SEC: Fast Private Boolean Circuit Evaluation from Encrypted Look-ups

Debadrita Talapatra
IIT Kharagpur, India
debadritat.fg2219@gmail.com

Nimish Mishra
IIT Kharagpur, India
neelam.nimish@gmail.com

Arnab Bag
IIT Kharagpur, India
amiarnabbolchi@gmail.com

Sikhar Patranabis
IBM Research India
sikharpatranabis@gmail.com

Debdeep Mukhopadhyay
IIT Kharagpur, India
debdeep.mukhopadhyay@gmail.com

Abstract

Encrypted computation has over the past thirty years, turned into one of the holy grails of modern cryptography especially with the advent of cloud computing. Modern cryptographic techniques like Fully Homomorphic Encryption (FHE) allow arbitrary Boolean circuit evaluation with encrypted inputs. However, the prohibitively high computation and storage overhead coupled with high communication bandwidth of FHE severely limit its scalability in practical applications like real-time analytics or machine learning inference. In summary, the current cryptographic literature lacks robust and scalable methods for efficient encrypted computation in practical outsourced applications.

In this work, we introduce a new approach for encrypted computation called SEC (Symmetric Encryption-based Computation) which offers fast Boolean circuit evaluation with optimal storage and communication overhead while scaling smoothly to real applications. SEC relies on an efficient Searchable Symmetric Encryption (SSE) construction to leverage the power of encrypted lookups in Boolean circuit evaluation. SEC is specifically suited for client-server systems, and the server, honest-but-curious receives the client's encrypted inputs and outputs the encrypted evaluation result while leaking only benign information to the server. SEC essentially extends the capabilities of SSE schemes from searching over encrypted databases to arbitrary function evaluation over encrypted inputs. SEC supports Boolean function composition, allowing it to evaluate complex functions efficiently without blowing up storage overhead. SEC outperforms the state-of-the-art FHE, namely, Torus FHE (TFHE) scheme with an average $10^3\times$ speed-up in basic Boolean gate evaluations. We present a prototype implementation of SEC and experimentally validate its practical efficiency. Our experiments show that SEC executes arbitrary depth Boolean circuit in a single round of communication between client and server with a significant improvement in performance than the fastest TFHE backends. We exemplify the applicability of our scheme by implementing one byte AES SBox using SEC and comparing the results with TFHE.

Contents

1 Introduction

Outsourced Computation. In modern digital infrastructures, outsourced data processing has gained significant attention from government organizations, industries, and academia. The advancement of cloud computing and other connected applications like the Internet of Things (IoT) involves storage and processing of sensitive data of millions of users. The majority of these tasks are handled by third-party manufacturers and cloud service providers who can access the associated data - be it sensor readings from home automation devices or large-scale enterprise servers executing huge machine learning models over medical data. This typically raises privacy concerns about the users' sensitive information as untrusted parties can gain access to sensitive information while processing outsourced data.

The densely connected network of low-cost embedded systems in several applications, including IoT, home automation, medical devices, and production lines etc provides an effective and robust way of managing and controlling essential systems. However, the limited computation capability of the end-point devices often calls for an asymmetric computation overhead. In this way, computationally heavy tasks can be offloaded to powerful back-end computing infrastructure, and the low-end embedded devices process computationally light tasks. Such task distribution is critical for real-time and high-throughput applications, such as biometric scanners for door locks or machine learning (ML) based driver assistant systems in cars. This asymmetric task distribution is adopted in state-of-the-art computation paradigms such as *edge computing*, and is demonstrated in multiple applications such as ML inferences [1, 2], health-care [3, 4, 5], and industrial production lines [6, 7, 8].

The end devices in the aforementioned applications rely on the back-end infrastructure for full system operation, which executes heavy computational tasks. With the adoption of modern cloud-based remote computing platforms, the back-end computation services are deployed (outsourced) on third-party cloud platforms. Naturally, without specific privacy mechanisms, the third-party remote servers can "see" all user data sent to the server for processing, thus leading to serious privacy concerns. In recent years, multiple attacks [9, 10] have been demonstrated targeting such systems in practice, essentially pushing towards the adoption of secure and scalable privacy mechanisms for protecting sensitive data from unauthorised access.

Privacy-preserving Computation Framework. While developing use-case centric privacy preserving solutions is one option, it is often restrictive because of the need to depend upon the specifics of the underlying offloaded computation as well. For instance, a privacy-preserving machine learning inference scheme [11, 12, 13] needs to consider the underlying machine learning algorithms in its design. As such, these solutions cannot be considered as *generic* privacy preserving compute mechanisms. Therefore, in recent years, much attention has been given to developing generic frameworks of arbitrary computation over encrypted data, so that such mechanisms support application agnostic privacy preserving capabilities. There exists in literature a number of elegant cryptographic solutions[1] that provide a secure way of computing over outsourced encrypted data, such as - Fully Homomorphic Encryption (FHE) [16] Oblivious Random Access Memory model (ORAM)

[1]Note that we do not consider implementations like [14] that rely on hardware assumptions to achieve this objective, because such assumptions are challenged by platform dependent side-channel attack vectors [15].

[17], Multi-party Computation (MPC) [18, 19], and Functional Encryption (FE) [20]. However, each has limitations from a practicality viewpoint [21]. For instance, ORAM promises oblivious memory access patters, but is hard to actualize in hardware, closed-source, and not tested against scaled databases [22]. Likewise, MPC involves functional computation on data disjointly owned by multiple clients, which is completely different threat model considered in this work. Finally, while FE provides fine-grained query processing over encrypted data, its practical efficiency is much worse than FHE and is unsuitable for deployment. While these FE based techniques offer an "ideal" notion of privacy, they incur high computation cost, communication bandwidth, and storage overhead. This essentially prohibits the adoption of these techniques in real cloud applications. Consequently, amongst these, FHE (even with limitations which we expound next) is currently the most practical alternative with real-world adoption. FHE thus becomes the basis of comparison with our proposal.

Fully Homomorphic Encryption (FHE). FHE has recently gained traction in the cryptographic literature for privacy-preserving computation with rich functionalities. Due to the *ideal* notion of privacy, FHE is widely researched and actively developed for practical adoption. Introduced by Gentry et al. [16], FHE achieves critical functional homomorphism properties necessary for encrypted processing and outsourced computation workloads.

Practical Limitations. The scheme by Gentry [16] is based of Learning-With-Errors (LWE) problem [23], which incurs incremental noise overhead with number of operations. This growing noise essentially restricts the number of encrypted evaluations before the output is corrupted leading to incorrect decryption or result. Gentry's scheme handled this noise growth by introducing *bootstrapping* that refreshes the ciphertext to reduce the amount of noise. However, bootstrapping is computationally extremely heavy and incurs enormous storage overhead that prohibits deploying FHE in real-time and critical workloads.

A number of works have attempted algorithmic improvements [24, 25, 26, 27, 28, 29, 30, 31, 32] and implementation-oriented optimisations [33, 34, 35, 36, 37, 38] which still do not attain real-time performance and optimal storage overhead. The state-of-the-art TFHE construction by Chillotti et al. [32] achieve fast sub-second bootstrapping time in practice. However, as each primitive function evaluation is preceded by bootstrapping, it results in prohibitively high computation delay. Combined with the huge storage overhead, FHE remains unsuitable for practical privacy-preserved computation to date.

While FHE offers a more direct approach for function evaluation over encrypted data, reducing the bootstrapping overhead certainly improves the performance. However, despite these clever and heavy optimisations, the actual improvement is insufficient and FHE still remains practically unsuitable for real workloads due to costly bootstrapping. Clearly, bypassing the bootstrapping phase in FHE would certainly open up the bottleneck and one possible way to achieve that by *not explicitly performing the function evaluation over encrypted inputs*. Rather, a more fundamental approach can be adopted by considering pre-computed primitive functions and use these is a secure way. In more general way, the FHE framework can be replaced by a lookup based approach that does not require any bootstrapping mechanism. We summarise the goal of this work by asking the following question.

Can we build an efficient and generic computation framework from encrypted lookup of basic primitives that potentially supports arbitrary computation depth?

In this work, we present our construction SEC that achieves the aforementioned goals. To elucidate more on the core idea of our construction, we detail the fundamental approach using a basic computation operation - a full adder, which we discuss below.

Function Evaluation via lookup. We consider the circuit model of computation in this discussion. we assume a poly-time function of n variables can be equivalently expressed as a poly-sized circuit of n variables. Equivalently any Boolean function representing a circuit can be decomposed into basic Boolean operations or gates operating on input variables. Consider a simple 1-bit full adder adding two 1-bit values A, B with carry C_{in}, without loss of generality.

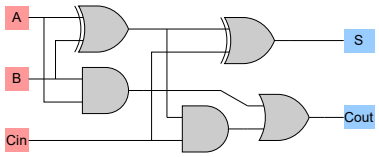

Figure 1: 1-bit full adder circuit comprising of two-input logic gates

Structurally, a full adder can be expressed by circuit presented in Figure 1, where the basic logic primitives {`AND,OR,XOR`} are evaluated on the respective input values. In this approach, the basic logic operations are *computed* on the inputs values. However, these computations can be replaced with corresponding lookup tables (LUT) encoding each logic operation. Thus, when the basic logic operations of the circuit is replaced with LUTs, the transformed circuit evaluates the same function (as shown in Figure 2). Since a lookup operation is typically faster than a computation, the overall function evaluates faster with lookup in physical implementations[2].

In applications, where the data is not encrypted, look-ups do not bring huge efficiency improvement over primitive computations. However, in an encrypted computation framework, the primitive computations are typically implemented using costly primitives like FHE, and thus affects the overall computation performance adversely. In contrast, encrypted look-ups are faster than primitive encrypted computations, and as a result, the overall computation time required reduces significantly for encrypted lookup based approach. Thus, when a basic logic circuit is used in encrypted computation application, the transformed circuit with encrypted lookup (Figure 3) offers faster computation time compared to usual encrypted computation based approaches such as FHE or MPC.

However, encrypted look-ups still need to be performed in a privacy-preserving way. This implies that generic lookup techniques for unencrypted data can not be directly used in our context. Hence we rely on another class of cryptographic schemes - Searchable Symmetric Encryption (SSE) schemes [39, 40, 41, 42] which provision users with *search* capabilities over symmetrically encrypted data. For example, consider a client that offloads an encrypted database of (potentially sensitive) emails to an untrusted server and later issues a *conjunctive* query of the form "*retrieve all emails received from xyz@foobar.org with the keyword "research" in the subject field*" (which is a conjunction of the queries "*retrieve all emails received from xyz@foobar.org* or "*retrieve all emails with the keyword "research" in the subject field*"). For any SSE scheme to be truly practical, it should at least support

[2]Note that, this difference may not be visible for small functions, but as the depth of the circuit increases, the lookup based evaluation is faster than straightforward computation

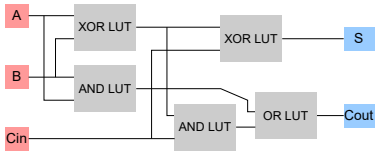

Figure 2: 1-bit full adder circuit - logic gates replaced with LUT

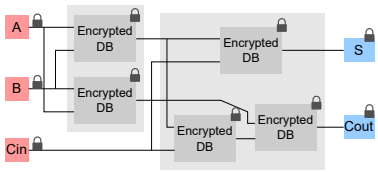

Figure 3: 1-bit full adder circuit - logic gates replaced with encrypted lookup.

such conjunctive keyword queries, i.e., given a set of keywords $(\mathbf{w}_1, \ldots, \mathbf{w}_n)$, it should be able to find and return the set of documents that contain *all* of these keywords. There exist today efficient SSE schemes that support conjunctive (and more general Boolean) queries [42, 43, 44]. Modern SSE schemes trade-off security for efficiency. Concretely, these schemes allow the server to learn "some" information during query execution, referred to as *leakage*. While SSE schemes are extremely fast and highly scalable with arbitrarily large real-world datasets, their restricted functionality renders them inapt for practical deployment in an encrypted computation framework.

In this work, we focus on developing an encrypted computation framework that internally uses encrypted look-ups for Boolean function evaluation by leveraging the highly efficient *search* capability of a conjunctive SSE scheme.

1.1 Our Contributions

In this work, we propose a novel approach towards developing an efficient and coherent *generic* encrypted computation framework called SEC which offers fast evaluation of arbitrary Boolean circuits with optimal storage and communication overhead while scaling smoothly to real-world applications. SEC leverages the computation efficiency and optimal storage overhead of a conjunctive SSE scheme for encrypted look-ups. The technical centerpiece of SEC lies in bypassing the explicit circuit evaluation (as in FHE) while evaluating a function over encrypted data. Instead the function is evaluated via an efficient search mechanism rendered by a conjunctive SSE scheme over encrypted lookup tables. To the best of our knowledge SEC *is the first encrypted computation framework capable of arbitrary function evaluation over encrypted data using fast and efficient look-ups.* We list the main technical achievements of this work below.

Fast Encrypted Computation. SEC relies on "encoding" a computation as a series of encrypted lookup tables of primitive operations, and use a fast encrypted search mechanism for necessary look-ups. This way, SEC bypasses explicit computation via look-ups, and does not incur heavy overhead similar to bootstrapping in FHE. Since, any function computation can be modeled into an equivalent logic circuit of Boolean variables and basic logic gates -

AND, OR, XOR, creating the encrypted lookup tables for these basic gates allows to evaluate the function in SEC. For fast lookup over encrypted databases (the lookup tables of primitive operations), we resort to state-of-the-art efficient adaptively secure SSE algorithms. In our case, we selected Oblivious Cross Tags (OXT) by Cash et al. [45] as the underlying conjunctive SSE scheme. Thus combining encrypted look-ups with SSE, allows extremely fast circuit evaluation (circuits representing functions), which is significantly faster than FHE with bootstrapping.

Supporting Function Composition. A pivotal feature of SEC is the support for function composition. The ability to evaluate a function composition is the key to evaluate an arbitrary function in SEC. An arbitrary function can be decomposed into an expression of function composition with lesser number of variables (following Shannon's theory, for Boolean circuits). Thus, a complex function can be easily evaluated systematically using SEC with optimal computation overhead proportional to the circuit size. At the same time, SEC incurs nominal storage overhead for encrypted lookup tables compared to huge storage required to store bootstrapping key. Furthermore, SEC supports arbitrary function evaluations in single round of communication with the expense of a small amount of additional storage, which is practically ideal for outsource computations and leaks less compared to a multi-round solution.

Concrete Security Analysis and Implementation. SEC relies on an efficient adaptively secure conjunctive SSE scheme for the encrypted lookup. Thus, SEC inherits adaptive security properties of the underlying SSE construction, in this case OXT, to achieve adaptive security as well. We present detailed security and leakage analysis of SEC following the security properties of OXT. We discuss the leakage profile and security analysis of SEC in Section 5. Furthermore, we demonstrate the efficacy of our framework by evaluating basic Boolean gates and cascaded gates as function composition. Section 6 gives a detailed analysis of our experimental evaluations.

We start with a basic version of the SEC framework called $\mathsf{SEC}_{\mathsf{Basic}}$ in Section 3. $\mathsf{SEC}_{\mathsf{Basic}}$ introduces the core technique and outlines the main algorithmic routines instantiated from conjunctive SSE subroutines. However, the basic $\mathsf{SEC}_{\mathsf{Basic}}$ construction does not support arbitrary function evaluation. Subsequently, we augment this $\mathsf{SEC}_{\mathsf{Basic}}$ with specific strategies to develop the final construction SEC that supports function composition evaluation. We outline the final construction is Section 4 which discusses the computation flow in detail.

1.2 Technical Challenges and Proposed Solutions

The fundamental technical novelty of SEC is fast and efficient encrypted computation via encrypted look-ups rendered by leveraging search capabilities of a conjunctive SSE scheme thereby bypassing the explicit circuit evaluation (as in FHE). While designing such a fast and generic privacy-preserving framework we encountered a number of technical challneges. We provide dedicated solutions to these challenges by proposing our novel techniques most crucial being encrypted computation via lookups. For simplicity of exposition, we first explain the technical challenges and our core solution ideas for the design of XOR function, without loss of generality. The designs for AND and OR follow suit. The objective of the entire design is to allow evaluation of a 2-bit XOR through encrypted lookup tables. That is,

given *encryptions* of two bits x and y, SEC uses a conjunctive SSE scheme which returns a single *encrypted* document which upon decryption outputs 0/1 in accordance with the actual value of $\texttt{XOR}(x, y)$.

Challenge 1: Keyword mapping. A generic SSE scheme maps alphanumeric strings as *keywords* and performs search queries on the encrypted database. However, evaluation of an arbitrary function requires logical bit inputs. Hence, there needs to be a *mapping* between one-bit inputs to the function and alphanumeric keywords that SEC can delegate for encrypted look-ups using an SSE scheme. However, a *generic* SSE construction does not support such a conversion.

Our Solution. We propose our novel solution with an example. Without loss of generality, we consider the example of XOR being computed by $\text{SEC}_{\mathsf{Basic}}$ (SEC performs such binary function evaluation by invoking $\text{SEC}_{\mathsf{Basic}}$). For arbitrary input bits $x, y \in \{0, 1\}$, we want $\text{SEC}_{\mathsf{Basic}}$ to compute $\texttt{XOR}(x, y)$. $\text{SEC}_{\mathsf{Basic}}$ thus first chooses alphanumeric strings $\mathbf{w}_1$ and $\mathbf{w}_2$ to serve as keywords. However, unlike a generic SSE scheme [46, 47, 48] where keywords are a part of the encrypted documents' contents, the keywords in $\text{SEC}_{\mathsf{Basic}}$ are instead encoding for bits x and y. As an example, $\mathbf{w}_1$ may be treated as an encoding for x (likewise the relationship between $\mathbf{w}_2$ and y). Thus, if $x = 0$, $\text{SEC}_{\mathsf{Basic}}$ looks for *absence* of the keyword $\mathbf{w}_1$ in the encrypted database (likewise for $x = 1$ where *presence* of $\mathbf{w}_1$ matters). Assume that an encrypted document subset $D^{\mathsf{w}_1} = \{D_1^{\mathsf{w}_1}, D_2^{\mathsf{w}_1}, D_3^{\mathsf{w}_1}, ..., D_n^{\mathsf{w}_1}\}$ is returned upon querying for w_1. Likewise, assume an encrypted document set $D^{\mathsf{w}_2} = \{D_1^{\mathsf{w}_2}, D_2^{\mathsf{w}_2}, D_3^{\mathsf{w}_2}, ..., D_n^{\mathsf{w}_2}\}$ is returned upon a query for w_2. The final result of $\text{SEC}_{\mathsf{Basic}}$ is the intersection $D^{\mathsf{w}_1} \cap D^{\mathsf{w}_2}$.

The technical centerpiece of our proposed scheme is to choose two alphanumeric strings w_1 and w_2, wherein w_1 is logically equivalent to an input x to XOR (likewise for w_2 being equivalent to the input y to XOR). Without loss of generality, assume $x = 1$. When SEC invokes an underlying generic conjunctive SSE, it includes w_1 in its search query thereby extracting all encrypted documents in which the keyword w_1 is *present*. In contrast, if $x = 0$, then SEC does *not* include w_1 in its search query, thereby seeking to extract all encrypted documents in which the keyword w_1 is *absent*.

Challenge 2: Absence of Keyword. We emphasise upon the fact that, *absence* of a keyword is tricky to handle for a conjunctive SSE scheme. Firstly, a *generic* conjunctive SSE is exclusively constructed to operate upon *presence* of keywords in the encrypted database. If keywords are *absent* from a subset of documents, such documents are never returned by the search query. Consequently, this means that using a *generic* conjunctive SSE instead of SEC would not return the correct set of documents when $x = 0$ and/or $y = 0$. Moreover, modern conjunctive SSE designs take a performance hit if searches are done for *absence* of keywords [42]. This is because *absence* of a keyword is logically equivalent to *presence* of *any* combination of other keywords. Depending on the total number of keywords in database under consideration, an abnormally large number of encrypted documents may be retrieved, causing major performance bottleneck in SSE's execution.

Our Solution. To solve this problem, we introduce the novel concept of an *inversion* map I. An inversion map I is a one-to-one map on the set of alphanumeric strings such that SEC can denote the *absence* of some keyword w_1 as *presence* of another *distinct* keyword

$I(\mathsf{w}_1)$. Note that I is chosen in a way to avoid collisions like $\mathsf{w}_1 \neq I(\mathsf{w}_1)$. Hence, the keyword mapping SEC introduces has exactly four *distinct* keywords- $\{\mathsf{w}_1, I(\mathsf{w}_1), \mathsf{w}_2, I(\mathsf{w}_2)\}$- associated with the input bits x and y of `XOR`. Concretely, if $x = 0$, the search query looks for documents in the encrypted lookup table in which the keyword $I(\mathsf{w}_1)$ is present. Likewise, for $y = 0$, the search proceeds with looking for the *presence* of the keyword $I(\mathsf{w}_2)$ in the encrypted lookup table.

Challenge 3: Mapping Primitive Operations to Encrypted Lookup Table . In a *generic* conjunctive SSE scheme, an arbitrary number of documents can be matched with a *search* query. In our case however, we desire exactly *one* document to be matched, such that for appropriate bit inputs x and y, SEC gives a deterministic evaluation of `XOR`(x, y). The encrypted database in $\mathrm{SEC}_{\mathsf{Basic}}$ should act as an *encrypted lookup* table for the function `XOR` (similarly for {`AND, OR`}). Therefore, to ensure functionally correct evaluation of the `XOR` it is necessary to ensure that the result of the lookup is always a singleton set, consisting of a single document. This is not trivially guaranteed by the underlying SSE scheme.

Our Solution: SEC guarantees correctness of a function evaluation by ensuring $D^{\mathsf{w}_1} \cap D^{\mathsf{w}_2}$ is a singleton that corresponds to the actual output of `XOR`(x, y). We create such an encrypted lookup table for `XOR` by assigning *exactly* one document to every possible combination of w_1 and w_2. Thereby, by extension of the keyword mapping already discussed, there is *exactly* one encrypted document against all four combinations of x and y. To complete the design of the encrypted lookup table, a document D against input bits x and y is itself an encryption of `XOR`(x, y), such that upon decryption, the content of the decrypted document is either $0/1$ which is exactly the same as the output bit obtained upon application of `XOR` to x and y.

Concretely, we *design* the encrypted lookup table in a way such that for arbitrary one-bit inputs x and y (as well as their corresponding keyword mappings), *exactly* one encrypted document is returned, which upon decryption reveals a single bit $b =$ `XOR`(x, y), thereby allowing SEC to correctly compute the function `XOR`. We elaborate on the same here. There are two aspects to the design: (i) the encrypted *content* of the documents in the encrypted lookup table, and (ii) the inverted index (i.e. defining the mapping between the keywords and the encrypted documents). Table 1 elucidates the contents of documents in the encrypted lookup table. There is a *single* and *uniquely identifiable* document corresponding to every possible input combination to the function `XOR` (similar design strategy applies to {`AND,OR`}). Moreover, for any possible input combination, that document is an encryption of a single bit corresponding to the actual output of `XOR`(x, y).

Table 1: Identifiers and contents of documents related to functional evaluation of 2-bit `XOR`, as well as mapping of document identifiers to their corresponding keywords (**KW 1** and **KW 2**). Here, Enc_k refers to any generic encryption scheme with private key k.

x	**KW 1**	**y**	**KW 2**	`XOR`(**x**, **y**)	doc_id	**Doc. content**
0	$I(\mathbf{w}_1)$	0	$I(\mathbf{w}_2)$	0	D_0	$\mathsf{Enc}_k(0)$
0	$I(\mathbf{w}_1)$	1	$\mathbf{w}_2$	1	D_1	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	0	$I(\mathbf{w}_2)$	1	D_2	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	1	$\mathbf{w}_2$	0	D_3	$\mathsf{Enc}_k(0)$

SEC is specifically designed to ensure that *exactly* one document (or record) matches with an unique combination of $\mathbf{x}$ and $\mathbf{y}$. The above discussion elaborates the relation between single-bit inputs $\mathbf{x}$ and $\mathbf{y}$, and the keyword set $\{\mathbf{w}_1, I(\mathbf{w}_1), \mathbf{w}_2, I(\mathbf{w}_2)\}$. An overview of this relation is illustrated in Table 1. We note that for every possible combination of x and y (and by extension, for possible combination of KW 1 and KW 2), there is *exactly* one document associated. This, along with the Doc. content expressed in Table 1, ensures that for any combination of one-bit inputs x and y, *exactly* one encrypted document is returned, which upon decryption provides the correct bit output of $\texttt{XOR}(\mathbf{x}, \mathbf{y})$.

Table 1 summarizes a *forward index*, a representation of keywords and doc_id mappings wherein every doc_id is enumerated against the keywords associated with it. For SEC's purpose, we can derive the *inverse index* too, or the mapping from a given keyword to the doc_ids wherein that keyword exists. An astute reader might note that *there is exactly one common document between any combination of KW 1 and KW2, which is returned as response to the* SEC *query.* This design, in conjunction with the Doc. content from Table 1, ensures that only the correct evaluation of XOR is obtained as a result of querying on SEC. The reader will also note that at no point in this entire process have we performed an explicit computation of XOR gate (unlike what FHE does). The entire computation has completed by *searching* over encrypted lookup tables.

2 Preliminaries and Background

In this section, we present the notations and the preliminary background material used in our construction. We begin by listing the notations used throughout the paper followed by a generic SSE syntax, a brief discussion on OXT scheme and the TSet data structure. Any other notation used is defined in-place within the context in the main text.

2.1 Notations

We write $x \xleftarrow{\$} \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x = \mathcal{A}$ and the output x' of a randomized algorithm $\mathcal{A}'$ is denoted by $x' \leftarrow \mathcal{A}'$. We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ any generic (unspecified) polynomial function and negligible function in λ, respectively[3]. We denote integers by $\mathbb{Z}$ and multiplicative group modulo some prime (q) over integers as $\mathbb{Z}_q$. F and F_q are pseudorandom functions with output range in $\{0,1\}^\lambda$ and $\mathbb{Z}_q$ respectively.

Databases. Let $\mathcal{W} = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$ be a dictionary of keywords where N is the total number of keywords in the database. The total number of documents in the database is denoted by d, each document is associated with a unique identifier denoted as id (or doc-id) and contains keywords from $\mathcal{W}$. We denote by $\mathbf{DB}$ a database of identifier-keyword pairs, such that $(\mathsf{id}, \mathsf{w}) \in \mathbf{DB}$ if and only if the document with identifier id contains the keyword w. We denote by $\mathbf{DB}(\mathsf{w})$ the set of all identifiers corresponding to documents containing w. We denote by $|\mathcal{W}|$ the number of distinct keywords in $\mathbf{DB}$, by $|\mathbf{DB}|$ the number of distinct id-w pairs in $\mathbf{DB}$, and by $|\mathbf{DB}(\mathsf{w})|$ the number of documents containing w. Maximum number of

[3]Note that a function $f : \mathbb{N} \to \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p, $f(\lambda) \leq 1/p(\lambda)$ when λ is sufficiently large

keywords in a query in denoted by n and the result set returned by the server to the client is denoted by R_q.

2.2 SSE: Syntax and Security Model

In this section, we formally define searchable symmetric encryption (SSE).

Formal Definition of SSE. A *searchable symmetric encryption (SSE) scheme* Π consists of an algorithm Setup and a protocol Search between the client and server, mentioned as follows:

- Setup takes as input a database **DB**, and outputs a secret key K along with an encrypted database **EDB**.
- The Search protocol is between a *client* and *server*, where the client takes as input the secret key K and a query q and the server takes as input **EDB**. At the end, the client outputs a set of identifiers, and the server has no output.

Correctness. An SSE scheme is said to be correct if for all inputs **DB** and queries q, if $(K, \mathbf{EDB}) \xleftarrow{\$} \textsc{Setup}(\mathbf{DB})$, after running Search with client input (K, q) and server input **EDB**, the client outputs the set of indices $\mathbf{DB}(q)$.

Adaptive Security of SSE. We recall the semantic security definitions of SSE from [41, 49]. The definition is parameterized by a *leakage function* $\mathcal{L}$, which describes what an adversary (the server) is allowed to learn about the database and queries. Formally, security says that the server's view during an adaptive attack (where the server selects the database and queries) can be simulated given only the output of $\mathcal{L}$.

Let $\Pi = (\textsc{Setup}, \textsc{Search})$ be an SSE scheme and let $\mathcal{L}$ be a stateful algorithm. For algorithms $\mathcal{A}$ (denoting the adversary) and SIM (denoting a simulator), we define the experiments (algorithms) $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda)$ as in Algorithm 1 and Algorithm 2, respectively. We say that Π is $\mathcal{L}$-semantically-secure against adaptive attacks if for all adversaries $\mathcal{A}$ there exists an algorithm SIM such that

$$| \Pr[\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda) = 1] | \leq \mathsf{negl}(\lambda).$$

The leakage function for SSE is expressed as

$$\mathcal{L}_{\mathrm{SSE}} = (\mathcal{L}^{\textsc{Setup}}_{\mathrm{SSE}}, \mathcal{L}^{\textsc{Search}}_{\mathrm{SSE}}),$$

where $\mathcal{L}^{\textsc{Setup}}_{\mathrm{SSE}}$ encapsulates the leakage to an adversarial server during the Setup phase, and $\mathcal{L}^{\textsc{Search}}_{\mathrm{SSE}}$ encapsulates the leakage to an adversarial server during each execution of the Search protocol.

Algorithm 1 Experiment $\mathbf{Real}^{\mathrm{SSE}}_{\mathcal{A}}(\lambda)$

function $\mathbf{Real}^{\mathrm{SSE}}_{\mathcal{A}}(\lambda)$
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{sk}, \mathsf{st}_0, \mathbf{EDB}_0) \leftarrow \mathrm{SSE.}\textsc{Setup}(\lambda, N)$
  for $k \leftarrow 1$ to Q **do**
    Let $q_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_k, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow$
      $\mathrm{SSE.}\textsc{Search}(\mathsf{sk}, \mathsf{st}_{k-1}, q_k; \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after
      the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Algorithm 2 Experiment $\mathbf{Ideal}^{\mathrm{SSE}}_{\mathcal{A},\mathrm{SIM}}(\lambda, Q, \mathcal{L})$

function $\mathbf{Ideal}^{\mathrm{SSE}}_{\mathcal{A},\mathrm{SIM}}(\lambda, Q, \mathcal{L})$
  Parse the leakage function $\mathcal{L}$ as:
  $\mathcal{L} = \left(\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Search}}_{\mathrm{SSE}}\right)$.
  $(\mathsf{st}_{\mathrm{SIM}}, \mathbf{EDB}_0) \leftarrow \mathrm{SIM}_{\textsc{Setup}}(\mathcal{L}^{\textsc{Setup}}(\lambda, N))$
  for $k \leftarrow 1$ to Q **do**
    Let $q_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_{\mathrm{SIM}}, \mathbf{EDB}_k, \tau_k) \leftarrow \mathrm{SIM}_{\textsc{Search}}$
      $(\mathsf{st}_{\mathrm{SIM}}, \mathcal{L}^{\textsc{Search}}_{\mathrm{SSE}}(q_k); \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after
      the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Conjunctive SSE Scheme . A conjunctive SSE scheme CSSE provisions the client with *conjunctive* search capabilities over the encrypted database. The Setup phase is similar to a generic SSE scheme where the encrypted database is generated and offloaded to the server. During the Search phase the client issues a conjunctive query of the form $(\mathbf{w}_1 \wedge \ldots \wedge \mathbf{w}_n)$. It sends search tokens to the server corresponding to each keyword in the query like - $(\mathsf{st}_1 \wedge \ldots \wedge \mathsf{st}_n)$. The operation $\wedge$ represents a logical `AND`, implying the final document identifiers (i.e. $\mathbf{DB}(\mathsf{st}_1 \wedge \ldots \wedge \mathsf{st}_n)$) returned to the client refer to the encrypted documents that contain *all* $\mathbf{w}_1$ to $\mathbf{w}_n$. Correctness and security guarantee of a CSSE scheme follows from a generic SSE scheme. A CSSE scheme follows the adaptive security definition of SSE discussed earlier in this section. We consider $\mathbf{w}_1$ as the least frequent keyword in the query without loss of generality. Throughout the paper we denote $\mathbf{w}_1$ as sterm and rest of the keywords in the query $\mathbf{w}_2, \ldots, \mathbf{w}_n$ as the xterm.

2.3 Overview of OXT

We adopted OXT by Cash et al. [45] as our underlying CSSE scheme for SEC. OXT is a state-of-the-art sublinear SSE scheme supporting conjunctive Boolean queries efficiently while incurring optimal linear storage overhead. OXT has the routines Setup and Search in its construction, defined in Algorithm 3 and 4.

Concretely, OXT uses *oblivious cross-tags* which are elements of a prime order subgroup

of the discrete-log hard group $\mathbb{Z}_p^*$. The cross-tags are elements in $\mathbb{Z}_p^*$ computed as a composition of keywords and the corresponding document identifiers, which are stored in a specialised data structure called cross-set or XSet. XSet facilitates conjunctive search as it allows to check the validity of a queried (w,id) pair using XSet lookup. At the end of SETUP, XSet is offloaded to the untrusted server as a part of the encrypted database **EDB**. The encrypted document identifiers and associated pre-computed blinding factors are stored in a specialised data-structure called TSet (see Section 2.4 for detail). TSet is offloaded to the untrusted server as a part of **EDB**, and the server looks up **EDB** during search using encrypted search tokens generated from client's query to retrieve matched encrypted document identifiers OXT is proven simulation-secure against semi-honest adversarial server. Hence, the server learns no information about the encrypted query keywords or the data stored in **EDB** based on the information received during search. Since SEC uses OXT for encrypted look up on the server, it is guaranteed by the simulation security guarantee of OXT that the server does not learn anything, such as the function itself or the encrypted inputs, beyond precisely defined benign information. Algorithm 3 and 4 explains formally the SETUP and SEARCH routine of OXT.

2.4 Overview of TSet

We briefly explain the syntax of the special primitive introduced in OXT, the tuple set or TSet. Intuitively, a TSet associates a list of fixed-sized data tuples (list for each keyword is made of every document identifier that contains the particular keyword) with each keyword in the database. The original OXT scheme uses it as an "expanded inverted index". For

Algorithm 3 Setup algorithm of OXT

Input: $1^\lambda, \mathbf{DB}$
Output: $\mathsf{mk}, param, \mathbf{EDB}$

function EDB.SETUP($1^\lambda, \mathbf{DB}$)
  Initialise $T \leftarrow \phi$ indexed by keywords $\mathcal{W}$
  Select key K_S for PRF F
  Select keys K_I, K_Z, K_Z for PRF F_p
  Initialise $\mathbf{EDB} \leftarrow \{\}$
  Initialise $\mathsf{XSet} \leftarrow \{\}$
  for $w \in \mathcal{W}$ do **do**
    Initialise $t \leftarrow \{\}$
    Compute $k_e \leftarrow F(K_S, \mathsf{w})$
    Set counter $c \leftarrow 1$
    for $\mathsf{id} \in \mathbf{DB}(\mathsf{w})$ do **do**
      Compute $xid \leftarrow F_p(K_I, \mathsf{id})$
      Compute $z_w \leftarrow F_p(K_Z, \mathsf{w}||c)$
      Compute $y_{\mathsf{id}} \leftarrow xid \cdot z_w^{-1}$
      Compute $e_c \leftarrow Sym.\mathsf{Enc}(k_e, \mathsf{id})$
      Set $xtag \leftarrow g^{F_p(K_X, \mathsf{w}) \cdot xid}$
      Add $xtag$ to XSet
      Append (y_{id}, e_c) to t and set $c \leftarrow c + 1$
    Set $T[\mathsf{w}] \leftarrow t$
  Compute $\{\mathsf{TSet}, K_T\} \leftarrow \mathsf{TSet.SetUp}(T)$ ▷ *(See [45] for TSet routines)*
  return $\mathsf{mk} = \{msk, K_S, K_I, K_Z, K_X, K_T\}$, $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$

Algorithm 4 Search algorithm of OXT

Input: $\mathsf{mk}, param, q = (\mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n), \mathbf{EDB}$
Output: Result R_q

function EDB.SEARCH($\mathsf{mk}, param, q, \mathbf{EDB}$)
  Client's inputs are $(\mathsf{mk}, param, q)$ and server's inputs are $(param, \mathbf{EDB})$
  Client initialises $R_q \leftarrow \{\}$ and computes $stag \leftarrow \mathsf{TSet.GetTag}(K_T, \mathsf{w}_1)$
  Client sends $stag$ to the server
  Server recovers $\mathbf{EDB}(1) = \mathsf{TSet}$. starts accepting $xtokens$ computed by client as follows:
  for $c = 1$: until server sends $stop$ **do**
    Client computes $f_{\mathsf{w}_1} \leftarrow F_p(k_Z, \mathsf{w}_1 || c)$
    for $l = 2 : n$ do **do**
      Client computes $xtoken[c, l] \leftarrow g^{f_{\mathsf{w}_1} \cdot F_p(k_X, \mathsf{w}_l)}$
    Client sets $xtoken[c] \leftarrow (xtoken[c, 2], \ldots, xtoken[c, n])$.
    Client send $xtoken[c]$ to server.
  Server initialises $\mathcal{E} \leftarrow \{\}$.
  Server computes $t \leftarrow \mathsf{TSet.Retrieve}(\mathsf{TSet}, stag)$
  for $c = 1 : |t|$ do **do**
    Server recovers (y_{id}, e_c) from c-th component of t.
    for $l = 2 : n$ do **do**
      Server computes $xtag = xtoken[c, l]^{y_{\mathsf{id}}}$
    If $\forall l \in [2, n]$, $xtag \in \mathsf{XSet}$, then send e_c to the client.
    When last tuple in t is reached, send $stop$ to client and halt.
  Client computes $K_e \leftarrow F(K_S, \mathsf{w}_1)$.
  Client computes $\mathsf{id}_c \leftarrow Sym.Dec(K_e, e_c)$, and adds id_c to R_q for all e_c received.
  return R_q

conjunctive keyword search, the TSet is used to store the encrypted indices of document along with some additional information, that is later used by the server to obliviously compute tokens for xtag generation. Formally TSet instantiation consists of three algorithms $\Sigma = (\mathsf{TSet.SetUp}, \mathsf{TSet.GetTag}, \mathsf{TSet.Retrieve})$, each of these algorithm is briefly explained below.

TSet Syntax. Formally, a TSet implementation $\Sigma = (\mathsf{TSet.SetUp}, \mathsf{TSet.GetTag}, \mathsf{TSet.Retrieve})$ will consist of three algorithms with the following syntax

- TSet.SetUp takes as input $T = (T_1, \ldots, T_N)$, where each T_i for $i \in [N]$ is an array of lists of equal-length bit strings indexed by the elements of $\mathcal{W}_i$, and outputs (TSet, K_T).
- TSet.GetTag takes as input the key K_T and a tuple $(i, \mathbf{w})$ and outputs stag_i.
- TSet.Retrieve takes as input TSet and stag_i, and returns a list of strings.

TSet Correctness. We say that Σ is *correct* if for all $\{\mathcal{W}_i\}_{i \in [N]}$, all $T = (T_1, \ldots, T_N)$, and any $\mathbf{w} \in \mathcal{W}_i$, we have

$$\mathsf{TSet.Retrieve}(\mathsf{TSet}, \mathsf{stag}) = T_i[\mathbf{w}],$$

when we have $(\mathsf{TSet}, K_T) \leftarrow \mathsf{TSet.SetUp}(T)$ and $\mathsf{stag} \leftarrow \mathsf{TSet.GetTag}(K_T, (i, \mathbf{w}))$. Intuitively, T holds lists of tuples associated with keywords and correctness guarantees that the TSet.Retrieve algorithm returns the data associated with the given keyword.

TSet Security. The security goal of a TSet implementation is to hide as much as possible about the tuples in $T = (T_1, \ldots, T_N)$ and the attribute-value pairs these tuples are associated to, except for the vectors $T_i[\mathbf{w}_1], T_i[\mathbf{w}_2], \ldots$ of tuples revealed by the client's queried attribute-value pairs $\mathbf{w}_1, \mathbf{w}_2, \ldots$. (For the purpose of TSet implementation we equate client's query with a single attribute-value pair.)

The formal definition of security for TSet is similar to that of keyword-search based SSE for single-keyword queries. We refer the reader to prior works [45, 43, 50] for the formal definition and concrete instantiations of TSet; in our paper, we adopt the same definition of security and the same concrete instantiation as used in these works.

3 $\mathsf{SEC}_{\mathsf{Basic}}$: Basic Construction

In this section we present our first encrypted compute-framework construction $\mathrm{SEC}_{\mathsf{Basic}}$ - an end-to-end privacy-preserving framework for evaluating arbitrary depth Boolean circuits using encrypted lookup tables. Before delving into the technical details of our construction we outline the formal syntax of SEC below.

3.1 Syntax

We briefly explain a general syntax of our construction SEC. The basic construction $\mathrm{SEC}_{\mathsf{Basic}}$ follow similar syntax. SEC uses static conjunctive SSE construction CSSE as a black-box following the syntax above. We assume a single benign client and a semi honest server[4] in SEC system model. A SEC scheme comprises of the following algorithms, as defined below.

• $\mathrm{SEC.Setup}(1^\lambda, \mathsf{f}, \{\mathbf{x}_i\}_{i\in[n]})$: SEC Setup is a PPT algorithm takes the security parameter λ and a collection of lookup tables f for basic gates/functions of n variables $\{\mathbf{x}_i\}_{i\in[n]}$, and outputs the master secret key mk and the encrypted database $\mathbf{EDB}$.

• $\mathrm{SEC.Evaluate}(\mathsf{f}, \{\mathbf{x}_i\}_{i\in[n]}, \mathsf{mk}, \mathbf{EDB})$: SEC Evaluate is a deterministic polynomial time protocol jointly executed by the client and the server, where the client's inputs are the input values to the circuit to be evaluated and the server's input is $\mathbf{EDB}$. At the end of protocol execution, the client receives the encrypted output of the circuit evaluation over the input values provided by the client and the server receives nothing.

Note that, although the Evaluate routine takes client input values in plain-text in this syntax, these values are encrypted on the client-side prior to sending to server and circuit evaluation. Thus, SEC operates on encrypted inputs and produces encrypted output.

Correctness. SEC is said to be correct if SEC.Evaluate returns a correct output for a circuit and inputs provided by the client.

Security of SEC. Formally, SEC is said to be adaptively secure with respect to a leakage function $\mathcal{L}$ if for any stateful PPT adversary $\mathcal{A}$ that evaluates a maximum of $F = \mathsf{poly}(\lambda)$ circuits, there exists a stateful PPT simulator $\mathrm{SIM} = (\mathrm{SIM}_{\mathrm{Setup}}, \mathrm{SIM}_{\mathrm{Evaluate}})$ such that the following holds:

$$\left|\Pr\left[\mathbf{Real}^{\mathrm{SEC}}_{\mathcal{A}}(\lambda, F) = 1\right] - \Pr\left[\mathbf{Ideal}^{\mathrm{SEC}}_{\mathcal{A},\mathrm{SIM}}(\lambda, F) = 1\right]\right| \leq \mathsf{negl}(\lambda),$$

[4]We consider a semi-honest server because traditional FHE schemes [16] consider a semi-honest server model. Verifiable FHE [51] which considers malicious-server model is incredibly inefficient.

where the "real" experiment $\mathbf{Real}^{\mathrm{SEC}}$ and the "ideal" experiment $\mathbf{Ideal}^{\mathrm{SEC}}$ are as described in Algorithm 5 and Algorithm 6. The leakage function for SEC is expressed as

$$\mathcal{L}_{\mathrm{SEC}} = (\mathcal{L}_{\mathrm{SEC}}^{\textsc{Setup}}, \mathcal{L}_{\mathrm{SEC}}^{\textsc{Evaluate}}),$$

where $\mathcal{L}_{\mathrm{SEC}}^{\textsc{Setup}}$ captures the leaked information during Setup, and $\mathcal{L}_{\mathrm{SEC}}^{\textsc{Evaluate}}$ captures the leaked information during Evaluate execution.

Algorithm 5 Experiment $\mathbf{Real}_{\mathcal{A}}^{\mathrm{SEC}}(\lambda, F)$

function $\mathbf{Real}_{\mathcal{A}}^{\mathrm{SEC}}(\lambda, F)$
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{sk}, \mathsf{st}_0, \mathbf{EDB}_0) \leftarrow \mathrm{SEC.}\textsc{Setup}(\lambda, N)$
  for $k \leftarrow 1$ to F **do**
    Let $f_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_k, \mathbf{EDB}_k, \mathbf{DB}(f_k)) \leftarrow$ $\mathrm{SEC.}\textsc{Evaluate}(\mathsf{sk}, \mathsf{st}_{k-1}, f_k; \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_F, \tau_1, \ldots, \tau_F)$
  return b

Algorithm 6 Experiment $\mathbf{Ideal}_{\mathcal{A},\mathrm{SIM}}^{\mathrm{SEC}}(\lambda, F, \mathcal{L})$

function $\mathbf{Ideal}_{\mathcal{A},\mathrm{SIM}}^{\mathrm{CSSE}}(\lambda, F, \mathcal{L})$
  Parse the leakage function $\mathcal{L}$ as: $\mathcal{L} = \left(\mathcal{L}^{\textsc{Setup}}, \mathcal{L}_{\mathrm{SEC}}^{\textsc{Search}}\right)$.
  $(\mathsf{st}_{\mathrm{SIM}}, \mathbf{EDB}_0) \leftarrow \mathrm{SIM}_{\textsc{Setup}}(\mathcal{L}^{\textsc{Setup}}(\lambda, N))$
  for $k \leftarrow 1$ to F **do**
    Let $f_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{st}_{\mathrm{SIM}}, \mathbf{EDB}_k, \tau_k) \leftarrow \mathrm{SIM}_{\textsc{Evaluate}}$ $(\mathsf{st}_{\mathrm{SIM}}, \mathcal{L}_{\mathrm{SEC}}^{\textsc{Evaluate}}(f_k); \mathbf{EDB}_{k-1})$
    Let τ_k denote the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_F, \tau_1, \ldots, \tau_F)$
  return b

3.2 Technical Details

For $\mathrm{SEC}_{\mathsf{Basic}}$, the objective is to implement the universal Boolean function set {`XOR`,`AND`,`OR`}, which in turn can be used for evaluation of *arbitrary* Boolean function. We define $\mathrm{SEC}_{\mathsf{Basic}}$ as a triple of algorithms {GenDB, Setup, Evaluate}.

GenDB Workflow. Algorithm 7 formally describes the GenDB routine for $\mathrm{SEC}_{\mathsf{Basic}}$ construction. It takes as input the set of functions f to be modelled (for example, the universal Boolean function set {`XOR`,`AND`,`OR`}), as well as the bits $\mathbf{x}_1$ and $\mathbf{x}_2$ to the Boolean functions in f. GenDB generates a *sub-database* for each function f_i in f as follows.

Algorithm 7 $\text{SEC}_{\mathsf{Basic}}$.GenDB

Input: $f = (f_1, \ldots, f_n), \mathbf{x}_1, \mathbf{x}_2$
Output: **DB**

function $\text{SEC}_{\mathsf{Basic}}$.GenDB$(f, \mathbf{x}_1, \mathbf{x}_2.)$
  for $i \in 1$ *to* $|f|$ **do**
    Assign unique keyword string to each bit value taken by $\mathbf{x}_1$ and $\mathbf{x}_2$
    Evaluate f_i on all possible combination of $\mathbf{x}_1$ and $\mathbf{x}_2$ and store the output bit after each evaluation in an encrypted document.
    Map the keywords to the documents such that the output bit stored in the document is obtained after evaluating f_i with the keyword under consideration as one of the inputs.
    Set $\mathbf{DB}_{f_i}$ as the keyword-document map thus generated.
    $\mathbf{DB} \leftarrow \mathbf{DB} \cup \mathbf{DB}_{f_i}$
  Return **DB**

- Following the novel keyword mapping introduced in Section 1.2, both bits $\mathbf{x}_1$ and $\mathbf{x}_2$ are mapped to suitable alphanumeric keywords.
- Against each possible value of bits $\mathbf{x}_1$ and $\mathbf{x}_2$, the functional evaluation $f_i(\mathbf{x}_1, \mathbf{x}_2)$ is stored in a document encrypted with a generic IND-CPA secure symmetric-key encryption scheme.
- Following the encrypted lookup table design detailed in Section 1.2, construct the *inverted index* for $\text{SEC}_{\mathsf{Basic}}$.

The final *unencrypted* database **DB** is a collection of disjoint databases. For instance, considering the set $\{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$, the final *unencrypted* database returned by GenDB function is $\mathbf{DB} \leftarrow \{\mathbf{DB}_{\texttt{XOR}} \cup \mathbf{DB}_{\texttt{AND}} \cup \mathbf{DB}_{\texttt{OR}}\}$. The structure of **DB** as a collection of the sub-databases however leads to an ambiguity: the server cannot distinguish between the different sub-databases when they are all collectively combined into **DB**, thereby perform a correct lookup. To circumvent this ambiguity, we append the function f_i in f along with each input value in such a way that the pointer to a particular entry in a specific encrypted sub-database is always correctly computed thereby ensuring a correct lookup by the server. Table 2 represents a better understanding of how the keywords are mapped to the corresponding document-identifiers ($\mathsf{doc\text{-}id}$) along with the function (f) appended with each entry. The actual **DB** generated by GenDB is in the form of an *inverted index* where each keyword is appended with a specific function f_i and maps to exactly two documents.

Algorithm 8 $\text{SEC}_{\mathsf{Basic}}$.Setup

Input: $1^\lambda, f = (f_1, \ldots, f_n), \mathbf{x}_1, \mathbf{x}_2$
Output: $\mathsf{mk}, \mathbf{EDB}$

function $\text{SEC}_{\mathsf{Basic}}$.Setup$(1^\lambda, f, \mathbf{x}_1, \mathbf{x}_2.)$
  $\mathbf{DB} \leftarrow$ GenDB$(f, \mathbf{x}_1, \mathbf{x}_2)$
  $(\mathbf{EDB}, \mathsf{mk}) \leftarrow \mathsf{OXT}.\text{Setup}(1^\lambda, \mathbf{DB})$
  return $\mathsf{mk} = \{K_S, K_I, K_Z, K_X, K_T\}$, $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$

Setup Workflow . The Setup routine is explained in Algorithm 8. It generates the *unencrypted* lookup table (denotes by **DB**) for the function set $f = (f_1, f_2, \cdots, f_n)$ by

Table 2: Mapping of document identifiers to their corresponding keywords along with the specific function (f). **KW** symbolizes the *keyword* linked to a specific doc_id. $I(w_i)$ refers to the absence of keyword w_i corresponding to doc_id. Enc_k symbolizes the encryption oracle of a IND-CPA secure symmetric-key encryption scheme.

$\mathbf{x_1}$	**KW 1**	$\mathbf{x_2}$	**KW 2**	f	doc_id	doc_content
0	$I(w_1)$	0	$I(w_2)$	AND	D_0	$\mathsf{Enc}_k(0)$
0	$I(w_1)$	1	$\mathbf{w}_2$	AND	D_1	$\mathsf{Enc}_k(0)$
1	$\mathbf{w}_1$	0	$I(w_2)$	AND	D_2	$\mathsf{Enc}_k(0)$
1	$\mathbf{w}_1$	1	$\mathbf{w}_2$	AND	D_3	$\mathsf{Enc}_k(1)$
0	$I(w_1)$	0	$I(w_2)$	OR	D_4	$\mathsf{Enc}_k(0)$
0	$I(w_1)$	1	$\mathbf{w}_2$	OR	D_5	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	0	$I(w_2)$	OR	D_6	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	1	$\mathbf{w}_2$	OR	D_7	$\mathsf{Enc}_k(1)$
0	$I(w_1)$	0	$I(w_2)$	XOR	D_8	$\mathsf{Enc}_k(0)$
0	$I(w_1)$	1	$\mathbf{w}_2$	XOR	D_9	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	0	$I(w_2)$	XOR	D_{10}	$\mathsf{Enc}_k(1)$
1	$\mathbf{w}_1$	1	$\mathbf{w}_2$	XOR	D_{11}	$\mathsf{Enc}_k(0)$

invoking GenDB function. Thereafter, it invokes the Setup routine of OXT to generate the encrypted database or **EDB**, which is a collection of the TSet and XSet data structure (Section 2.3). Along with **EDB** it also returns a parameter mk which encapsulates the keys used for different PRFs and TSet operations specific to OXT. Concretely, **EDB** is the encrypted lookup table that is offloaded to the cloud server. Note that Setup is a *one-time* process that is executed only once at the beginning of the $\mathsf{SEC}_{\mathsf{Basic}}$ scheme. It may be emphasized that the Setup is like generating the basic libraries based upon which *any* function can be evaluated, and thus the Setup is a one-step process catering to all possible function evaluations.

Algorithm 9 $\mathsf{SEC}_{\mathsf{Basic}}$.Evaluate

Input: $f_i, \mathbf{x}_1, \mathbf{x}_2, \mathsf{mk}, \mathbf{EDB}$
Output: $\mathsf{y} = f_i(\mathbf{x}_1, \mathbf{x}_2)$

1: **function** $\mathsf{SEC}_{\mathsf{Basic}}.\mathsf{Evaluate}(f_i, \mathbf{x}_1, \mathbf{x}_2, \mathsf{mk}, \mathbf{EDB}.)$
2: **Client**: Generate a query q_{f_i} using keyword mappings of $\mathbf{x}_1$, $\mathbf{x}_2$, appended with f_i
3: **Server**: Run $\mathsf{OXT}.\textsc{Search}(f_i, \mathbf{EDB}, q_{f_i}, \mathsf{mk})$ and retrieve $\mathbf{EDB}(\mathbf{x}_1 \wedge \mathbf{x}_2)$
4: **Server**: Return $\mathsf{doc\_id} \leftarrow \mathbf{EDB}(\mathbf{x}_1 \wedge \mathbf{x}_2)$ to the client at the end of the protocol.
5: **Client**: $\mathsf{y} \leftarrow \mathsf{Dec}_{\mathsf{k}}(D_{doc\_id})$
6: Client obtains y which is equal to the output of $f_i(\mathbf{x}_1, \mathbf{x}_2)$.

Evaluate Specifications . The corresponding algorithm for the Evaluate phase of $\mathsf{SEC}_{\mathsf{Basic}}$ is summarized in Algorithm 9. To evaluate an arbitrary Boolean function f_i in f on the input bits $\mathbf{x}_1$ and $\mathbf{x}_2$, Evaluate first generates a query $q_{f_i} = \mathsf{st}_{\mathbf{x}_1} \wedge \mathsf{st}_{\mathbf{x}_2}$. Herein, $\mathsf{st}_{\mathbf{x}_1}$ represents the search token corresponding to the keyword matched with the specific bit value of $\mathbf{x}_1$ appended with the specific function that is to be evaluated. Likewise, $\mathsf{st}_{\mathbf{x}_2}$ represents the search token corresponding to the keyword matched with $\mathbf{x}_2$ appended with

the specific function f_i. The query q is sent to the server, which runs the OXT.Search algorithm to retrieve a *single* document identifier $\mathsf{doc\_id}$ against the specific values of $\mathbf{x}_1$, $\mathbf{x}_2$, and the function (f_i. From Table 2 it can be guaranteed that only one document exists per each possible value of the entry $(\mathbf{x}_1, \mathbf{x}_2, f_i)$. This *encrypted* $\mathsf{doc\_id}$ is returned back to the client, which decrypts it to get the document corresponding to the evaluated result of $f_i(\mathbf{x}_1, \mathbf{x}_2)$. In one extra round of communication between the server and the client, the actual encrypted document corresponding to $\mathsf{doc\_id}$ is retrieved by the client from the server, which upon decryption gives the unencrypted bit $\mathsf{y} = f_i(\mathbf{x}_1, \mathbf{x}_2)$. The design of **EDB**, as is evident from Table 2 ensures only a single $\mathsf{doc\_id}$ is returned for every possible variation of the search query $q_{f_i} = \mathsf{st}_{\mathbf{x}_1} \wedge \mathsf{st}_{\mathbf{x}_2}$.

3.3 Proof of Correctness of $\mathrm{SEC}_{\mathsf{Basic}}$

The proof of correctness for $\mathrm{SEC}_{\mathsf{Basic}}$ follows from the correctness of CSSE. The correctness of CSSE ensures that a conjunctive query $q = \mathsf{w}_1 \wedge \cdots \wedge \mathsf{w}_n$ over an encrypted database satisfies the following relations.

$$\begin{aligned} \mathbf{EDB} &\leftarrow \text{CSSE.Setup}(\mathbf{DB}), \\ \mathbf{DB}(\mathsf{w}_1) \cap \cdots \cap \mathbf{DB}(\mathsf{w}_n) &= \text{CSSE.Search}(q, \mathbf{EDB}). \end{aligned}$$

Proof. Consider a binary function f which takes as input two bits say $\mathbf{x}_1$ and $\mathbf{x}_2$. The evaluation of $f(\mathbf{x}_1, \mathbf{x}_2)$ should return and output bit y. The goal of $\mathrm{SEC}_{\mathsf{Basic}}$ is to compute $\mathsf{y} = f(\mathbf{x}_1, \mathbf{x}_2)$ when both $\mathbf{x}_1$ and $\mathbf{x}_2$ are encrypted. By deploying CSSE as a black-box, $\mathrm{SEC}_{\mathsf{Basic}}$ generates the encrypted database specific to the functions supported by the scheme. The database consists of keywords corresponding to input bits and documents corresponding to the output of the function evaluation. During a conjunctive search the CSSE.Search protocol generates search tokens corresponding to the input bits to the function. The search token retrieves a document which consists of the encrypted output of the computation. The client decrypts it locally and obtains $\mathsf{y} = f(\mathbf{x}_1, \mathbf{x}_2)$.

The result returned by the CSSE.Search protocol is equivalent to the output y evaluated by $f(\mathbf{x}_1, \mathbf{x}_2)$. *For a functionally correct and exact conjunctive SSE scheme* CSSE*, a set of functions $f = (f_1, \ldots, f_n)$ supported by the scheme, and input literals $\mathbf{x}_1, \mathbf{x}_2$,* $\mathrm{SEC}_{\mathsf{Basic}}$ *is functionally correct if the following expressions hold.*

$$\begin{aligned} \mathsf{mk}, \mathbf{EDB} &\leftarrow \mathrm{SEC}_{\mathsf{Basic}}.\text{Setup}(1^\lambda, f, \mathbf{x}_1, \mathbf{x}_2), \\ \mathsf{y} &= \mathrm{SEC}_{\mathsf{Basic}}.\text{Evaluate}(f_i, \mathbf{x}_1, \mathbf{x}_2, \mathsf{mk}, \mathbf{EDB}). \end{aligned}$$

where, y is the output of the evaluation of some i-th function from the function set f, expressed as $\mathsf{y} = f_i(\mathbf{x}_1, \mathbf{x}_2)$.

3.4 $\mathrm{SEC}_{\mathsf{Basic}}$ Complexity Analysis

Computation Overhead. The evaluation time of $\mathrm{SEC}_{\mathsf{Basic}}$ for computing a binary function $f_i(\mathbf{x}_1, \mathbf{x}_2)$ over encrypted inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ scales linearly with the search time complexity of the underlying conjunctive SSE scheme. According to OXT the search tokens generated are searched for in the encrypted database and the time required to retrieve the documents scale with the frequency of the least frequent keyword in the conjunctive query. Since the database under consideration is very small the search time scales to few milliseconds. The document returned by the $\mathsf{OXT\_Search}$ is returned to the client, who decrypts it

locally and obtains the output bit. Hence the overall time required to compute a function f_i is essentially equal to the search time complexity of the underlying CSSE scheme, which in our case is OXT. $\text{SEC}_{\mathsf{Basic}}$ *is hence highly efficient and fast with sub-linear time complexity for evaluating a binary function of the form* $f(\mathbf{x}_1, \mathbf{x}_2)$ *over encrypted inputs* $\mathbf{x}_1$ *and* $\mathbf{x}_2$.

Storage Overhead. The server-side storage required for $\text{SEC}_{\mathsf{Basic}}$ depends upon the number of functions the scheme claims to support. Each function specific sub-database consists of exactly eight keyword-document pairs. The storage for each sub-database will scale with the total number of keyword-document pairs (as shown in Section 1.2). $\text{SEC}_{\mathsf{Basic}}$ is functionally complete as it supports three basic universal logic gates $\{\texttt{XOR},\texttt{AND},\texttt{OR}\}$. Therefore, the final $\mathbf{DB}$ is collectively composed of three sub-databases $\mathbf{DB} = \{\mathbf{DB}_{\texttt{AND}}, \mathbf{DB}_{\texttt{OR}}, \mathbf{DB}_{\texttt{XOR}}\}$, using which we can evaluate any arbitrary Boolean function over encrypted data. Since, we consider 2-bit input functions the space complexity scales with a constant number of keyword-document pair (the constant here is equal to 8). It is evident from the above discussion that the server-side storage required by $\text{SEC}_{\mathsf{Basic}}$ is extremely small and highly optimized.

3.5 Limitations of $\text{SEC}_{\mathsf{Basic}}$

Any *arbitrary* privacy-preserving computation framework should be capable of evaluating arbitrary Boolean circuits of unbounded depth. While $\text{SEC}_{\mathsf{Basic}}$ harbors this capability, there is a huge bottleneck. The specific design of the encrypted lookup table (Section 1.2) allows the server to return a *single* document id $\mathsf{doc\_id}$ as illustrated in $\text{SEC}_{\mathsf{Basic}}$.EVALUATE (Algorithm 9).

However, to evaluate function compositions of the form $f_k(f_i(.), f_j(.))$ (where f_i, f_j, $f_k \in f$), the $\mathsf{doc\_id}$ retrieved by the innermost functional evaluation (q_{f_k}) needs to be returned to the client (Step 5; Algorithm 9). The client retrieves the encrypted document, decrypts it to retrive y (Step 6; Algorithm 9), creates a new query q_{f_j}, and resends it to the server for next encrypted table lookup. Concretely, the resultant encrypted document D by running $\text{SEC}_{\mathsf{Basic}}$.EVALUATE on query $q_{f_j} = \mathsf{st}_{\mathbf{x}_1} \wedge \mathsf{st}_{\mathbf{x}_2}$ *cannot* be directly used as a search token for *future* queries. This requires the intervention of the client, which creates search tokens for *future* queries after decrypting the document returned by query q_{f_j}.

However, this drawback negates any advantages offered by the usage of encrypted lookup tables. Not only did it introduce (i) multiple rounds of communication between the server and the client for securely computing Boolean circuits of arbitrary depth, but also (ii) increased the computation load of the (already) resource-constrained client by requiring decryptions for every functional evaluation. To circumvent these issues, we introduce the novel idea of replacing the elements in the encrypted lookup table with *search tokens* specific to input bits and functions in a way such that, *it allows the results of* SEC *search queries to act as search queries themselves.* This then aids in arbitrary depth Boolean function evaluation on the server side without requiring any intermediate round of communication with the client. We elaborate our final construction SEC below.

4 SEC: Final Construction

Herein, we propose our final construction SEC that supports extremely fast and efficient encrypted function evaluation of arbitrary depth with a *single* round of communication

between the client and server. We introduce a novel encrypted lookup table design that stores search tokens such that the entries retrieved by an inner query serves as the search token for the outer query. We elucidate the technical details of the novel lookup table design subsequently.

Composition via Pseudo-documents. For computing an n-depth Boolean circuit, we store an additional `pseudo-doc` along with the documents corresponding to every keyword for every n-1 levels of functions. The `pseudo-doc` consists of search tokens required for evaluation of the function at the immediate higher level (immediate higher level for $f_{n-2}(\cdot)$ corresponds to $f_{n-1}(\cdot)$). Only the last or n-th level function f_n will retrieve encrypted values that maps the corresponding input keywords (bits) to the documents (output bits). For all other functions at (n-1) levels, search tokens will retrieve a `pseudo-doc` which returns another search token required to evaluate function at the immediate higher level. Our design and implementation of SEC is specific to OXT , but we note that it can be incorporated using any conjunctive SSE scheme. To the best of our knowledge, we are the first to show that arbitrary depth Boolean functions can be computed over encrypted data with a *single* round of communication between client and server by leveraging the extremely efficient sublinear search mechanism of a conjunctive SSE scheme (OXT in our construction).

Consider the following example setting for a better understanding. We would expect SEC to evaluate functional compositions of the form $f_k(f_i(\mathbf{x}_1, \mathbf{x}_2), f_j(\mathbf{x}_3, \mathbf{x}_4))$ (where f_i, f_j, $f_k \in f$ for any universal function set f. Example- {`AND`, `OR`, `XOR`}). Likewise, $\mathbf{x}_i : \forall i \in \{1, 2, 3, 4\}$ are input bits. For ease of exposition we consider an example of the form $f_k(\mathbf{w}, f_i(\mathbf{x}_1, \mathbf{x}_2))$ (where f_i, $f_k \in f$). Without loss of generality, we assume the inner function f_i is `AND` and the outer function f_k is `XOR`. Under this context, the precise objective of SEC is to support *conjunctive* search query (say $q_{\texttt{AND}}$) over the sub-database of `AND` (Table 2) such that the result of $q_{\texttt{AND}}$ is a valid input to the *conjunctive* search query to $q_{\texttt{XOR}}$.

4.1 Revised Encrypted Look-up Table Design

The database generation process in SEC comprises of two phases as follows.

Phase-I. The first phase is similar to $\text{SEC}_{\mathsf{Basic}}$ where the function specific sub-databases are generated. The sub-databases comprises of keywords that correspond to the input bits. Each keyword map to a pair of documents. Documents store the encrypted output bit value of the function with the particular keyword as one of the inputs. As mentioned earlier, CSSE scheme deployed in SEC is OXT and hence the CSSE.Setup and CSSE.Search protocol is directly extended from original OXT scheme in $\text{SEC}_{\mathsf{Basic}}$. In order to understand the technical novelty devised in Phase-II, we give a brief overview of the mathematical structure of each component used to build the encrypted database in Phase-I. The documents (id) are encrypted as $e \leftarrow \mathsf{Enc}(K_e, \mathsf{id})$. Specific to OXT a blinding factor z is calculated as $F(K_z, \mathbf{x}||f||c)$, where f is the specific function for which the sub-database is being generated, and c is a counter value that keeps a count of the frequency of the keyword $\mathbf{x}$. The blinding factor z is an element in $\mathbb{Z}_p^*$ which is inverted and multiplied with $\mathsf{xid} = F(K_I, \mathsf{id})$ (which is also a value in $\mathbb{Z}_p^*$) to pre-compute a blinded value y and stored in an encrypted data-structure. For each keyword-id pair an entry (y, e) is stored in the TSet by invoking the $\mathsf{TSet.SetUp}$ routine (Section 2.4). The cross-tags ($\mathsf{xtag} = g^{F(K_X, \mathbf{x}) \cdot F(K_I, \mathsf{id})}$) are generated as elements in a prime order subgroup of $\mathbb{Z}_p^*$ and stored in a special data-structure called

XSet. The TSet and XSet collectively comprise of the encrypted database **EDB** which is offloaded to the server.

Phase-II. In the second phase for every keyword in a sub-database an additional "pseudo-document" (`pseudo-doc`) is stored. The `pseudo-doc` actually stores a search token that involves a particular keyword and a function that could be possibly composed with the function of the sub-database under consideration. For evaluating a function composition over encrypted bits of the form - $f_1(\mathbf{w}, f_2(\mathbf{x}_1, \mathbf{x}_2))$ we devise some modifications to the structure of elements stored in **EDB**. The idea is to use *blinded exponentiation in a prime order sub-group $\mathbb{G}$ of $\mathbb{Z}_p^*$ (as done in DH-based oblivious PRF) to generate the search tokens for computing the inner level of function (f_2) in the composition.* For each input $\mathbf{x}$ of f_2, xid' is computed as $F_p(K_I, \mathbf{w}||\mathbf{x}||f_2)$ which is again an element in $\mathbb{Z}_p^*$. The blinding factor which is an element in $\mathbb{Z}_p^*$ is calculated as $z = F_p(K_Z, \mathbf{x}||f_2||c)$, it is calculated for each input to the inner function. The blinded value $y'_{\mathbf{x}_1} = \mathsf{xid}' \cdot z^{-1}$ is calculated similar to that in OXT. The encrypted value e' (`pseudo-doc`) is computed as $e' = g^{F_p(K_X, \mathbf{x})}$ and stored in **EDB** along with $y_{\mathbf{x}_1}$. It is to be noted here that e' does not hold the encrypted document identifier (output bit), but it stores a search token (an element in $\mathbb{G}$). The entire database generation process is formally explained in Algorithm 10. We explain the database generation process with the following simple example.

4.2 An Illustrative Example

Let us consider the setting where a client who wants to compute a function composition of the form $f_{\mathtt{OR}}(\mathbf{w}, f_{\mathtt{AND}}(\mathbf{x}_1, \mathbf{x}_2)) = \mathbf{w} + (\mathbf{x}_1 \cdot \mathbf{x}_2)$. Our construction SEC supports the set of three basic logic gates that are functionally complete. Therefore $\mathbf{DB} = \{\mathbf{DB}_{\mathtt{OR}}, \mathbf{DB}_{\mathtt{AND}}, \mathbf{DB}_{\mathtt{XOR}}\}$. The SEC.Setupexecution proceeds as follows.

Generating **EDB**

The routine SEC.Setupin Algorithm 10 generates $\mathbf{EDB} = \{\mathbf{EDB}_{\mathtt{OR}}, \mathbf{EDB}_{\mathtt{AND}}, \mathbf{EDB}_{\mathtt{XOR}}\}$. It comprises of two phases for every sub-database generation. We elaborate one such sub-database generation below -

$\mathbf{EDB}_{\mathtt{OR}}$:

Phase-I: Generates the database $\mathbf{EDB}_{\mathtt{OR}}$ mapping each keyword (input bit) to the corresponding documents (containing encrypted output bit) using $\mathrm{SEC}_{\mathsf{Basic}}$.Setup (Algorithm 8) as the black box.

Phase-II: Adds an additional pseudo-document in $\mathbf{DB}_{\mathtt{OR}}$ for every keyword. A pseudo-document consists of a (partial) search token e$'$. A blinded value $y'_{\mathbf{x}}$ and cross-tags are generated and stored in **EDB**.
At the end $\mathbf{EDB}_{\mathtt{OR}}$ is generated and added to the final $\mathbf{EDB} = \mathbf{EDB} \cup \{\mathbf{EDB}_{\mathtt{OR}}\}$. The entire **EDB** generation process is formally explained in Algorithm 10.

4.3 Evaluating Function Compositions

We briefly outline the workflow of OXT Search routine (provided in Section 2.3) that is necessary for the exposition of function composition technique of SEC. During the

SEARCH phase of OXT, a search token, denoted by xtoken), is generated by the client pertaining to each keyword in the conjunctive query. The search tokens are essentially elements in a discrete log hard group (prime order sub-group of $\mathbb{Z}_p^*$), hence by the structure of xtoken the underlying queried keywords are indistinguishable form random elements in $\mathbb{Z}_p^*$. The search tokens are generated on the fly as - $\mathsf{xtoken} = g^{F(K_x,\mathbf{x})\cdot F(K_z,\mathbf{w}||c)}$ according to the conjunctive query (say, $q = \mathbf{w} \wedge \mathbf{x}$) issued by the client. The xtoken is raised to some pre-computed value $y_{\mathbf{w}} = F(K_I, \mathsf{id}) \cdot F(K_z, \mathbf{w}||c)^{-1}$ by the server to obliviously compute the cross-tag. After the cross-tag is matched in XSet the documents stored in the encrypted database corresponding to the queried keywords are retrieved.

During the EVALUATE phase as presented in Algorithm 11, the client sends a value $r = F_p(K_Z, \mathbf{w}||c||f_1) \in \mathbb{Z}_p^*$ along with the search token (for f_2) which is used to retrieve the `pseudo-doc` e'. To obliviously calculate the search token (for f_1) the server raises $(e')^r$ which eventually gives $\mathsf{xtoken} = (e')^r = g^{[F_p(K_X,\mathbf{x}))\cdot(F_p(K_Z,\mathbf{w}||c||f_1)]}$. This oblivious computation of xtoken ensures that the server does not learn any information about the function being evaluated as well as the underlying inputs to the function. We meticulously design and bifurcate the search tokens into two parts - one stored in the `pseudo-doc` e' (consists of the part involving the queried keyword), the second part is sent by client as $r \in \mathbb{Z}_p^*$ (consists of the underlying function). This ensures two things - *(i) search tokens are generated on-the-fly depending upon the queried keywords and the functions to evaluate; (ii) The server can neither learn any information about the functions or the keywords (input bits) being evaluated nor can it infer any information from cross-query leakage.* We explain the EVALUATE phase with the following example.

Computing $f_{\mathtt{OR}}(\mathbf{w}, f_{\mathtt{AND}}(\mathbf{x}_1, \mathbf{x}_2))$

Let us assume, that $f_{\mathtt{OR}}(\mathbf{w}, f_{\mathtt{AND}}(\mathbf{x}_1, \mathbf{x}_2)) = \mathbf{w} + (\mathbf{x}_1 \cdot \mathbf{x}_2)$, [where $\mathbf{w} \to 1; \mathbf{x}_1 \to 0; \mathbf{x}_2 \to 1$]. The output of the above equation should therefore be equal to 1. SEC.EVALUATE protocol first evaluates `AND` operation ($f_{\mathtt{AND}}$) by searching for common pseudo-documents in $\mathbf{EDB}_{\mathtt{AND}}$. It then retrieves the search token for the outer function and computes $f_{\mathtt{OR}}$ by performing OXT.SEARCH on $\mathbf{EDB}_{\mathtt{OR}}$. The matched documents are checked and the result returned is a singleton set with *one* document which is equivalent to the encrypted output bit of the computation $f_{\mathtt{OR}}(\mathbf{w}, f_{\mathtt{AND}}(\mathbf{x}_1, \mathbf{x}_2))$. By correctness of our construction the the client receives a value equal to 1 after decrypting the document locally.

Client: It identifies the sterms (we consider the first input as the sterm since frequency of all keywords are same) in both functions and computes stag and stag' corresponding to $\mathbf{w}$, and $\mathbf{x}_1$, respectively (by invoking TSet.GETTAG, provided in Section 2.4). The search tokens are also generated as elements in $\mathbb{G}$ for computing $f_{\mathtt{AND}}$. Also it computes a value $r \in \mathbb{Z}_p^*$ which is used to generate the search token obliviously for $f_{\mathtt{OR}}$ at the server.

Server: The server receives xtoken' (for $f_{\mathtt{AND}}$) from the client and raises it to y' and obtains the cross-tag (xtag') obliviously. The cross-tags are matched in XSet and e' is recovered ((e', y') is recovered from $\mathsf{TSet.Retrieve}(\mathsf{TSet}, stag')$ (Section 2.4)). The server computes xtoken (for $f_{\mathtt{OR}}$) obliviously by raising e' to the power r. The xtoken computed thus, is raised to y ((e, y) is recovered from $\mathsf{TSet.Retrieve}(\mathsf{TSet}, stag)$) and xtag is obtained. Finally the XSet is checked for the presence of the corresponding cross-tag and e is returned to the client for all matched documents.

Algorithm 10 SEC.SETUP

Input: $1^\lambda, f = (f_1, f_2), \mathbf{x}_1, \mathbf{x}_2$
Output: $\mathsf{mk}, \mathbf{EDB}$

function SEC.SETUP$(1^\lambda, f, \mathbf{x}_1, \mathbf{x}_2.)$
  $\mathbf{DB} \leftarrow$ GENDB$(f, \mathbf{x}_1, \mathbf{x}_2)$
  PHASE - I
  $\mathbf{EDB}_\mathsf{I} \leftarrow \text{SEC}_{\mathsf{Basic}}.\text{SETUP}(1^\lambda, \mathbf{DB}, f)$
  $\mathbf{EDB} \leftarrow \mathbf{EDB} \cup \{\mathbf{EDB}_\mathsf{I}\}$
  PHASE - II
  for $\mathbf{w} \in \mathcal{W}$ **do**
    Set counter $c \leftarrow 1$
    for $\mathbf{x} \in \mathcal{W} \setminus \{\mathbf{w}\}$ **do**
      Compute $\mathsf{xid}' \leftarrow F_p(K_I, \mathbf{w}||\mathbf{x}||f_2)$
      Compute $z_x \leftarrow F_p(K_Z, \mathbf{x}||c||f_1)$
      Compute $y'_\mathbf{x} \leftarrow \mathsf{xid}' \cdot z_x^{-1}$
      Compute $e'_c \leftarrow g^{F_p(K_X, \mathbf{x})}$
      Set $\mathsf{xtag}' \leftarrow g^{F_p(K_X, \mathbf{w}||\mathbf{x}) \cdot \mathsf{xid}'}$
      Add xtag' to XSet
      Append $(y'_\mathbf{x}, e'_c)$ to t' and set $c \leftarrow c + 1$
    Set $T[\mathbf{w}||f_2] \leftarrow t'$
  Compute $\{\mathsf{TSet}, K_T\} \leftarrow \mathsf{TSet.SetUp}(T)$
  $\mathbf{EDB}_\mathsf{II} = (\mathsf{TSet}, \mathsf{XSet})$
  $\mathbf{EDB} \leftarrow \mathbf{EDB} \cup \{\mathbf{EDB}_\mathsf{II}\}$
  return $\mathsf{mk} = \{K_S, K_I, K_Z, K_X, K_T\}$, $\mathbf{EDB}$

4.4 Proof of Correctness

The proof of correctness for SEC follows from the correctness of $\text{SEC}_{\mathsf{Basic}}$ and the underlying CSSE scheme i.e. OXT. Consider a function composition of the form $f_1(\mathbf{w}, f_2(\mathbf{x}_1, \mathbf{x}_2))$, where $\mathbf{w}, \mathbf{x}_1, \mathbf{x}_2$ are keywords (input bit to the function). The evaluation of $f_2(\mathbf{x}_1, \mathbf{x}_2)$ should return a pseudo-document which is essentially a (partial) search token e'. The output of this evaluation will return a single pseudo-document, hence a single search token will be returned which is used as the second input for f_1.

$$e' = f_2(\mathbf{x}_1, \mathbf{x}_2),$$

As soon as the server retrieves e' (partial search token) it computes the final search token (xtoken) for f_1 by raising $(e')^r$, where from Algorithm 11 r is observed as an element in $\mathbb{Z}_p^*$. Once the search token is generated, $f_1(\mathbf{w}, \mathsf{xtoken})$ is evaluated by using OXT as the underlying CSSE scheme in $\text{SEC}_{\mathsf{Basic}}$. The output is returned as a singleton set consisting of an encrypted document which essentially encapsulates the resultant bit after evaluating $f_1(\mathbf{w}, f_2(\mathbf{x}_1, \mathbf{x}_2))$.
By correctness of OXT we claim that the search token generated obliviously at the server will retrieve the correct pseudo-document corresponding to the particular query.

4.5 Computation and Storage Overhead

Computation Overhead. The evaluation time of SEC for computing arbitrary depth Boolean circuit over encrypted data scales linearly with the search time complexity of the underlying conjunctive SSE scheme (OXT) times some constant which depends upon the

Algorithm 11 SEC.EVALUATE

Input: $f_1, f_2, \mathbf{w}, \mathbf{x}_1, \mathbf{x}_2, \mathsf{mk}, \mathbf{EDB}$
Output: $\mathsf{y} = f_1(\mathbf{w}, f_2(\mathbf{x}_1, \mathbf{x}_2))$

function SEC.EVALUATE($f_1, f_2, \mathbf{w}, \mathbf{x}_1, \mathbf{x}_2, \mathsf{mk}, \mathbf{EDB}$)
  ▷ *Tweak* OXT_SEARCH *as follows -*
  Client
  Computes $\mathsf{stag}' \leftarrow \mathsf{TSet.GetTag}(K_T, \mathbf{x}_1||f_2)$; $\mathsf{stag} \leftarrow \mathsf{TSet.GetTag}(K_T, \mathbf{w})$ and sends to Server.
  Computes xtoken' as follows: ▷ *for* f_2
  for $c = 1$ *to* $|\mathbf{DB}(\mathbf{x}_1)|$ **do**
    for $\mathbf{x} \in \mathcal{W} \setminus \{\mathbf{x}_1\}$ **do**
      Computes $\mathsf{xtoken}'[c, x] \leftarrow g^{F_p(K_Z, \mathbf{x}||c|f_1) \cdot F_p(K_X, \mathbf{w}||\mathbf{x}||f_2)}$
    Sets $\mathsf{xtoken}'[c] \leftarrow (\mathsf{xtoken}'[c, \mathbf{x}])$ and sends to Server.
  for $c' = 1$ *to* $|\mathbf{DB}(\mathbf{w})|$ **do**
    Sets $r \leftarrow Fp(K_Z, \mathbf{w}||c'||f_1)$ and sends to Server
  Server
  $t' \leftarrow \mathsf{TSet.Retrieve}(\mathsf{TSet}, \mathsf{stag}')$; $t \leftarrow \mathsf{TSet.Retrieve}(\mathsf{TSet}, \mathsf{stag})$
  for $c = 1 : |t'|$ **do**
    Server recovers $(y'_{\mathbf{x}_1}, e'_c)$ from c-th component of t'.
    for $l = 2 : n$ **do**
      Server computes $\mathsf{xtag}' = \mathsf{xtoken}'[c, l]^{y'_{\mathbf{x}_1}}$ ▷ *for* f_1
    if for all $l \in [2, n], \mathsf{xtag}' \in \mathsf{XSet}$ **then**
      Computes $\mathsf{xtoken} = (e'_c)^r$
    When last tuple in t' is reached, send *stop* to client and halt.
  for $c = 1 : |t|$ **do**
    Server recovers $(y_{\mathbf{w}}, e_c)$ from c-th component of t.
    for $l = 2 : n$ do **do**
      Server computes $\mathsf{xtag} = \mathsf{xtoken}[c, l]^{y_{\mathbf{w}}}$
    if for all $l \in [2, n]$, $\mathsf{xtag} \in \mathsf{XSet}$ **then**
      Send e_c to the client.
  Client
  $\mathsf{y} \leftarrow \mathsf{Decrypt}(e_c)$
Client obtains y which is equal to the output of $f_1(\mathbf{w}, f_2(\mathbf{x}_1, \mathbf{x}_2))$.

depth of the circuit. For evaluating an n-bit Boolean gate, we require an arbitrary number of 2-input equivalent gates. SEC is extremely efficient and fast with average time complexity in order of milliseconds for evaluating a function composition of the form $f_i(f_j(\cdot), f_k(\cdot))$ over encrypted data.

Storage Overhead. The storage required for SEC depends upon the number of functions the scheme claims to support along with the possible combinations of function composition (as discussed in Section 4. Each function specific sub-database consists of exactly eight keyword-document pairs. The storage for each sub-database will scale with the total number of keyword-document pairs. With a set of three universal gates (functions) there are nine possible combinations in which any two function can be composed. Hence, for a set of functions supported by SEC i.e, $f = \{\mathsf{f}_{\mathtt{AND}}, f_{\mathtt{OR}}, f_{\mathtt{XOR}}\}$, the space overhead will be $|f|^2 \times |(\mathsf{keyword\text{-}id})|$ pairs per sub-database, which is a constant. It is evident from the above discussion that the storage required is extremely small and highly optimized.

5 Security of SEC

We informally analyse the security of SEC construction here. We follow a semi-honest adversarial setting for this analysis where the remote server is assumed to be a honest-but-curious entity. That implies the untrusted server follows the algorithmic specification exactly, but can also observe and record additional information for analysis.

SEC inherits security properties and leakage profile from the underlying OXT construction. We assume OXT construction is an adaptively secure sub-linear conjunctive SSE algorithm which is secure against a semi-honest adversary $\mathcal{A}$ and the leakage of OXT is characterised by the leakage function $\mathcal{L}_{\mathsf{OXT}}$. The leakage function $\mathcal{L}_{\mathsf{OXT}}$ is an ensemble of the leakage functions for Setup and Search individually, expressed in the following way.

$$\mathcal{L}_{\mathsf{OXT}} = \{\mathcal{L}^{\text{Setup}}_{\mathsf{OXT}}, \mathcal{L}^{\text{Search}}_{\mathsf{OXT}}, \}$$

Given the above OXT leakage functions, security of SEC can be analysed using SEC leakage function $\mathcal{L}_{\text{SEC}}$ in the same adaptive semi-honest adversarial model. Similar to $\mathcal{L}_{\mathsf{OXT}}$, $\mathcal{L}_{\text{SEC}}$ is composed of two separate leakage functions for Setup and Evaluate, as expressed below, that capture the leakage from SEC execution in the meta-keyword setting.

$$\mathcal{L}_{\text{SEC}} = \{\mathcal{L}^{\text{Setup}}_{\text{SEC}}, \mathcal{L}^{\text{Evaluate}}_{\text{SEC}}, \}$$

Concretely, $\mathcal{L}_{\text{SEC}}$ is identical to $\mathcal{L}_{\mathsf{OXT}}$ with f (the set of functions) as an additional benign component. In other words, we show that $\mathcal{L}_{\text{SEC}}$ is equal to $\bar{\mathcal{L}}_{\mathsf{OXT}}$ where $\bar{\mathcal{L}}_{\mathsf{OXT}}$ is $\mathcal{L}_{OXT}$ in the context of input bits and $f = \{f_{\texttt{AND}}, f_{\texttt{OR}}, f_{\texttt{NOT}}\}$. At a high level, $\mathcal{L}^{\text{Setup}}_{\text{SEC}}$ incorporates **DB** generated by the GenDB during SEC.Setup. Similarly, the evaluation leakage encapsulates leakages from input bits and function to be evaluated. We quantify this through a leakage function instance that encapsulates the leakage during an evaluation of a binary function over encrypted input bits. We show that this leakage is covered by the same as of the OXT construction. Due to the uniform keyword frequency SEC restricts certain non-trivial leakages like *size-pattern*, *result-pattern* leakages that analyses the frequency pattern of the sterm and xterm in OXT over multiple queries. .

5.1 Leakage Profile Analysis of SEC

Leakage Profile of SEC. The significance of each component of the leakage function in SEC is equivalent to that in OXT except for a few which we point out subsequently. We define each leakage component as follows.

- $N = \sum_{i=1}^{d} |\mathcal{W}_i|$ - the total number of appearances of keywords in documents. The parameter N signifies an upper bound which is equivalent to the total size of **EDB**. Leaking such a bound is unavoidable and is considered as a trivial leakage in literature of SSE.

- $\bar{s} \in [m]^n$ - equality pattern of $s \in \mathcal{W}^n$ that indicates which queries have the equal sterms. Repetition of sterms of different queries is leaked by $\bar{s}$. This occurs due to the optimization technique devised in OXT in order to ensure sublinear search complexity by filtering out the least frequent term during search. In SEC this leaks the first input of two functions if they are similar.

- SP - size pattern of the queries i.e., the number of documents matching the sterm in each query. Formally, $\mathsf{SP} \in [d]^n$ and $\mathsf{SP}[i] = |\mathbf{DB}(s[i])|$. It leaks the number of documents satisfying the sterm in a query. In SEC this is always constant (equal to 2).
- RP - result pattern of the queries or the indices of documents matching the entire conjunction. Formally, RP is vector of size n with $\mathsf{RP}[i] = \mathbf{DB}(s[i]) \cap \mathbf{DB}(x[i])$ for each i. It is the final output of the search query and is not considered as a real leakage in the context of SSE. This is always a single document in SEC.

Leakage Mitigation in SEC. The strategic database generation process of SEC, tends to mitigate certain non-trivial information leakage which is otherwise leaked by the underlying OXT scheme. Due to the uniform keyword frequency SEC restricts certain non-trivial leakages like *size-pattern*, *result-pattern* leakages that analyses the frequency pattern of the sterm and xterm in OXTover multiple queries. Furthermore, a crucial yet subtle leakage in OXT called the *Conditional Intersection Pattern leakage* is prevented in SEC due to the design structure of the database.

Size Pattern Leakage. It leaks the number of documents satisfying the sterm in a query. In SEC since every keyword maps to exactly two documents this leakage reveals no significant information as every sterm has the same frequency.

Result Pattern Leakage. It is the final output of the search query i.e. indices of documents matching the entire conjunction. By the design of SEC the result of a conjunctive query is always a single document and hence this does not reveal any significant information to the server. Although it is not considered as a real leakage in the context of SSE, SEC prevents this leakage which is otherwise present in OXT.

Conditional Intersection Pattern Leakage. It is a subtle leakage in OXT that occurs when two distinct queries have a common xterm but different sterm and there exists a document that satisfies both the sterms. In such a scenario the set of document indices matching both sterms is leaked (if no document matching both sterms exist then nothing is leaked). This leakage is not present in our SEC as by the design of the encrypted database, two different sterms will never have a document in common (c.f. Table 1).

Resistance to Volume Attacks. The uniform frequency distribution of the keywords in the database makes SEC robust against volume-based attacks even without padding or using volume-hiding encrypted multimaps. Such leakage abuse attacks pose serious threats to many existing conjunctive SSE scheme in literature.

Ordering of function composition. Apart from hiding the size pattern, result pattern leakages thereby resisting volume-based attack, SEC also conceals the order in which the functions are composed. Since the function to be evaluated is encapsulated in input to a PRF, the server cannot distinguish which particular function is being evaluated during a particular search operation. This is ensured by the PRF security guaranteed by OXT. Therefore, SEC is able to compute arbitrary functions over encrypted database without

revealing the inputs as well as the function being evaluated to the server, which is leaked in the state-of-the-art FHE schemes.

Theorem 1 (Security of SEC) *Given that* OXT *is an adaptively secure SSE scheme with respect to the leakage function* $\mathcal{L}_{\text{OXT}} = \{\mathcal{L}_{\text{OXT}}^{\text{Setup}}, \mathcal{L}_{\text{OXT}}^{\text{Search}}\}$ *against a polynomially-bound adaptive adversary,* SEC *is also an adaptively secure encrypted computation scheme with respect to the leakage function* $\mathcal{L}_{\text{SEC}} = \{\mathcal{L}_{\text{SEC}}^{\text{Setup}}, \mathcal{L}_{\text{SEC}}^{\text{Evaluate}}\}$*, where the* SEC *instantiation operates on a plaintext database* **DB** *and a pair of two-input functions* $\{f_1, f_2\}$*.*

Remark. Note that, even though we outline Theorem 1 for depth two (comprising of two functions f_1 and f_2) for the ease of exposition, the extension to depth n is straightforward.

Proof. The security analysis of SEC (proof of Theorem 1) stems from the provable security guarantee of OXT. We first outline the leakage sources of OXT, with respect to which OXT is simulation secure. Subsequently, we show that the leakage profile of SEC is covered by the same leakage profile with respect to the workflow of SEC, and the simulator can simulate the real instance of SEC just from the leakage information available.

We resort to the same simulation-based security analysis approach for SEC as of OXT. We show that SEC is secure against an adaptive semi-honest adversary $\mathcal{A}$, which has access to leakages from $\mathcal{L}_{\text{SEC}}$. We build a simulator SIM for SEC **Real** experiment where the simulator emulates SEC execution just from the knowledge of public information and leakage $\mathcal{L}_{\text{SEC}}$.

Leakage Cover. We briefly describe the significance of each individual leakage components of $\mathcal{L}_{\text{SEC}}$ comprising of $N, \bar{s}, \mathsf{SP}, \mathsf{RP}$ for correct simulation results by the simulator. In order to simulate SEC correctly each of the leakage components are critically analysed and their significance is justified. N or the total number of appearances of keywords in the database gives the size of the XSet, i.e. number of xtag entries for each keyword-id pair in the database. The equality pattern $\bar{s}$ is important as it indicates the queries with same stags. By the design of OXT the stags are deterministic hence the server can observe repetition in stags of different queries. In order to know the number of documents matching the sterm, the *size pattern* leakage component is important. In SEC SP is always equal for all queries. SP is equal to the number of tuples returned by the TSet. SEC does not incur conditional intersection pattern leakage by virtue of its design strategy. Therefore for evaluating functions of the form $f(\mathbf{x}_1, \mathbf{x}_2) = q = \mathsf{st}_1 \wedge \mathsf{st}_2$, the server cannot observe the queries in which the xterms $(\mathbf{x}_2)$ repeat and that have a document identifier common in their stag $(\mathbf{x}_1)$.

Simulating SEC Setup and Evaluate. For the adaptive proof of security, we assume an adaptively secure instantiation of TSet in the standard model. While the original construction of TSet [45] requires random oracles for adaptive security, the authors of [45] also discuss an alternative instantiation of adaptively secure TSets in the standard model without incurring additional rounds of communication. In the context of our SEC protocol, using the standard model instantiation of TSet increases the communication overhead for the TSet component, but not the (asymptotic) communication overhead for the overall search protocol. The main crux of our adaptive security proof is that the simulator for SEC initializes the XSet to consist entirely of uniformly random elements initially (while relying

on the DDH assumption for indistinguishability of the real and simulated XSet entries). Additionally, the simulator for SEC can directly invoke the simulator for the adaptively secure TSet to simulate the TSet entries at setup and the corresponding TSet tokens during searches. The simulator also uses the (adaptive) result pattern leakage to program the xtoken entries to be consistent with the adversarially issued queries. Overall SIM takes as input the leakage components as defined by $\mathcal{L}_{\text{SEC}}$ and produces the result pattern which is similar to the document identifiers returned by the original scheme on input of an adaptive conjunctive query.

***Simulating* SEC.Setup.** To simulate SEC Setup, observe that, SEC Setup comprises of the GenDB, OXT.Setup, and a number of values generated through PRF and group operations, which are finally inserted into TSet using TSet.Setup. Note that, GenDB routine creates the plain look-up table for the supported primitive operations, and it is executed on the client side. Hence, the adversarial server learns no information from the GenDB execution itself and thus the leakage from GenDB can be expressed as null.

$$\mathcal{L}_{\text{SEC}}^{\text{Setup,GenDB}} = \perp,$$

Thus, the simulator SIM can exactly simulate GenDB execution straightforwardly.

Subsequently, the OXT.Setup is invoked with the plain **DB** generated by SEC.GenDB. Since the OXT.Setup algorithm is executed in black-box way, the leakage from this phase of SEC.Setup is same as the OXT.Setup executed over **DB**. Thus, the leakage for this phase can be expressed as below.

$$\mathcal{L}_{\text{SEC}}^{\text{Setup,Phase-I}}(\mathbf{DB}) = \mathcal{L}_{\text{OXT}}^{\text{Setup}}(\mathbf{DB}),$$

Therefore, SIM can run execute OXT.Setup internally to simulate Phase-I of SEC.Setup.

The execution of SEC.Setup Phase-II follows a sequence of PRF evaluations and group operations. Essentially, during simulation, each of the PRF evaluations can be replaced with a uniformly randomly sampled value from the PRF space, and each group operation output can be replaced by a uniformly randomly sampled element of the group. This is executed entirely on the client side, and the server learns nothing about the computation of this phase.

$$\mathcal{L}_{\text{SEC}}^{\text{Setup,Phase-II}} = \perp,$$

Hence, the simulator SIM can simulate Phase-II by replacing the PRF and group operations.

Finally, the TSet Setup execution does not leak additional information apart from already known public information (the size of the dictionary $|\mathcal{W}|$ is known). The leakage for this part can be expressed as below.

$$\mathcal{L}_{\text{SEC}}^{\text{Setup},\mathsf{TSet}} = |\mathcal{W}| = \perp \text{ as this is a public information,}$$

SIM can run the TSet simulator (as discussed in the original paper [45]). Combined all, the simulator for Setup ($\text{SIM}_{\text{Setup}}$) simulates SEC Setup with access to following the leakage.

$$\begin{aligned}\mathcal{L}_{\text{SEC}}^{\text{Setup}} =& \{\mathcal{L}_{\text{SEC}}^{\text{Setup,GenDB}}, \mathcal{L}_{\text{SEC}}^{\text{Setup,Phase-I}}(\mathbf{DB}), \mathcal{L}_{\text{SEC}}^{\text{Setup,Phase-II}},\\ & \mathcal{L}_{\text{SEC}}^{\text{Setup},\mathsf{TSet}}\} \text{ for } f_{\mathtt{AND}}, f_{\mathtt{OR}}, f_{\mathtt{XOR}} \text{ data in } \mathbf{DB}\\ =& \mathcal{L}_{\text{OXT}}^{\text{Setup}}(\mathbf{DB}),\end{aligned}$$

***Simulating* SEC.Evaluate.** For simulating SEC EVALUATE, we observe that the EVALUATE routine of SEC is similar to OXT with specific additional steps. Thus, we write the additional hybrids needed to simulate SEC EVALUATE on top of the existing hybrids of OXT, as presented in [45].

First, the client side computation in Algorithm 11 incorporates xtoken' computation as additional steps from OXT. Thus we need to introduce additional hybrids into the sequence of games as presented in [42]. Note that, this particular game (additional) considers the view of adversary before and after modifying the steps from line 4 to line 10 in Algorithm 11.

Game a_0 . View of the adversary till line 4 according to the sequence of games of OXT in [45].

Game a_1 . View of the adversary till line 10 after replacing all F_p instances with values sampled uniformly at random from the range of F_p.

Lemma 1 *Game* a_0 *and Game* a_1 *are computationally indistinguishable, given* F_p *is a secure PRF.*

Proof. By the indistinguishability property of a PRF output from a random value, Game a_0 and Game a_1 are identical.

Thus, SIM can simulate EVALUATE with the leakage

$$\mathcal{L}_{\text{SEC}}^{\text{EVALUATE}} = \mathcal{L}_{\text{OXT}}^{\text{SEARCH}}(f_1, f_2) \text{ for } f_{\texttt{AND}}, f_{\texttt{OR}}, \text{ and } f_{\texttt{XOR}} \text{ data},$$

This implies the **Real** experiment of SEC (Algorithm 1) is indistinguishable from the **Ideal** experiment (Algorithm 2), and proves Theorem 1.

6 Experimental Results

In this section, we report on a prototype implementation of SEC and $\text{SEC}_{\mathsf{Basic}}$ and compare it with a prototype implementation of the TFHE library [38], which implements an efficient and fast gate-by-gate bootstrapping [32].

Implementation Details. Our prototype implementations are developed in C++ and we used Redis as the database backend. More specifically, we realize all PRF operations using AES-256 in counter mode, BLAKE3 hash function for computing all hash operations and all group operations in SEC and $\text{SEC}_{\mathsf{Basic}}$ over the elliptic curve Curve25519 [52]. We implement the TSet data structure using Redis, which serves as a key-value store, while the XSet dictionary is realized using a Bloom filter [53].

Platform. For our experiments, we used a *single* node with 64-bit Intel Xeon Silver 4214R v4 3.27GHz processors, running Ubuntu 20.04.4 LTS, with 128GB RAM and 1TB SSD hard disk.

Evaluation of Storage Overhead. As discussed in Section 4.5 the storage required for SEC depends upon the number of functions supported along with all possible combinations of function composition. Each function specific sub-database consists of exactly eight keyword-document pairs (since we consider 1-bit binary functions). For a set of functions supported by SEC i.e, $f = \{\mathsf{f}_{\mathtt{AND}}, f_{\mathtt{OR}}, f_{\mathtt{XOR}}\}$, the storage overhead is $|f|^2 \times |(\mathsf{keyword\text{-}id})|$ pairs per sub-database, which is a constant. In our implementations the server storage required to store TSet and XSet is around 63 KB. We thus note that SEC is highly optimized and scalable with significantly less storage requirements than state-of-the- art FHE schemes. Table 3 compares the storage overhead comparison of SEC with state-of-the-art FHE schemes.

Table 3: Storage Overhead comparison (in MB) of SEC with existing FHE schemes in literature. Storage overhead of FHE scheme typically indicate the bootstrapping key size where as for SEC it implies the size of the encrypted database stored at the cloud server.

Scheme	Storage Overhead (in MB)
Gentry et. al[25]	3700
Gentry et. al[29]	2300
Halevi et. al.[34]	1600
Ducas et. al[33]	1000
Chillotti et. al.[32]	24
SEC	0.063

Evaluation of computation time. The evaluation time of $\mathsf{SEC}_{\mathsf{Basic}}$ for computation of basic binary function of the form $f_i(\mathbf{x}_1, \mathbf{x}_2)$ (where $f_i \in f = \{\mathtt{XOR}, \mathtt{AND}, \mathtt{OR}\}$ and $\mathbf{x}_1$ and $\mathbf{x}_2$ are encrypted input bits) scales linearly with the search time complexity of OXT. According to OXT the time required to retrieve the documents corresponding to a conjunctive query

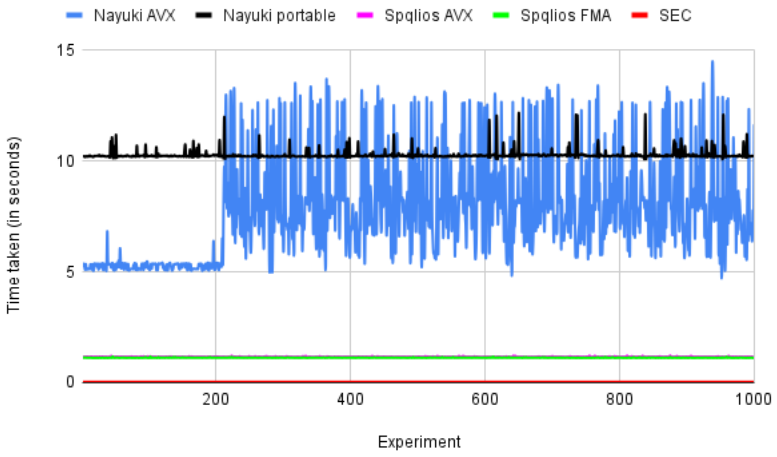

Figure 4: Time taken (in seconds) for 1000 invocations of $\mathsf{SEC}_{\mathsf{Basic}}$.Evaluate against different TFHE backends.

scales with the frequency of the least frequent keyword in the query. Since the database under consideration is very small the search time of OXT would be significantly less. The average time required by SEC for one binary function evaluation is extremely fast, in order of milliseconds. Figure 4 plots the time taken for different iterations of $\mathsf{SEC}_{\mathsf{Basic}}$.Evaluate (note that $\mathsf{SEC}_{\mathsf{Basic}}$.Setup is a one-time process, and thus its time is not included in our analysis).

The evaluation time of SEC for computing arbitrary depth Boolean circuit over encrypted data also scales linearly with the search time complexity of OXT times some constant which depends upon the depth of the circuit. For evaluating an n-bit Boolean gate, we require an arbitrary number of 2-input equivalent gates. Our experimental results validates that SEC is highly efficient and fast with sub-linear time complexity for evaluating arbitrary Boolean function of the form $f_i(f_j(\cdot), f_k(\cdot))$ over encrypted data. Figure 5 compares the execution time of SECwith different TFHE backends for varying depth of circuits.

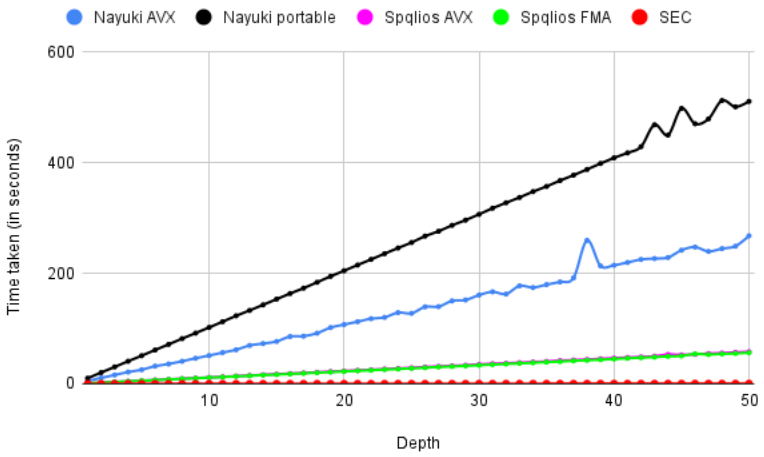

Figure 5: Time taken (in seconds) for different circuit depths of SEC.Evaluate against different TFHE backends.

Table 4: Time taken (in seconds) for evaluation of one byte AES SBox by SEC.Evaluate against different TFHE backends.

Scheme	Time taken (in seconds)
Nayuki AVX	214.63
Nayuki Portable	408.968
Spqlios AVX	46.6688
Spqlios FMA	44.948
SEC	0.38664

Comparison with FHE. We compare $\mathsf{SEC}_{\mathsf{Basic}}$ and SEC with different variations of TFHE in Figure 4. One variation is Nayuki portable (non AVX) and AVX builds, which

implement very efficient versions of Fast Fourier Transform. Another back-end family is spqlios AVX and spqlios FMA back-ends, which are efficient assembly implementations of ring operations. It is observed from Figure 4 $\text{SEC}_{\mathsf{Basic}}$ is $10^3\times$ faster than all four TFHE backends that we compared in our experiments. Figure 5 compares the increase of execution time with an increase in the depth of the circuit. SEC performs function evaluation of arbitrary depth in order of milliseconds which is again $10^3\times$ improvement over the fastest TFHE backend using Spqlios AVX optimization. Furthermore, in table 4, we also show a practical use case of the AES SBox[5], wherein SEC again outperforms various TFHE backends.

7 Conclusion and Future Work

Encrypted computation enables us to develop complex privacy-preserved solutions for practical applications with outsourced processing, for instance - IoT networks, home automation, and machine learning inference, to name a few. However, the existing solutions, like FHE or MPC, are prohibitively computation-intensive and do not scale to large real applications. In this work, we introduce a fast, scalable, and efficient technique for encrypted computation called SEC. Concretely, SEC transforms primitive functions or operations into encrypted look-ups and uses an efficient conjunctive SSE scheme to process the encrypted look-ups necessary to evaluate a function privately. SEC is extremely fast due to the use of an underlying conjunctive SSE scheme based on fast classical symmetric-key cryptographic primitives and incurs minimal encrypted storage overhead that is linear in the size of the look-up table of the primitive operations. This way, SEC executes composition of functions almost 10^3 times faster compared to FHE and incurs 5.8×10^4 times less storage overhead than FHE in our experiments.

In this paper, we presented the core SEC construction outlining the fundamental idea of encrypted computation via encrypted look-ups. Our current construction is developed from state-of-the-art conjunctive SSE scheme built using classical cryptographic primitives. However, the rising threat of quantum computers against classical encryption primitives poses a severe concern for future applications. Therefore, developing a post-quantum secure version of SEC is a prominent research direction with immense practical relevance and we leave this as an interesting future work.

References

[1] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*. IEEE, 2015, pp. 896–902.

[2] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107496, 2020.

[3] X. Yuan, H. Tian, H. Wang, H. Su, J. Liu, and A. Taherkordi, "Edge-enabled wbans for efficient qos provisioning healthcare monitoring: A two-stage potential game-based computation offloading strategy," *IEEE Access*, vol. 8, pp. 92 718–92 730, 2020.

[5]For fair evaluation, we use *unparallelized* version of the SBox [54], which involves 5 XORs per bit in the output, thereby totalling 40 XORs for the entire byte.

[4] Y. Qiu, H. Zhang, and K. Long, "Computation offloading and wireless resource management for healthcare monitoring in fog-computing-based internet of medical things," *IEEE Internet of Things Journal*, vol. 8, no. 21, pp. 15 875–15 883, 2021.

[5] X. Li, R. Ding, X. Liu, W. Yan, J. Xu, H. Gao, and X. Zheng, "Comec: Computation offloading for video-based heart rate detection app in mobile edge computing," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications*. IEEE, 2018.

[6] R. Chaari, O. Cheikhrouhou, A. Koubâa, H. Youssef, and H. Hmam, "Towards a distributed computation offloading architecture for cloud robotics," in *2019 (IWCMC)*. IEEE, 2019.

[7] Z. Hong, H. Huang, S. Guo, W. Chen, and Z. Zheng, "Qos-aware cooperative computation offloading for robot swarms in cloud robotics," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 4, pp. 4027–4041, 2019.

[8] A. Koubâa, A. Ammar, M. Alahdab, A. Kanhouch, and A. T. Azar, "Deepbrain: Experimental evaluation of cloud-based computation offloading and edge computing in the internet-of-drones for deep learning applications," *Sensors*, vol. 20, no. 18, p. 5240, 2020.

[9] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, D. Zou, H. Jin, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation." in *USENIX Security Symposium*, 2020.

[10] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *28th USENIX Security Symposium*, 2019.

[11] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[12] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service." *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 3, pp. 123–142, 2018.

[13] H. C. Tanuwidjaja, R. Choi, S. Baek, and K. Kim, "Privacy-preserving deep learning on machine learning as a service—a comprehensive survey," *IEEE Access*, vol. 8, pp. 167 425–167 447, 2020.

[14] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 264–278.

[15] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[16] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[17] R. Ostrovsky, "Efficient computation on oblivious rams," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 514–523.

[18] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, 1986, pp. 162–167.

[19] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *TOC, 1987*, A. V. Aho, Ed. ACM, 1987.

[20] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings 8*. Springer, 2011, pp. 253–273.

[21] C. Mascia, M. Sala, and I. Villa, "A survey on functional encryption," *arXiv preprint arXiv:2106.06306*, 2021.

[22] Z. Chang, D. Xie, and F. Li, "Oblivious ram: A dissection and experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, 2016.

[23] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, H. N. Gabow and R. Fagin, Eds., 2005.

[24] Z. Brakerski, C. Gentry, and S. Halevi, "Packed ciphertexts in lwe-based homomorphic encryption," in *Public-Key Cryptography–PKC 2013*. Springer, 2013, pp. 1–13.

[25] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *CRYPTO 2012*. Springer, 2012.

[26] ——, "Fully homomorphic encryption with polylog overhead." in *Eurocrypt*, vol. 7237. Springer, 2012, pp. 465–482.

[27] ——, "Better bootstrapping in fully homomorphic encryption," in *PKC 2012*. Springer, 2012, pp. 1–16.

[28] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart, "Ring switching in bgv-style homomorphic encryption," in *SCN*. Springer, 2012.

[29] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *EUROCRYPT 2011*. Springer, 2011.

[30] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 2014, pp. 297–314.

[31] ——, "Practical bootstrapping in quasilinear time," in *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer, 2013, pp. 1–20.

[32] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT 2016*. Springer, 2016, pp. 3–33.

[33] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *EUROCRYPT 2015*. Springer, 2015, pp. 617–640.

[34] S. Halevi and V. Shoup, "Design and implementation of helib: a homomorphic encryption library," *Cryptology ePrint Archive*, 2020.

[35] F. Boemer, S. Kim, G. Seifu, F. DM de Souza, and V. Gopal, "Intel hexl: accelerating homomorphic encryption with intel avx512-ifma52," in *WAHC*, 2021, pp. 57–62.

[36] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *WAHC*, 2022, pp. 53–63.

[37] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT 2017*. Springer, 2017, pp. 409–437.

[38] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption library," 2019.

[39] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE symposium on security and privacy. S&P 2000*. IEEE, 2000, pp. 44–55.

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

[41] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT 2010*. Springer, 2010, pp. 577–594.

[42] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO*. Springer, 2013.

[43] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*, 2014.

[44] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *NDSS 2021*, 2021.

[45] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*, 2013, pp. 353–373.

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

[47] G. S. Poh, J.-J. Chin, W.-C. Yau, K.-K. R. Choo, and M. S. Mohamad, "Searchable symmetric encryption: designs and challenges," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–37, 2017.

[48] R. Dowsley, A. Michalas, M. Nagel, and N. Paladi, "A survey on design and implementation of protected searchable data in the cloud," *Computer Science Review*, vol. 26, pp. 17–30, 2017.

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

[50] C. Jutla and S. Patranabis, "Efficient searchable symmetric encryption for join queries," *Cryptology ePrint Archive*, 2021.

[51] A. El-Yahyaoui and M. D. E. E. Kettani, "A verifiable fully homomorphic encryption scheme for cloud computing security," *CoRR*, 2018.

[52] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC*, ser. Lecture Notes in Computer Science, 2006.

[53] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, 1970.

[54] N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "A systematic evaluation of compact hardware implementations for the rijndael s-box," in *Topics in Cryptology–CT-RSA 2005*. Springer, 2005.