Segev, G., Shahaf, I.: Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In: Proc. ACM SIGACT, ACM (2016) 1101–1114

[93] Demertzis, I., Papadopoulos, D., Papamanthou, C.: Searchable Encryption with Optimal Locality: Achieving Sublogarithmic Read Efficiency. In: CRYPTO'18

[94] Demertzis, I., Talapatra, R., Papamanthou, C.: Efficient Searchable Encryption Through Compression. Proc. VLDB Endow. (2018)

[95] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In: CRYPTO'13. (2013)

[96] Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: Proc. EUROCRYPT, Springer (2017)

[97] Zuo, C., Sun, S.F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption schemes supporting range queries with forward (and backward) security. In: ESORICS'18

[98] Zuo, C., Sun, S.F., Liu, J., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption with forward and stronger backward privacy. In: ESORICS'19

[99] Yuan, X., Wang, X., Wang, C., Yu, C., Nutanong, S.: Privacy-preserving similarity joins over encrypted data. IEEE TIFS **12**(11) (2017) 2763–2775

[100] Liu, X., Yuan, X., Wang, C.: Encsim: An encrypted similarity search service for distributed high-dimensional datasets. In: Proc. IEEE/ACM IWQoS. (2017)

[101] Yuan, X., Cui, H., Wang, X., Wang, C.: Enabling privacy-assured similarity retrieval over millions of encrypted records. In: Proc. ESORICS. (2015)

[102] Borges, G., Domingos, H., Ferreira, B., Leitão, J., Oliveira, T., Portela, B.: BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage. In: SRDS'19

[103] Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: NDSS. (2021)

[104] Sun, S.F., Liu, J.K., Sakzad, A., Steinfeld, R., Yuen, T.H.: An efficient non-interactive multi-client searchable encryption with support for boolean queries. In: Proc. ESORICS. (2016)

[105] Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A., Garofalakis, M.: Practical Private Range Search Revisited. In: ACM SIGMOD'16. (2016)

[106] Ren, K., Guo, Y., Jiaqi, L., Jia, X., Wang, C., Zhou, Y., Wang, S., Cao, N., Li, F.: Hybridx: New hybrid index for volume-hiding range queries in data outsourcing services. In: ICDCS'20

[107] Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-Abuse Attacks against Searchable Encryption. In: ACM CCS'15. (2015)

[108] Xu, L., Yuan, X., Wang, C., Wang, Q., Xu, C.: Hardening database padding for searchable encryption. In: Proc. IEEE INFOCOM. (2019)

[109] Grubbs, P., Lacharite, M.S., Minaud, B., Paterson, K.G.: Pump up the volume: Practical database reconstruction from volume leakage on range queries. In: CCS '18. (2018)

[110] Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: NDSS. (2020)

[111] Kamara, S., Moataz, T.: Computationally volume-hiding structured encryption. In: EUROCRYPT19. (2019)

[112] Patel, S., Persiano, G., Yeo, K., Yung, M.: Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In: ACM CCS. (2019)

[113] Song, X., Dong, C., Yuan, D., Xu, I., Zhao, M.: Forward private searchable symmetric encryption with optimized i/o efficiency. IEEE TDSC (2018)

[114] Li, J., Huang, Y., Wei, Y., Lv, S., Liu, Z., Dong, C., Lou, W.: Searchable symmetric encryption with forward search privacy. IEEE Transactions on Dependable and Secure Computing (2021)

[115] Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.H.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Proc. ACM CCS, ACM (2017)

[116] Microsoft Azure: Azure confidential computing. Online at `https://azure.microsoft.com/en-us/solutions/confidential-compute/` (2020)

[117] Christian, P., Kapil, V., Manuel, C.: EnclaveDB: A Secure Database using SGX. In: IEEE S&P'18

[118] Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: An Efficient Oblivious Search Index. In: IEEE S&P'18

[119] Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.: HardIDX: Practical and Secure Index with SGX. In: DBSec'17

[120] Stefanov, E., Papamanthou, C., Shi, E.: Practice dynamic searchable encryption with small leakage. In: Proc. of NDSS, The Internet Society (2014)

[121] Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M., Steiner, M.: Outsourced Symmetric Private Information Retrieval. In: Proc. ACM CCS. (2013)

[122] Zhang, Z.: Statistical Implications of Turing's Formula. Wiley, New Jersey (2017)

[123] Zhang, Y., Tangwongsan, K., Tirthapura, S.: Streaming k-means clustering with fast queries. ICDE (2017)

[124] Gomes, H.M., Read, J., Bifet, A., Barddal, J.P., Gama, J.a.: Machine learning for streaming data: State of the art, challenges, and opportunities. SIGKDD Explor. Newsl. (2019)

[125] Blackstone, L., Kamara, S., Moataz, T.: Revisiting Leakage Abuse Attacks. In: NDSS. (2020)

[126] Lei, X., Liu, A.X., Li, R., Tu, G.: SecEQP: A Secure and Efficient Scheme for SkNN Query Problem Over Encrypted Geodata on Cloud. In: IEEE ICDE'19

[127] Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic Attacks on Secure Outsourced Databases. In: Proc. ACM CCS. (2016)

[128] Duan, H., Wang, C., Yuan, X., Zhou, Y., Wang, Q., Ren, K.: LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed. In: ACM CCS'19

[129] Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security'14

[130] Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT'17

[131] Shinde, S., Chua, Z.L., Narayanan, V., Saxena, P.: Preventing Page Faults from Telling Your Secrets. In: ACM AsiaCCS'16

[132] Taassori, M., Shafiee, A., Balasubramonian, R.: VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In: ACM ASPLOS'18

[133] McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., Rozas, C.: Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In: HASP'16. (2016)

[134] Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache Attacks on Intel SGX. In: EuroSec'17

[135] Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: Using sgx to conceal cache attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment. (2017)

[136] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In: CCS '17. (2017)

[137] Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: IEEE S&P'15

[138] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In: USENIX Security'18. (2018)

[139] Biondo, A., Conti, M., Davi, L., Frassetto, T., Sadeghi, A.R.: The guard's dilemma: Efficient code-reuse attacks against intel sgx. In: USENIX Security'18. (2018)

[140] Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in darkness: Return-oriented programming against secure enclaves. In: USENIX Security'17. (2017)

[141] Murdock, K., et al.: Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: IEEE S&P'20. (2020)

[142] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of aes. In: Topics in Cryptology – CT-RSA 2006. (2006)

[143] Marshall, A., Howard, M., Bugher, G., Harden, B.: Security best practices for developing windows azure applications. Microsoft Corp (2010)

[144] Brasser, F., Capkun, S., Dmitrienko, A., Frassetto, T., Kostiainen, K., Sadeghi, A.R.: DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In: ACSAC'19

[145] Gruss, D., Lettner, J., Schuster, F., Ohrimenko, O., Haller, I., Costa, M.: Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In: UNSENIX Security'17

[146] Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing digital side-channels through obfuscated execution. In: USENIX Security'15. (2015)

[147] Sasy, S., Gorbunov, S., Fletcher, C.W.: Zerotrace : Oblivious memory primitives from intel sgx. In: (NDSS)'18. (2018)

[148] Ohrimenko, O., Schuster, F., Fournet, C., Mehta, A., Nowozin, S., Vaswani, K., Costa, M.: Oblivious multi-party machine learning on trusted processors. In: USENIX Security'16. (2016)

[149] Oleksenko, O., Trach, B., Krahn, R., Martin, A., Fetzer, C., Silberstein, M.: Varys: Protecting SGX Enclaves from Practical Side-channel Attacks. In: USENIX ATC'18

[150] Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-sgx: Eradicating controlled-channel attacks against enclave programs. In: NDSS. (2017)

[151] Chen, S., Zhang, X., Reiter, M.K., Zhang, Y.: Detecting privileged side-channel attacks in shielded execution with déjà vu. In: ASIA CCS'17. (2017)

[152] Fu, Y., Bauman, E., Quinonez, R., Lin, Z.: Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In: RAID. (2017)

[153] Sinha, R., Rajamani, S., Seshia, S.A.: A compiler and verifier for page access oblivious computation. In: ESEC/FSE 2017. (2017)

[154] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: Kaslr is dead: Long live kaslr. In: ESSoS. (2017)

[155] Cutress, I.: Analyzing Core i9-9900K Performance with Spectre and Meltdown Hardware Mitigations. Intel Corp (2018) `https://www.anandtech.com/`.

[156] Zhao, W., Lu, K., Qi, Y., Qi, S.: Mptee: Bringing flexible and efficient memory protection to intel sgx. In: EuroSys'20. (2020)

[157] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., Fetzer, C.: Sgxbounds: Memory safety for shielded execution. In: EuroSys'17. (2017)

[158] Seo, J., Lee, B., Kim, S., Shih, M.W., Shin, I., Han, D., Kim, T.: Sgx-shield: Enabling address space layout randomization for sgx programs. In: NDSS. (2017)

[159] Cloosters, T., Rodler, M., Davi, L.: Teerex: Discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In: USENIX Security'20. (2020)

[160] Intel: Intel processors voltage settings modification advisory. Online at `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html` (2020)

[161] Cai, C., Yuan, X., Wang, C.: Towards trustworthy and private keyword search in encrypted decentralized storage. In: Proc. of IEEE ICC. (2017)

[162] Yuan, X., Wang, X., Wang, C., Squicciarini, A., Ren, K.: Enabling privacy-preserving image-centric social discovery. In: Proc. IEEE ICDCS. (2014)

[163] Cui, H., Yuan, X., Wang, C.: Harnessing encrypted data in cloud for secure and efficient image sharing from mobile devices. In: Proc. IEEE INFOCOM. (2015)

[164] Cui, H., Yuan, X., Wang, C.: Harnessing encrypted data in cloud for secure and efficient mobile image sharing. IEEE TMC **16**(5) (2017) 1315–1329

[165] Zheng, Y., Cui, H., Wang, C., Zhou, J.: Privacy-preserving image denoising from external cloud databases. IEEE TIFS **12**(6) (2017) 1285–1298

[166] Yuan, X., Wang, C., Ren, K.: Enabling ip protection for outsourced integrated circuit design. In: Proc. ACM AsiaCCS. (2015)

[167] Yuan, X., Wang, X., Lin, J., Wang, C.: Privacy-preserving deep packet inspection in outsourced middleboxes. In: Proc. IEEE INFOCOM. (2016)

[168] Wang, C., Yuan, X., Cui, Y., Ren, K.: Towards secure outsourced middlebox services: Practices, challenges, and beyond. IEEE Network Magazine (2017)

[169] Cai, C., Yuan, X., Wang, C.: Hardening distributed and encrypted keyword search via blockchain. In: Proc. of IEEE PAC. (2017)

[170] Zheng, W., Deng, R., Chen, W., Popa, R.A., Panda, A., Stoica, I.: Cerebro: A platform for multi-party cryptographic collaborative learning. In: USENIX Security'16

[171] Poddar, R., Kalra, S., Yanai, A., Deng, R., Popa, R.A., Hellerstein, J.: Senate: A maliciously-secure mpc platform for collaborative analytics. In: USENIX Security'20

[172] Dauterman, E., Feng, E., Luo, E., Popa, R.A., Stoica, I.: Dory: An encrypted search system with distributed trust. In: OSDI'20

[173] Zheng, W., Popa, R.A., Gonzalez, J., Stoica, I.: Helen: Maliciously secure coopetitive learning for linear models. 2019 IEEE Symposium on Security and Privacy (SP) (2019) 724–738