SEC: Symmetric Encrypted Computation via Fast Look-ups

Debadrita Talapatra
IIT Kharagpur, India

Nimish Mishra
IIT Kharagpur, India

Arnab Bag
IMEC Belgium

Sikhar Patranabis
IBM Research India

Debdeep Mukhopadhyay
IIT Kharagpur, India

February 14, 2025

Abstract

Encrypted computation allows a client to securely outsource the storage and processing of sensitive private data to an untrusted third party cloud server. Fully Homomorphic Encryption (FHE) and Garbled Circuit (GC) are state-of-the-art general-purpose primitives that support encrypted computation. FHE enables arbitrary encrypted computation ensuring data-privacy but suffers from huge computation overhead and poor scalability. GC additionally provides function privacy, but is often not suitable for practical deployment because its translation tables need to be refreshed for every evaluation. We extend Searchable Symmetric Encryption (SSE), beyond encrypted searches, to perform *arbitrary* Boolean circuit evaluations over symmetrically encrypted data via look-ups, while ensuring data privacy.

In this work, we propose Symmetric Encrypted Computation (SEC), the first practically efficient and provably secure lookup-based construction that supports evaluation of arbitrary Boolean circuits over symmetrically encrypted data, while ensuring data-privacy guarantees. With a single setup of encrypted look-up tables, SEC supports $O(n!)$ re-evaluations of a circuit of size n. This is an improvement on GC that supports a single evaluation with a single setup. While SEC supports *bounded* re-evaluations with a single setup (unlike FHE), it is asymptotically large enough to scale to practical deployments of realistic circuits. Moreover, SEC completely bypasses bootstrapping thereby significantly improving on performance efficiency. SEC achieves this by relying on purely symmetric-key crypto primitives by extending and generalizing the functional capabilities of SSE, while inheriting its desirable performance benefits. The leakages incurred by underlying SSE scheme are rendered inconsequential with respect to SEC's security guarantees due to its meticulous design choices.

We provide a concrete construction of SEC and analyze its security with respect to a rigorous leakage profile. We also experimentally validate its practical efficiency. SEC outperforms state-of-the-art FHE schemes (such as Torus FHE) substantially, with around 1000× speed-up in basic Boolean gate evaluations. We further showcase the scalability of SEC for functions with multi-bit inputs via experiments performing encrypted evaluation of the entire AES-128 circuit, as well as three max-pooling layers of AlexNet architecture. For both sets of experiments, SEC outperforms state-of-the-art and accelerated FHE implementations by 1000× in terms of processing time, while incurring 250× lower storage.

Contents

1 Introduction

Outsourced Storage. The upswing in data production in today's digitally-driven world has motivated the concept of outsourcing data to third-party cloud servers for storage. However, such *outsourced storage* solutions are often plagued by security breaches that lead to disclosure of client data [24, 52, 54]. Without specific privacy mechanisms, third-party cloud servers could gain access to sensitive user data, thus leading to serious privacy concerns. This establishes the requirement of adopting secure and scalable privacy mechanisms for protecting sensitive outsourced data from unauthorized access. A straightforward solution to this problem is to encrypt this data before offloading to the third-party server, thereby ensuring data privacy and preventing disclosures. However, this leads to the challenge of securely computing queries (or more generally, executing functions/programs) directly on the encrypted data without decrypting it first.

"Secure Computation" on Encrypted Outsourced Data. The question of privacy-preserving computation on encrypted, outsourced data has been studied extensively in the cryptographic literature. There exist elegant solutions such as Fully Homomorphic Encryption (FHE), Functional Encryption (FE), and Multi-party computation (MPC), all of which vary in terms of adversarial structure, communication models, and security guarantees. Classic FHE [13, 28, 34] works in the single client, single (adversarial) server setting and supports evaluating *any* (poly-sized) circuit directly over encrypted data, but the adversary does not learn anything about the data without the knowledge of the secret decryption key. Leveled Homomorphic Encryption (LHE) [12, 14, 37, 39, 41], on the other hand, allows evaluating circuits of bounded depth without requiring bootstrapping and provides a balance between efficiency and expressiveness. However, LHE can only evaluate functions up to a fixed depth (e.g., $10 - 100$ multiplicative levels), causing a decryption failure due to excessive noise if computations exceed this depth. Traditional FE [10, 44] offers the capability of more fine-grained query evaluation on encrypted data, while only leaking the output of the computation to an adversarial server. While significant optimizations have been made to FHE schemes and its implementations in recent years [1, 2, 3, 9, 11, 20, 21, 22, 26, 29, 30, 31, 32, 33, 40], FHE and FE solutions remain computationally expensive and do not scale efficiently to large datasets in practice. MPC operates in a different setting where the client "shares" its data across *multiple* servers, with the guarantee that these servers learn nothing about the client's data apart from the output of the computation so long as the adversary does not corrupt more than a threshold number of parties. Certain MPC protocols are based on garbled circuits (GCs) [5, 36, 42], which allow hiding a circuit/program as long it is evaluated on only a single input. There exist practically efficient implementations of MPC and GCs [6, 38, 47, 49, 50], particularly in the setting where the adversary corrupts a minority of the parties (the "honest majority" setting).

Encrypted Computation via Table Lookups. In this paper, we focus on applications that adhere to the single server, single client setting of outsourced computation (unlike MPC). In addition, we consider applications where the program/circuit being evaluated on the encrypted data does not need to be private (unlike GCs). This is in line with the traditional FHE setting, where the focus is to maintain the privacy of the plaintext inputs to the function as well as the privacy of output of the function, against a (semi-honest) corrupt

Properties	GC	LHE	SEC (This Work)	FHE
Reusability	One-time SFE	Bounded ($10 \sim 50$ levels [39, 41])	Bounded ($O(n!)$)	Unbounded
Bootstrapping	✗	✗	✗	✓
Computation Model	Lookup Table	Circuit Evaluation	Lookup Table	Circuit Evaluation
Interaction Round	Setup for every evaluation	One-time Setup	One-time Setup	One-time Setup
Privacy Guarantee	SFE + Data Privacy	Data Privacy	Data Privacy	Data Privacy
Security Assumption	Symmetric-key Encryption	Lattice-based	Symmetric-key Encryption	Lattice-based

Table 1: Comparison between state-of-the-art privacy-preserving encrypted frameworks with SEC. In the table GC: Garbled Circuit; LHE: Leveled Homomorphic Encryption; SEC: Symmetric Encrypted Computation; FHE - Fully Homomorphic Encryption; SFE: Secure Function Evaluation; n: size of the circuit, *controlled by client.* : gives a tradeoff between practical efficiency and optimal functionality; : Performance Bottleneck; : Optimal Functionality. By design, SEC supports $O(n!)$ re-evaluations for a circuit of size n with a single setup (note that n is client-controlled, since the client offloads the function description to the server). This allows SEC to completely bypass bootstrapping, thereby offering high efficiency and scalability making it suitable for large-scale practical deployments.

server[1]. In this setting, we explore the possibility of providing an accelerated FHE-style encrypted evaluation of Boolean circuits with a *table look-up* based approach of evaluating Boolean gates over encrypted binary inputs. Concretely, we ask the following question:

Can we support arbitrary Boolean circuit evaluation over encrypted data via efficient table look-ups?

We answer this question in the affirmative in the case of symmetrically encrypted outsourced data. We leverage existing approaches for evaluating restricted classes of Boolean circuits over symmetrically encrypted structured databases (called Searchable Symmetric Encryption or SSE in short [16, 19, 25]) and show, for the first time, how to use such look-up based approaches to evaluate *arbitrary* Boolean circuits over encrypted Boolean inputs (with no additional structure whatsoever). We ensure re-usability of the same look-up table for multiple gate evaluations (via formal security proofs) in a secure manner without any non-negligible information leakage. By allowing for practically viable fine-grained tradeoffs between leakage and efficiency, we design a novel framework for symmetric encrypted computation that significantly outperforms its (symmetric-key) FHE-based counterparts in terms of computational efficiency and storage requirements (we validate this via practical experiments).

1.1 Our Contributions

We introduce *Symmetric Encrypted Computation* (SEC) – a novel framework for practically efficient evaluation of arbitrary Boolean functions over symmetrically encrypted data. The technical centerpiece of SEC is a mapping of Boolean gate computations over encrypted Boolean inputs to look-ups over encrypted tables. We show how to realize such encrypted lookup computations by leveraging existing practically efficient SSE schemes that

[1]Verifiable FHE [27] considers malicious servers, but all known constructions are inefficient in practice. Security against malicious servers is outside the scope of this work.

support searching for conjunctive predicates over encrypted structured databases (several such schemes exist in the SSE literature, e.g., [16, 45]). Since SEC replaces certain algebraic operations that are inherent to any FHE scheme by table look-ups, it avoids many computationally expensive operations that limit the scalability of existing FHE solutions (most notably, bootstrapping). SEC supports arbitrary Boolean circuit evaluation of size n (client-controlled) bounded by $O(n!)$ evaluations with a single setup of the encrypted lookup tables. We present a rigorous security analysis of SEC in the widely adopted simulation security paradigm. We also present a prototype implementation of SEC and present experimental evaluations showcasing its practical efficiency as compared to the most efficient FHE implementations today. To the best of our knowledge, SEC is the *first* provably secure framework capable of arbitrary function evaluation over encrypted data using fast and efficient look-ups.

Structural Overview. SEC relies on "encoding" any arbitrary Boolean circuit as a series of lookup tables, where each table corresponds to a primitive Boolean gate. It then replaces explicit circuit computation via encrypted look-up operations by deploying a fast, encrypted search mechanism. Informally speaking, the design mechanism of SEC relies on the observation that any Boolean function can be represented by an equivalent logic circuit composed of Boolean variables and basic Boolean logic gates - $\{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$[2]. It involves creating the encrypted lookup tables for these basic gates to facilitate the evaluation of arbitrary Boolean functions. We then leverage, in a black-box way, existing SSE techniques for fast conjunctive look-ups over such encrypted lookup tables for the primitive operations (which we model as "encrypted search indices" as in standard SSE scheme. Refer to Section 2.1 for background on SSE). As it turns out, this enables extremely fast circuit evaluation surpassing the performance of the most efficient FHE schemes by several orders of magnitude. We present two concrete instantiations of the above SEC framework based on two practically efficient conjunctive SSE constructions: Oblivious Cross Tags (OXT) [17] (Refer to Section 2.3 for overview of OXT) and ConjFilter [45] (the former was the first practically efficient conjunctive SSE scheme to be proposed while plugging in non-trivial leakages, while latter supports particularly fast searches since it is based on purely symmetric-key cryptoprimitives). We call the resulting schemes $\text{SEC}_{\mathsf{OXT}}$ and $\text{SEC}_{\text{ConjFilter}}$, respectively.

Supporting Arbitrary Boolean Functions. A pivotal feature of SEC is its ability to evaluate function compositions, which is the key to evaluating arbitrary circuits. An arbitrary Boolean function can be decomposed into an expression of function composition with a lesser number of variables (following Shannon's theory, for Boolean circuits). Consequently, a complex function can be easily evaluated systematically using SEC with optimal computation overhead proportional to the circuit size (See Table 8). For a circuit of size n, SEC can compute $O(n!)$ evaluations of the circuit with a single setup of the encrypted lookup tables, without leaking any correlation between multiple evaluations. Additionally, SEC incurs nominal storage overhead for encrypted lookup tables compared to the substantial storage required to store the bootstrapping key in FHE schemes (See Table 7). Furthermore, SEC supports arbitrary function evaluations in a single round of communication with the expense of a small amount of additional storage while minimizing leakages compared to a multi-round solution. Collectively all these features render SEC practically ideal for encrypted outsourced computation frameworks.

Security Analysis. SEC relies on an efficient, adaptively secure conjunctive SSE scheme

[2] $\{\texttt{XOR}, \texttt{AND}\}$ and $\{\texttt{OR}, \texttt{AND}\}$ are functionally complete. We provide support for 3 gates for convenience in computation.

for efficient encrypted lookup-based function evaluation. However, the inherent design of SEC averts the direct extrapolation of several non-trivial leakages of the underlying SSE scheme. This in turn enables SEC to prevent correlation between multiple circuit evaluations. *We emphasize, that the same look-up table can be used for $O(n!)$ evaluations of an n-size Boolean circuit , while ensuring the adversary can infer no correlation between two gates in the same circuit, or two isomorphic gates*[3] *in different circuits with same/different encrypted inputs.* The privacy of input/output data bits is ensured by the semantic security guarantee of the IND-CPA secure symmetric-key encryption using which the bits are encrypted. We present a detailed security and leakage profile analysis of $\text{SEC}_{\mathsf{OXT}}$ following the security properties of the underlying conjunctive SSE scheme OXT. We elucidate the improvements in leakage due to our improvised construction of the lookup tables in Section 4 and Section 5.2.

Performance And Scalability. We demonstrate the efficacy and scalability of $\text{SEC}_{\mathsf{OXT}}$ and $\text{SEC}_{\text{ConjFilter}}$ by evaluating basic Boolean gates and cascaded gates as function composition. Section 6 gives a detailed analysis of our experimental evaluations. SEC decomposes any arbitrary (> 2 number of input-bits) circuit into universal binary gates $\{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$ (similar to state-of-the-art FHE schemes) and reuses this storage across multiple computations of the same gate, thus *preventing exponential storage-blowup.* We showcase scalability of $\text{SEC}_{\mathsf{OXT}}$ and $\text{SEC}_{\text{ConjFilter}}$ for functions with multi-bit inputs by using it for encrypted evaluation of the entire AES-128 circuit and three max-pooling layers of AlexNet architecture[4]. In Table 2 below, we show a practical use case of the AES SBox[5], wherein SEC outperforms various TFHE [22] backends by orders of magnitude.

Table 2: Time taken (in seconds) and storage overhead (in MB) for evaluation of one byte AES SBox by $\text{SEC}_{\mathsf{OXT}}$ and $\text{SEC}_{\text{ConjFilter}}$ against different TFHE backends.

Scheme	Time taken (in seconds)	Storage (in MB)
TFHE-Nayuki AVX	14.85	24
TFHE-Nayuki Portable	22.862	24
TFHE-Spqlios AVX	4.21	24
TFHE-Spqlios FMA	2.57	24
$\text{SEC}_{\text{ConjFilter}}$ (**This work**)	0.1013	0.449
$\text{SEC}_{\mathsf{OXT}}$ (**This work**)	0.96	0.098

1.2 Technical Overview

Efficient "search" over Encrypted Lookup Tables. The first step towards constructing SEC is to create efficient mechanisms for encrypted table lookups, for which we rely on Searchable Symmetric Encryption (SSE) [16, 25, 45, 46, 48] specifically supporting conjunctive queries [16, 45]. SSE offers a *restricted* yet *efficient* set of capabilities than arbitrary computation frameworks; SSE schemes allow fast and efficient searches over symmetrically encrypted data. Although traditionally, SSE schemes have been used exclusively for searches, we show that SSE data structures are amenable to design modifications such

[3]Gates at the same level, across multiple evaluations of a given circuit topology.

[4]KSH17 Imagenet classification with deep convolutional neural networks

[5]For fair evaluation, we use *unparallelized* version of the SBox [43], which involves 5 XORs per bit in the output, thereby totaling 40 XORs for the entire byte.

that *efficient searches can be used to compute arbitrary Boolean functions on encrypted data*. Any function computation can be modeled into an equivalent logic circuit composed of Boolean variables and universal basic logic gate set {XOR, AND, OR}. Hence, ① creating encrypted lookup tables for these basic gates and ② using efficient *search* capabilities of SSE constructions supporting conjuncitve queries over these encrypted tables, is equivalent to *computing* the logic gates over encrypted inputs.

Lookup Table Design. We explicate the foundational building blocks of the SEC framework with the help of a concrete example - by explaining the end-to-end evaluation of a XOR gate (AND and OR follow suit). Given *encryptions* of two plaintext bits x and y by the client to the server, SEC uses efficient search capabilities of a generic *conjunctive* SSE scheme, CSSE[6], (See Section 2 for details) which returns a single *encrypted* output bit. Upon decryption on the client side, a plaintext bit 0/1 is obtained, that is equal to the actual value of XOR(x, y). A representative *encrypted lookup table* corresponding to XOR used by SEC is illustrated in Table 3.

Table 3: Contents of documents related to the functional evaluation of 2-bit XOR, as well as mapping of document identifiers to their corresponding keywords. Here, Enc_k refers to any generic symmetric encryption scheme with secret key k.

Input bit x	Keyword Map of x	Input bit y	Keyword Map of y	XOR(x, y)	doc_id	Doc. content
0	$\bar{\mathbf{w}}_1$	0	$\bar{\mathbf{w}}_2$	0	D_0	$Enc_k(0)$
0	$\bar{\mathbf{w}}_1$	1	$\mathbf{w}_2$	1	D_1	$Enc_k(1)$
1	$\mathbf{w}_1$	0	$\bar{\mathbf{w}}_2$	1	D_2	$Enc_k(1)$
1	$\mathbf{w}_1$	1	$\mathbf{w}_2$	0	D_3	$Enc_k(0)$

Using generic SSE notation, every plaintext bit (x or y) is *mapped* to some keyword ($\mathbf{w}$). For example, when the plaintext input bit $x = 1$, it is mapped to keyword $\mathbf{w}_1$; conversely, when $x = 0$, we map it to keyword $\bar{\mathbf{w}}_1$. Likewise, input bit y is mapped to $\mathbf{w}_2$ or $\bar{\mathbf{w}}_2$ depending on whether it is 1 or 0 respectively. The actual plaintext output of XOR is encrypted and stored as the document set indexed by identifiers doc_id $(D_i) : i \in \{0, 1, 2, 3\}$. For underlying SSE-based encrypted lookup in SEC, we populate the *inverted search index* representation (as shown in Table 4). The idea is to map a given keyword to the doc_ids which are related to the keyword (as shown in Table 3). The search index (improvised lookup table for XOR) is encrypted and offloaded to the server as done in a typical SSE scheme.

For computing a function, say, XOR(x, y), keywords corresponding to encrypted inputs x and y are chosen and sent as a conjunctive query to the server. We note that *there is exactly one common document between any combination of* "Keyword Map of x" and "Keyword Map of y" (in Table 3) which is returned as a response to the underlying conjunctive SSE query. This design, in conjunction with the "Doc. content" from Table 3, ensures that only the correct evaluation of XOR is returned as a result of querying on SEC. It is crucial to note that at no point in this entire process have we performed an explicit computation of XOR gate (as done in FHE). The entire evaluation is completed by *searching* over encrypted lookup tables (search index) which is extremely fast and efficient.

"Dummy" Keywords/Documents. We add n "special" terms $\mathbf{w}_d^1, \ldots, \mathbf{w}_d^n$ (See Table 4)

[6]Our analysis, proofs, and implementations are based on a CSSE scheme and we will use SSE, conjunctive SSE and CSSE interchangeably for our explanations henceforth.

for a circuit with n gates. "Special" terms are random alphanumeric strings (used as dummy keywords), that are mapped to all documents $(D_0, \ldots, D_3)$ that are present in the search index. It is also observed (in Table 4) that some dummy documents $(D'_0, D'_1 \ldots)$ are added to the search index to ensure that the frequency of "actual" keywords $(\mathbf{w}_1, \mathbf{w}_2, \bar{\mathbf{w}}_1, \bar{\mathbf{w}}_2)$ matches that of the "special" terms $(\mathbf{w}_d^1, \ldots, \mathbf{w}_d^n)$, thereby ensuring a uniform frequency of all keywords in the final encrypted database. These dummy documents comprise encryptions of unique random alphanumeric values. Note that this design choice of adding dummy keywords/documents does not have any impact on the correctness of the functional evaluation. The significance of this design choice is later reflected in reusing the same look-up table for computing the same/different circuits more than once without revealing any correlation for the adversary to infer between the computations.

Table 4: An inverted search index representation of the database (search index) for `XOR` function.

`Keywords`	`doc_id`
$\bar{\mathbf{w}}_1$	D_0, D_1, D'_0, D'_1
$\mathbf{w}_1$	D_2, D_3, D'_2, D'_3
$\bar{\mathbf{w}}_2$	D_0, D_2, D'_4, D'_5
$\mathbf{w}_2$	D_1, D_3, D'_6, D'_7
$\mathbf{w}_d^1$	D_0, D_1, D_2, D_3
$\mathbf{w}_d^2$	D_0, D_1, D_2, D_3
$\vdots$	$\vdots$
$\mathbf{w}_d^n$	D_0, D_1, D_2, D_3

SEC Workflow: Without loss of generality, assume evaluation of `XOR(1,1)` using SEC. This translates to a conjunctive query $q = \mathbf{w}_d^1 \wedge \mathbf{w}_1 \wedge \mathbf{w}_2$ (as inferred from Table 3), where w_d^1 is randomly chosen from the set of n "special" terms for one gate evaluation (i.e. for one conjunctive query). We explicate the sequence of operations that are consequently executed (assuming a black-box conjunctive SSE search), and also draw parallels with conjunctive SSE terminology used in literature below:

1. **Processing "special" Term**: Use the first keyword in q (i.e. $\mathbf{w}_d^1$), generate an address, index into the encrypted lookup table, and retrieve an encrypted list of `doc_id` mapped to $\mathbf{w}_d^1$. For consistency, we denote $\mathtt{sval}_{\mathbf{w},D_j}$ as the retrieved output from this phase for some arbitrary keyword $\mathbf{w}$ and document D_j. In our example, this phase shall return the encrypted list $[\mathtt{sval}_{\mathbf{w}_d^1,D_0}, \mathtt{sval}_{\mathbf{w}_d^1,D_1}, \mathtt{sval}_{\mathbf{w}_d^1,D_2}, \mathtt{sval}_{\mathbf{w}_d^1,D_3}]$ (the encrypted entries corresponding to $\mathbf{w}_d^1$, see Table 4). The choice of the "special" term is dependent on the underlying SSE scheme; generally in OXT or ConjFilter the "special" term is selected according to the least frequent keyword/conjunct in a given conjunctive query for efficiency purposes. In our instantiation of $\text{SEC}_{\mathsf{OXT}}$ and $\text{SEC}_{\text{ConjFilter}}$, it is set to the first keyword/conjunct (since the frequency of all keywords is the same).

2. **Processing "actual" keywords**. In this phase, some auxiliary information dependent on the tuples $(\mathbf{w}_d^1, \mathbf{w}_1)$ and $(\mathbf{w}_d^1, \mathbf{w}_2)$ is used in conjunction with retrieved `sval` set to create a *search token*. For consistency, we denote $\mathtt{Token}_{\mathbf{w}_d^1,\mathbf{w}_i,D_j}$ to denote a search token generated upon combination of $\mathtt{sval}_{\mathbf{w}_d^1,D_j}$ and auxiliary information dependent

on tuple $(\mathbf{w}_d^1, \mathbf{w}_i)$ ($\mathbf{w}_i$ refers to the actual keywords (representing the encrypted input bits) in q, in this example $\mathbf{w}_i \in \{\mathbf{w}_1, \mathbf{w}_2\}$). Upon being indexed into SSE specific data structures, $\texttt{Token}_{\mathbf{w}_d^1, \mathbf{w}_i, D_j}$ returns a binary decision on whether keyword $\mathbf{w}_i$ is present in document D_j. In our example thus, for every $\texttt{sval}_{\mathbf{w}_d^1, D_j}$ in [$\texttt{sval}_{\mathbf{w}_d^1, D_0}$, $\texttt{sval}_{\mathbf{w}_d^1, D_1}$, $\texttt{sval}_{\mathbf{w}_d^1, D_2}$, $\texttt{sval}_{\mathbf{w}_d^1, D_3}$], one token is generated for every keyword of the query other than $\mathbf{w}_d^1$ i.e., $\mathbf{w}_1$ and $\mathbf{w}_2$. We denote them by $\texttt{Token}_{\mathbf{w}_d^1, \mathbf{w}_1, D_j}$ and $\texttt{Token}_{\mathbf{w}_d^1, \mathbf{w}_2, D_j}$ respectively. These tokens are then indexed into SSE specific data structures to return a binary decision on whether both keywords (i.e. $\mathbf{w}_d^1$ and $\mathbf{w}_1/\mathbf{w}_2$) are present in `doc_id` D_j[7].

3. **Query Result**. Any document identifier which is included in the output of step ①, and for which *all* search tokens generated in step ② give a positive binary result, is included in the *result* of the query. Note that, for our example, only document D_3 contains all the keywords $\mathbf{w}_d^1$, $\mathbf{w}_1$, and $\mathbf{w}_2$. Hence, the output of the query $q = (\mathbf{w}_d^1 \wedge \mathbf{w}_1 \wedge \mathbf{w}_2)$, for this example, is $\texttt{sval}_{\mathbf{w}_d^1, D_3}$. This result is returned back to the client, which upon decryption obtains D_3. It follows trivially that since $D_3 = Enc_k(0)$ (see Table 3), query q and associated *search* over the encrypted lookup table has effectively *computed* `XOR(1,1)`.

Functional Correctness. Concretely, we design the encrypted database (search index) in a way such that for arbitrary encrypted one-bit inputs x and y (their corresponding keyword mappings), *exactly* one encrypted document is returned, which upon decryption reveals a single bit $b = \texttt{XOR}(x, y)$, (thereby allowing SEC to correctly compute the function `XOR`).

Reusability of Lookup Tables for Multiple Evaluations. SEC enables reusability of the same lookup table for multiple evaluations of the same/different circuits while ensuring data privacy guarantees. The client chooses a "special" term for evaluating a particular gate, hence for evaluating the same/different gate more than once, the client can randomly choose a unique "special" term for each gate evaluation. Since only the "special" term in a given conjunctive query is used to index into the encrypted look-up table in the memory and retrieve respective encrypted documents, choosing random "special" terms for each query (equivalent to each gate evaluation at same/different depth of the circuit) obfuscates the *access pattern* of the memory location accessed by the "special" terms. Thus even if the client computes the same function, say $\texttt{XOR}(x, y)$, twice on the same encrypted input, the server (adversary) cannot infer any correlation between the two computations. This ensures the reusability of the same encrypted look-up table for multiple circuit evaluations without any significant information leakage to the semi-honest server. Such reusability amortizes client's communication overhead, since it sets the "special terms" linear in the circuit-size once and reuses them for multiple evaluations while preventing exponential growth in storage/communication. The client does not require prior knowledge of circuit composition. Concretely, for a publicly-known circuit with n gates, the search-index has n "special terms". Reusability is then derived from client-controlled "permutation" of assigning unique "special term" to isomorphic gates across multiple runs, ensuring unique access-pattern across multiple gate evaluations. Concretely, the upper bound on reusability is the possible derangements, given by $n! - \Sigma_{i=0}^{n} \frac{(-1)^{i+1}}{i!}$ ($= O(n!)$ asymptotically). "special

[7]This flow of token generation is specific to underlying SSE algorithm.

terms" are permuted using a Psuedo-Random Permutation primitive, for which the client maintains $O(1)$ state. More details follow in Section 4.

Composable Function Evaluation. While SEC harbors the capability of efficiently evaluating a binary Boolean gate, extending this to an n-size circuit is non-trivial. One way to use SEC in order to evaluate compositions of form $f_k(f_i(\cdot,\cdot), f_j(\cdot,\cdot))$ (where f_i, f_j, $f_k \in \{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$) is to first evaluate $f_i(\cdot,\cdot), f_j(\cdot,\cdot)$, send the result to the client who constructs the query for the *outer* function, and then execute the outer function query. However, this incurs extra communication rounds that scale linearly with the circuit size.

To circumvent this issue, we augment SEC to perform query construction for the outer function on the (semi-honest) server itself. That is, instead of the need to decrypt the result (i.e. $\texttt{sval}_{\mathbf{w}_d^i, D_i}$ for $f_i(\cdot)$ and $\texttt{sval}_{\mathbf{w}_d^j, D_j}$ for $f_j(\cdot)$) of the inner function to recover D_i and D_j, we use mechanisms to directly map $\texttt{sval}_{\mathbf{w}_d^i, D_i}$ and $\texttt{sval}_{\mathbf{w}_d^j, D_j}$ to construct relevant query used by the outer function $f_k(\cdot)$. To do so, we extend Table 3 to allow compositions of `XOR` (as shown in Table 5). We note that compositions of `AND`/`OR`, as well as intermixing of `XOR`/`AND`/`OR` follow suit.

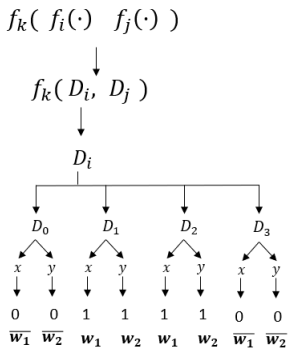

Figure 1: Construction of query for $f_k(\cdot)$ from the output of $f_i(\cdot)$ and $f_j(\cdot)$.

There are four possible outcomes for each of the inner `XOR` evaluation (denoted by plaintext document identifiers D_0, D_1, D_2, D_3), and each outcome can either behave as the first input (corresponding to bit x) or the second input (corresponding to bit y) for the outer function. For example, should the inner function output D_0 (corresponding to the evaluation of `XOR`(0, 0); see Table 3), then for the outer function, either $x = 0$ or $y = 0$ depending on specific wiring topology of these gates (See Figure 1).

Therefore, according to Table 5, either keyword $\bar{\mathbf{w}}_1$ or $\bar{\mathbf{w}}_2$ shall be involved in the outer function's query. This mapping hence allows query construction of the outer function evaluation on the server side itself. For further compositions, (like the output f_k being used in some outer computation), the same table is *reused*; this prevents any exponential blowup in storage. Concretely, a single mapping of form Table 5 is sufficient for evaluating composed functions of arbitrary depth. Table 5 appended with "special" keywords and dummy documents entail the final search index for the `XOR` function. Similar construction follow for the search index of `AND`/`OR` gates as well.

Illustrative Example 1. Consider a client who wishes to evaluate the circuit in Figure 2, using SEC without involving communication rounds with the client after inner function

Table 5: Mapping between the result of an inner function (say $f_i(\cdot)$ here), to keywords (input bits) used in querying the outer function ($f_k(\cdot)$). Mapping of doc_id is directly extrapolated from Table 4. For ease of exposition, we omit "special" terms and dummy documents from this illustration; they are added to the search index similarly as explained previously.

Output of $f_i(\cdot)$	Input bits of $f_k(\cdot)$	Keyword	Mapped doc_id
D_0	$x = 0$	$\mathbf{w}_1$	D_0, D_1
D_1	$x = 1$	$\mathbf{w}_1$	D_2, D_3
D_2	$x = 1$	$\mathbf{w}_1$	D_2, D_3
D_3	$x = 0$	$\mathbf{w}_1$	D_0, D_1
D_0	$y = 0$	$\mathbf{w}_2$	D_0, D_2
D_1	$y = 1$	$\mathbf{w}_2$	D_1, D_3
D_2	$y = 1$	$\mathbf{w}_2$	D_1, D_3
D_3	$y = 0$	$\mathbf{w}_2$	D_0, D_2

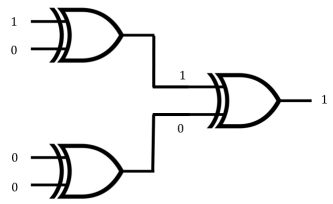

Figure 2: Evaluate: $f_{\texttt{XOR}}(f_{\texttt{XOR}}(1,0), f_{\texttt{XOR}}(0,0))$

evaluations and using the *same* lookup table (Table 5; that is encrypted and offloaded to the server once at the beginning). The encrypted lookup tables for all three functions $\texttt{f} = \{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$ are generated once and offloaded to the server. We emphasize here that the size of the lookup table is agnostic of the circuit being evaluated. For ensuring a single communication round between the client and server, the query construction for outer XOR needs to happen solely on the server side. For now, we assume $\texttt{ConstructQuery}(\mathbf{w}_d^k, \texttt{sval}_{\mathbf{w}_d^l, D_j}, b)$ to abstract the following mapping: Given that $\texttt{sval}_{\mathbf{w}_d^l, D_j}$ is output by the inner function evaluation, ConstructQuery returns the search tokens of the form $\texttt{Token}_{\mathbf{w}_d^k, \mathbf{w}_i, D_j}$ depending on the mapping in Table 5. Bit b denotes whether $\texttt{sval}_{\mathbf{w}_d^k, D_j}$ is first or second input to the outer function. We defer details of the exact operation of ConstructQuery to Section 3.2. We also note that since *three* evaluations of XOR occur, we use a *distinct* "special" term for each. We now explain how the evaluation proceeds.

SEC first constructs a query $q_1 = (\mathbf{w}_d^1 \wedge \mathbf{w}_1 \wedge \bar{\mathbf{w}}_2)$ for $f_{\texttt{XOR}}(1,0)$, resulting in $\texttt{sval}_{\mathbf{w}_d^1, D_2}$ ($\mathbf{w}_d^1$ is a randomly sampled "special" term; D_2 occurs in both $\mathbf{w}_1 \wedge \bar{\mathbf{w}}_2$, see Table 5). Likewise, SEC constructs a query $q_2 = (\mathbf{w}_d^2 \wedge \bar{\mathbf{w}}_1 \wedge \bar{\mathbf{w}}_2)$ for $f_{\texttt{XOR}}(0,0)$, resulting in $\texttt{sval}_{\mathbf{w}_d^2, D_0}$ ($\mathbf{w}_d^2$ is a randomly sampled "special" term; D_0 occurs in both $\bar{\mathbf{w}}_1 \wedge \bar{\mathbf{w}}_2$, see Table 5). Note that the result of $f_{\texttt{XOR}}(1,0)$ drives the first input of outer XOR. Thus, SEC invokes $\texttt{ConstructQuery}(\mathbf{w}_d^3, \texttt{sval}_{\mathbf{w}_d^1, D_2}, 1)$ to obtain $\texttt{Token}_{\mathbf{w}_d^3, \mathbf{w}_1, D_2}$ and $\texttt{Token}_{\mathbf{w}_d^3, \mathbf{w}_1, D_3}$ ($\mathbf{w}_d^3$ is a randomly sampled "special" term; $\mathbf{w}_1$ is chosen as the keyword for the first input of outer XOR because the output of $f_{\texttt{XOR}}(1,0)$ gives D_2, and from the corresponding row entry 3 in Table 5 we get $\mathbf{w}_1$; (D_2, D_3) are documents corresponding to $\mathbf{w}_1$; 1 in ConstructQuery() denotes this is the first input of the gate). Likewise, since $f_{\texttt{XOR}}(0,0)$ drives the second input of outer XOR, SEC invokes $\texttt{ConstructQuery}(\mathbf{w}_d^3, \texttt{sval}_{\mathbf{w}_d^2, D_0}, 0)$ to obtain $\texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_0}$ and $\texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_2}$ ($\mathbf{w}_d^3$ is a randomly sampled "special" term; $\bar{\mathbf{w}}_2$ is chosen as the keyword

for the second input of outer XOR because the output of $f_{\texttt{XOR}}(0,0)$ gives D_0, and from the corresponding row entry 5 in Table 5 we get $\bar{\mathbf{w}}_2$; (D_0, D_2) are documents corresponding to $\bar{\mathbf{w}}_2$; 0 in ConstructQuery() denotes this is the first second of the gate). Overall, for the outer XOR, the constructed query is $q_3 = (\mathbf{w}_d^3 \wedge \mathbf{w}_1 \wedge \bar{\mathbf{w}}_2)$. It is straightforward to see that the result of the functional composition, by design of Table 5 shall be $\texttt{sval}_{\mathbf{w}_d^3, D_2}$ (D_2 is the common document between all four Tokens generated by ConstructQuery()). The server sends this *final* result to the client. Upon decryption, the client obtains the output plaintext bit 1 ($D_2 = Enc_k(1)$), which is the correct evaluation of the circuit. We emphasize that since the "special" term changes across all gate evaluations (due to the random selection of "special" terms for each gate by the client), from the server's perspective, it can not correlate the underlying computations.

Illustrative Example 2: Reusability. We demonstrate another scenario in this example, where the client wishes to consecutively evaluate the circuit in Figure 2 twice. Assume that for Evaluation 1, SEC constructs a query similar to the previous example. For Evaluation 2, the client assigns a different set of "special" terms and SEC constructs corresponding query as - $q_1 = (\mathbf{w}_d^3 \wedge \mathbf{w}_1 \wedge \bar{\mathbf{w}}_2)$ for $f_{\texttt{XOR}}(1,0)$, resulting in $\texttt{sval}_{\mathbf{w}_d^3, D_2}$. Likewise, a query $q_2 = (\mathbf{w}_d^1 \wedge \bar{\mathbf{w}}_1 \wedge \bar{\mathbf{w}}_2)$ for $f_{\texttt{XOR}}(0,0)$, resulting in $\texttt{sval}_{\mathbf{w}_d^1, D_0}$. As the result of $f_{\texttt{XOR}}(1,0)$ drives the first input of outer XOR, SEC invokes $\texttt{ConstructQuery}(\mathbf{w}_d^2, \texttt{sval}_{\mathbf{w}_d^3, D_1}, 1)$ to obtain $\texttt{Token}_{\mathbf{w}_d^2, \mathbf{w}_1, D_2}$ and $\texttt{Token}_{\mathbf{w}_d^2, \mathbf{w}_1, D_3}$ (corresponding to row entry 3 in Table 5). Likewise, since $f_{\texttt{XOR}}(0,0)$ drives the second input of outer XOR, SEC invokes $\texttt{ConstructQuery}(\mathbf{w}_d^2, \texttt{sval}_{\mathbf{w}_d^1, D_0}, 0)$ to obtain $\texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_0}$ and $\texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_2}$ (corresponding to row 5 in Table 5). Overall, for the outer XOR, the constructed query is $q_3 = (\mathbf{w}_d^2 \wedge \mathbf{w}_1 \wedge \bar{\mathbf{w}}_2)$ and the final result is $\texttt{sval}_{\mathbf{w}_d^2, D_2}$. Note that the "special" terms used for q_1, q_2, q_3 are different hence, different sval and Tokens are generated for the same circuit in Evaluation 2. This obfuscates memory access pattern. However the result of the final evaluation will be the same (this ensures functional correctness).

It is to be noted that Table 5 is an abstract overview of the lookup table representation, hence for ease of explanation, we show that the entries related to a search tokens ($\{\texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_0}$ and $\texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_2}\}$ for Evaluation 1 and $\{\texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_0}$ and $\texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_2}\}$ for Evaluation 2), correspond to the same row of the lookup table (row 5 in Table 5) for both evaluations. In practice, however, every entry corresponding to a "special" term is stored separately, hence although the content of the encrypted document is the same, the locations in memory will vary. This is crucial to prevent the server from correlating between two (similar/different) circuit evaluations, thereby rendering the reusability of the same lookup table for multiple evaluations secure, in terms of data privacy.

From the server's perspective (in Figure 2), for gate 1, the server observes access patterns related to $\{\ \texttt{sval}_{\mathbf{w}_d^1, D_2}\ \}$ in Evaluation 1 and $\{\ \texttt{sval}_{\mathbf{w}_d^3, D_2}\ \}$ in Evaluation 2. Similarly, for gate 2, access patterns observed are $\{\ \texttt{sval}_{\mathbf{w}_d^2, D_0}\ \}$ in Evaluation 1 and $\{\ \texttt{sval}_{\mathbf{w}_d^1, D_0}\ \}$ in Evaluation 2. Likewise, for gate 3, the server observes access patterns for $\{\ \texttt{Token}_{\mathbf{w}_d^3, \mathbf{w}_1, D_2}$, $\texttt{Token}_{\mathbf{w}_d^3, \mathbf{w}_1, D_3}, \texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_0}, \texttt{Token}_{\mathbf{w}_d^3, \bar{\mathbf{w}}_2, D_2}, \texttt{sval}_{\mathbf{w}_d^3, D_2}\ \}$ in Evaluation 1 and $\{\ \texttt{Token}_{\mathbf{w}_d^2, \mathbf{w}_1, D_2}$, $\texttt{Token}_{\mathbf{w}_d^2, \mathbf{w}_1, D_3}, \texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_0}, \texttt{Token}_{\mathbf{w}_d^2, \bar{\mathbf{w}}_2, D_2}, \texttt{sval}_{\mathbf{w}_d^2, D_2}\ \}$ in Evaluation 2. Briefly, through client-controlled permutation, assignment of "special" terms to *non-isomorphic* gates[8] prevents the server from correlating between two (similar/different) circuit evaluations, thereby

[8]If in Evaluation 2, the client had assigned $\mathbf{w}_d^2$ to gate 2, then this assignment is isomorphic to that in

making the reusability of the same lookup table for multiple evaluations secure, in terms of data privacy.

Note. It is important to note that, for ease of exposition, we exemplified the working mechanism of SEC for solving a circuit with `XOR` *gates only. An exactly similar execution methodology is used for evaluating any arbitrary circuit f with any combination of gates, $f \in \{\texttt{XOR}, \texttt{AND}, \texttt{OR}\}$, since both* `sval` *and* `Token` *are function agnostic.*

Communication Complexity. The bulk of communication overhead is a *one-time* setup[9] where the client generates and offloads data structures related to SEC to the server. For a circuit $\mathcal{C}$, consisting of n gates $\{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_n\}$, we assign a unique "special" term sequence to each circuit and some *dummy* documents (to maintain a uniform frequency of all keywords in the encrypted database). This prevents any two gates in $\mathcal{C}$ from having the same "special" terms. The space complexity of SEC thus scales linearly with $\mathcal{O}(\mid \mathcal{C} \mid)$.[10]

Security in Reusability. We stress that the same look-up tables can be used for evaluation of multiple circuits. We do this through client-controlled *permutation* of the pool of "special" terms available in SEC's encrypted look-up tables. To exemplify, reconsider the example circuit $f_{\texttt{XOR}}(f_{\texttt{XOR}}(1,0), f_{\texttt{XOR}}(0,0))$, and assume this needs to be executed *twice*. For first evaluation, let's say the client assigns permutation of "special" terms $\mathbf{w}_d^1, \mathbf{w}_d^2, \mathbf{w}_d^3$ to the three gates respectively. For the second evaluation, let's say, the client chooses a *new* permutation: $\mathbf{w}_d^3, \mathbf{w}_d^1, \mathbf{w}_d^2$. This enables *reusability* of SEC's look-up tables across different circuits while ensuring ① no two gates in the *same* circuit share a common "special" term, and ② two isomorphic gates in *different* circuits also do not share corresponding "special" terms; thereby preventing the server from inferring any correlation between two computations. We elaborately discuss and analyze the leakage profile of SEC in Section 4.

1.3 Related Work

Garbled Circuits. It was first introduced by Yao in his groundbreaking work on secure function evaluation [51]. Due to their versatility, various garbling techniques (for arithmetic/Boolean circuits) [5, 36, 42, 51] have been developed over the years. GCs typically provide privacy for the input/output bits and entire circuit that is being computed. However, such constructions have an inherent disadvantage of not being reusable that makes it unnameable for practical deployment. As such, there have been attempts to construct reusable Garbled Circuits [36], but the underlying primitive used is Functional Encryption (using FHE as a black box) which is computationally expensive.

Fully Homomorphic Encryption (FHE). In practice, computing over encrypted outsourced databases has used sophisticated and highly structured primitives, such as Fully Homomorphic Encryption (FHE) [2, 3, 21, 28, 30, 31, 32, 34] which views generic computations as either arithmetic or Boolean circuits (hence are typically expensive), and provides

Evaluation 1. Consequently, the server's leakage profile for gate 2 is $\{ \texttt{sval}_{\mathbf{w}_d^2, D_0} \}$ for both evaluations, hence leaking that both evaluations have the *same type of gate*. Note that, however, neither the gate description (i.e. whether it is `AND`/`OR`/`XOR`) nor the exact plaintext bit in $\{ \texttt{sval}_{\mathbf{w}_d^2, D_0} \}$ is leaked, since this information is encrypted, thereby still protecting data privacy.

[9] Across evaluations of multiple circuits.

[10] Keywords specific to functional evaluation are $\mathcal{O}(1)$; refer Table 4 and Table 5.

an all-or-nothing flavor of security. FHE has recently gained traction in the cryptographic literature for privacy-preserving computation with rich functionalities along with the *ideal* notion of privacy. However, as each primitive function evaluation is realized by explicit circuit evaluation followed by an expensive *bootstrapping* operation. FHE has prohibitively high computation costs and storage overheads.

Searchable Symmetric Encryption (SSE). SSE schemes [16, 19, 25, 48] provision users with *search* capabilities over symmetrically encrypted data. There exist today efficient SSE schemes that support conjunctive (and more general Boolean) queries [15, 16, 46]. An *ideal* SSE construction using Oblivious RAM (ORAM) promises oblivious memory access patterns (and hence *no* leakage) but is hard to actualize in hardware, is closed-source, and has not been tested against scaled databases [18]. Modern SSE schemes thus trade-off security for efficiency. These schemes allow the server to learn "some" information during query execution, detailed by their *leakage* profile. While SSE schemes are extremely fast and highly scalable with arbitrarily large real-world datasets, their restricted functionality (to only *search*) renders them inapt for practical deployment in an encrypted computation framework. In this work, we leverage the efficient look-up capabilities of SSE while extending the limited functionality of existing SSE schemes to supporting encrypted computations of arbitrary Boolean functions.

1.4 How SEC differs from GC and HE

In its essence, SEC guarantees privacy of input and output bits to and from a Boolean function. The rationale behind the construction of SEC is to provide an efficient and accelerated alternative to the existing state-of-the-art general purpose encrypted computing frameworks while leaking some non-trivial information that is proven to be benign. We would like to emphasize that the security guarantee of SEC is based on the IND-CPA security of a symmetric key encryption scheme. It is important to mention that the function evaluated by SEC is assumed to publicly known. We provide an elaborate comparison of SEC wrt. the most closely related encrypted computation frameworks in literature.

Garbled Circuits. We emphasize that with a single setup of $O(n)$ special-terms and its client-controlled permutation, SEC ensures multiple executions of the same circuit without leaking any equality-correlation to the server. This is the core difference between SEC and GC. While GC uses two inputs to evaluate a single binary-gate by indexing into its translation tables, SEC uses three inputs[11] to evaluate the same gate. The extra input neither encodes the actual bit nor participates in the computation, but just randomizes memory accesses for the same gate across multiple executions. This allows up to $O(n!)$ re-executions of the same circuit with a single setup, while GC can tolerate just one execution. **"Illustrative Example 2"** demonstrates how SEC can execute the same circuit (with same inputs) twice without leaking correlations (unlike GC with a single setup). Footnote 8 gives an example where an *incorrect* permutation of "special" terms leaks to leakage in SEC. It is the client's responsibility to ensure the same "special" terms are not assigned to isomorphic gates.

SEC also differs from GCs that use structured encryption underneath. For instance, Kamara

[11] Two inputs corresponding to gate input wires, and an additional "special" term.

et. al. proposed a garbled circuit construction from a structured encryption scheme in [42], which seems closely related to SEC in terms of its design choice and functionality. We emphasize, however, SEC is inherently distinct from the construction proposed in [42], as it relies on the efficient encrypted look-up operation of a conjunctive SSE scheme to evaluate arbitrary Boolean functions. Essentially [42] reduces the problem of designing special-purpose garbled circuits to the problem of designing structured encryption schemes. Whereas, SEC reduces the problem of designing an arbitrary Boolean function evaluation framework on encrypted input bits to the problem of designing a conjunctive SSE scheme. As such SEC ensures data-privacy guarantees, while assuming the circuit is publicly known. SEC entails a single round of setup for $O(n!)$ circuit evaluations (n-size circuit), without leaking any correlation between the circuit evaluations[12]. On the contrary [42] requires the setup phase to run for every evaluation, i.e. for an n-sized circuit, a garbled circuit of size n needs to be setup every-time for evaluating the circuit without leaking any information.

Fully Homomorphic Encryption. In this work, we provide an encrypted computation framework that is more efficient than FHE in terms of computation and storage complexity, while providing similar security guarantees of data-privacy. The construction of SEC fundamentally differs from FHE schemes because SEC bypasses explicit circuit evaluation, hence the consequent need for bootstrapping operation, that essentially contributes to FHE's computational bottleneck.

We use TFHE as the concrete implementation of FHE for comparison with SEC. Functional bootstrapping in TFHE enables the evaluation of homomorphic functions, which can be used to perform encrypted table lookups. The lookup is performed using a homomorphic truth table representation and lookup table evaluation. It enables arbitrary function computation over encrypted data. Lookups are performed using blind rotation and GGSW ciphertext [34] manipulations. The process is computationally expensive because bootstrapping is costly. The fundamental difference of TFHE with SEC is that the later relies on the efficient encrypted lookup operations of a conjunctive SSE scheme. The lookup operation in SSE is simpler and extremely efficient. On the other hand, while TFHE can support arbitrary number of evaluations with a single setup, SEC can manage bounded evaluations. We argue however, that such a bound is asymptotically large for practical deployments: for a single setup of a circuit of size n, SEC bounds the number of re-evaluations it can perform by $O(n!)$.

With respect of comparison of security, SEC provides data-privacy guarantee by encrypting the input/output bits to/from a gate, using an IND-CPA secure symmetric-key encryption scheme. The incorporation of an SSE scheme as a black-box makes it straightforward to see that the leakages that exist in state-of-the-art SSE schemes will affect the overall security of SEC. However we prove that the leakages incurred by the underlying SSE does not transitively reflect in the leakage profile of SEC due to appropriate measures and design choices made. We claim that the leakages from the underlying SSE are handled in SEC, which renders them inconsequential and preserves its data-privacy guarantees[13]. Hence,

[12]Note, the privacy of input bits to the circuit and output bit after evaluation is guaranteed by the IND-CPA symmetric-key encryption scheme with which the input and output bits are encrypted

[13]Informally, SEC uses three inputs to evaluate a binary gate, and ties SSE related leakages to the "special" term, which does not encode the real gate inputs. In other words, although SEC leaks, its leakages are tied to the "special" terms and not the real gate inputs. By randomizing the permutation of the "special" terms across different evaluations, SEC prevents any exploitable leakage.

although unlike FHE, SEC is prone to underlying SSE leakages, we underscore that such leakages are benign and has no adverse affect on the overall security of the scheme. We provide elaborate details on the security analysis of SEC in Section 4.

Leveled Homomorphic Encryption. As seen from Table 1, Leveled Homomorphic Encryption (LHE) [12, 14, 37] allows evaluating circuits of bounded size without requiring bootstrapping, and provides a balance between efficiency and expressiveness. However, LHE can only evaluate circuits up to a fixed depth (e.g., $10 \sim 20$ [39], $10 \sim 50$ [41] multiplicative levels), causing a decryption failure due to excessive noise if computations exceed this depth. Whereas SEC supports $O(n!)$ evaluations of a circuit of size n, where n is a client-controlled parameter and not specific to the design of SEC. This is an important feature as it accelerates the scalability of SEC and renders it suitable for practical real-world deployment.

2 Preliminaries and Background

We present preliminary concepts and background in this section. Table 6 lists basic notations used in this paper. Any other notation used is defined in-place within the context of the main text.

Table 6: Summary of notations

λ	security parameter
$\mathsf{id}/\mathsf{doc\_id}$	document identifier
$\mathbf{w}$	a keyword
$\mathcal{W}$	dictionary of keywords $\mathcal{W} = \{\mathsf{w}_1, \ldots, \mathsf{w}_N\}$
$\mathbf{DB}$	database $(\mathsf{id}_j, \mathbf{w}_i)_{i=1\ j=1}^{N\ \ \lvert\mathbf{DB}(\mathbf{w})\rvert} \in \mathbf{DB}$
$\mathbf{DB}(\mathbf{w})$	all documents containing $\mathbf{w}$
n	max. number of keywords per conjunctive query.
$x \xleftarrow{\$} \chi$	uniformly sampling x from χ
$x = \mathcal{A}$	x is output of a deterministic algorithm
$x \leftarrow \mathcal{A}'$	x is output of a randomized algorithm

2.1 Conjunctive SSE: Syntax and Security Model

A *Conjunctive Searchable Symmetric Encryption* scheme (CSSE) provisions the client with *conjunctive* search capability (i.e. search Boolean queries of the form $\mathbf{w}_1 \wedge \mathbf{w}_2 \wedge \ldots \wedge \mathbf{w}_\mathsf{n}$) over an encrypted database. A CSSE scheme can be formally defined as an ensemble of four polynomial-time algorithms {KeyGen, Encrypt, GenToken, Search} [14] such that:

• KeyGen(λ) is a probabilistic algorithm that takes the security parameter λ as input. The output of this algorithm is the client's secret key sk.
• Encrypt($\mathsf{sk}, \mathbf{DB}$) is a probabilistic algorithm that takes as input the client secret key sk

[14] Our syntax for a conjunctive SSE is different from the syntax of traditional conjunctive SSE scheme {Setup, GenToken, Search}, but the underlying functionality is exactly similar

and a plain database **DB**. The output is an encrypted database **EDB**.
• GenToken($\mathsf{sk}, q = \mathbf{w}_1 \wedge \ldots \wedge \mathbf{w}_\mathsf{n}$) is a deterministic algorithm executed by the client that takes as input secret key sk and a conjunctive query $q = \mathbf{w}_1 \wedge \ldots \wedge \mathbf{w}_\mathsf{n}$. It generates *search tokens* (st_q) corresponding to the conjunctive query q as the output.
• Search($\mathbf{EDB}, \mathsf{st}_q$) is a deterministic algorithm executed by the *server* that takes as input **EDB** and the search token st_q corresponding to a conjunctive query q. It returns the encrypted document identifiers $\mathbf{DB}(\mathsf{st}_q)$ corresponding to the conjunction $q = \mathbf{w}_1 \wedge \ldots \wedge \mathbf{w}_\mathsf{n}$ as output.

Correctness. A CSSE scheme is said to be correct if for an **EDB** generated from a **DB** using Encrypt, for a search token st_q generated by GenToken from any conjunctive Boolean query q formed over the keywords $\mathbf{w}_i$ in $\mathcal{W}$, the Search routine returns a set of ids as result which is the same as $\mathbf{DB}(q)$ with high probability.

Security. The security of a CSSE scheme is parameterized by a *leakage function* $\mathcal{L}$, which encapsulates the information that can be learnt (potentially by an adversary) from the encrypted database and query transcripts. Formally, the security notion 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 CSSE = {KeyGen, Encrypt, GenToken, Search} be a CSSE 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) $\mathbf{Real}_{\mathcal{A}}^{\text{CSSE}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\text{SIM}}^{\text{CSSE}}(\lambda)$, as in Algorithm 1 and Algorithm 2, respectively (see Appendix 2.2). We say that CSSE is $\mathcal{L}$-semantically-secure against adaptive attacks if for all adversaries $\mathcal{A}$ there exists an algorithm SIM such that

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

In these experiments, the leakage function for CSSE is expressed as

$$\mathcal{L}_{\text{CSSE}} = (\mathcal{L}_{\text{CSSE}}^{\text{Encrypt}}, \mathcal{L}_{\text{CSSE}}^{\text{GenToken}}, \mathcal{L}_{\text{CSSE}}^{\text{Search}}),$$

where $\mathcal{L}_{\text{CSSE}}^{\text{Encrypt}}$ encapsulates the leakage to an adversarial server during the Encrypt phase, $\mathcal{L}_{\text{CSSE}}^{\text{GenToken}}$ encapsulates the leakage to an adversarial server during the GenToken phase, and $\mathcal{L}_{\text{CSSE}}^{\text{Search}}$ encapsulates the leakage to an adversarial server during each execution of the Search protocol.

2.2 Adaptive Security of CSSE

We present the **Real** and **Ideal** experiments for the security analysis of a conjunctive SSE scheme CSSE in this Appendix. In these experiments, the leakage function for CSSE is expressed as

$$\mathcal{L}_{\text{CSSE}} = (\mathcal{L}_{\text{CSSE}}^{\text{Encrypt}}, \mathcal{L}_{\text{CSSE}}^{\text{GenToken}}, \mathcal{L}_{\text{CSSE}}^{\text{Search}}),$$

where $\mathcal{L}_{\text{CSSE}}^{\text{Encrypt}}$ encapsulates the leakage to an adversarial server during the Encrypt phase, $\mathcal{L}_{\text{CSSE}}^{\text{GenToken}}$ encapsulates the leakage to an adversarial server during the GenToken phase, and $\mathcal{L}_{\text{CSSE}}^{\text{Search}}$ encapsulates the leakage to an adversarial server during each execution of the Search protocol.

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

function $\mathbf{Real}^{\text{CSSE}}_{\mathcal{A}}(\lambda)$
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{sk}, \mathsf{s}_0, \mathbf{EDB}_0) \leftarrow \text{CSSE.}\textsc{Encrypt}(\lambda, N)$
  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 $(\mathsf{s}_k, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow$ CSSE.$\textsc{Search}(\mathsf{sk}, \mathsf{s}_{k-1}, q_k; \mathbf{EDB}_{k-1})$
    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 $\mathbf{Ideal}^{\text{CSSE}}_{\mathcal{A},\text{SIM}}(\lambda, Q, \mathcal{L})$

function $\mathbf{Ideal}^{\text{CSSE}}_{\mathcal{A},\text{SIM}}(\lambda, Q, \mathcal{L})$
  Parse the leakage function $\mathcal{L}$ as: $\mathcal{L} = \left(\mathcal{L}^{\textsc{Encrypt}}_{\text{CSSE}}, \mathcal{L}^{\textsc{Search}}_{\text{CSSE}}\right)$.
  $(\mathsf{s}_{\text{SIM}}, \mathbf{EDB}_0) \leftarrow \text{SIM}_{\textsc{Setup}}(\mathcal{L}^{\textsc{Encrypt}}(\lambda, N))$
  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 $(\mathsf{s}_{\text{SIM}}, \mathbf{EDB}_k, \tau_k) \leftarrow \text{SIM}_{\textsc{Search}}(\mathsf{s}_{\text{SIM}}, \mathcal{L}^{\textsc{Search}}_{\text{CSSE}}(q_k); \mathbf{EDB}_{k-1})$
    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

2.3 Oblivious Cross-Tag Protocol (OXT): Overview

We provide a brief understanding of the the technical details of the OXT protocol. OXT [17] relies on specially structured pseudo-random functions that can be incorporated using discrete-log hard groups. The idea is that the client encrypts the data (using symmetric-key encryption) and offloads it to the server (*honest-but-curious*). The client queries the server with a conjunction of keywords to which it returns a set of encrypted pointers that point to documents containing all the client's queried keywords. The client decrypts these pointers locally to obtain the required documents matching the conjunctive query. The server does not have the ability to perform decryption nor can it learn any information about the queried keywords. The entire protocol constitutes one-round of interaction/communication between the client and the server. This is delegated by an *oblivious shared computation of cross-tags* between client and server. In order to achieve minimum communication complexity while ensuring privacy of the client's queried keywords and corresponding documents, OXT incorporates an *oblivious cross-tag* generation process. The cross-tag is computed using blinded exponentiation in prime order cyclic groups, analogous to *Diffie-Hellman based oblivious PRF*. It pre-computes the blinding part of the oblivious computation and stores them in encrypted form at the server. During search, the client sends tokens to the server using which it unlocks these pre-computed values and computes the cross-tag *obliviously* using

which it matches the documents corresponding to the queried keywords.

The core technical idea of OXT is the process of oblivious cross-tag generation, that relies on public-key computations in a discrete-log hard group. The hardness assumption is based on the fact that *DDH assumption* is conjectured to hold in a prime order subgroup of Z_p^* (p is a large prime). The inherent reliance of OXT on discrete-log hard groups renders it vulnerable to quantum attacks and can be effectively broken by a scalable quantum computer. Therefore, although OXT provides security against efficient adaptive adversaries in classical setting, it is not secure in the post-quantum setting. The motivation of our work is hence, to develop a post-quantum secure construction of OXT that preserves its asymptotic search and communication complexity while ensuring its scalability with arbitrary large datasets.

The Cross-Tag. The most fundamental component of OXT which is incorporated to check for the presence of a keyword in a particular document without leaking any extra information to the server, is the *cross-tag* (denoted as xtag in the paper). The fundamental building block that makes OXT one of the most efficient, highly scalable and secure conjunctive SSE scheme is an *oblivious computation of the* xtag *at the server*, which is incorporated using a DH-based oblivious PRF type computation. The idea is, for every keyword and every document in which it is present, the client pre-computes a xtag (an element in the prime-order subgroup of Z_p^*) and stores it in a data-structure called XSet which is offloaded to the server.

$$\mathsf{xtag} \leftarrow g^{F_p(K_X,\mathsf{w})\cdot F_p(K_I,\mathsf{id})}$$

where, g is the generator of the subgroup of Z_p^* and F_p is a PRF that takes as input a keyword or a document identifier and outputs an element in Z_p^*.

Along with the queried keywords, the server receives some tokens (xtoken) from the client (also an element in the prime-order subgroup of Z_p^*) during any conjunctive search. The beauty of OXT lies in the fact that the whole process of xtag computation by the server takes place obliviously without revealing the keyword-document pair for which the xtag is being computed. This is done by raising the xtoken to a blinded value y (an element in Z_p^*) which is pre-computed by the client and stored at the server.

3 Symmetric Encrypted Computation

We introduce Symmetric Encrypted Computation (SEC) as an efficient framework for fast outsourced privacy-preserved Boolean circuit evaluation in the symmetric-key setting. SEC supports circuit evaluation of arbitrary size via encrypted look-up and a single round of communication between the client and the remote server. We begin by presenting the high-level syntax of SEC in this section before delving into the elaborate technical details of the framework subsequently.

3.1 Syntax of SEC

We briefly explain a general syntax of our proposed primitive here. It uses a static conjunctive SSE construction CSSE as a black-box (refer to Section 2.1 for details on syntax and security definitions). We assume a single (honest) client and a (semi-honest) server in SEC. SEC is abstracted as a tuple of four polynomial-time algorithms {KeyGen, Encrypt, Evaluate, Decrypt}, as defined below:

• KeyGen(λ): A probabilistic algorithm executed by the client that takes as input the security parameter λ. It outputs a client secret key sk and a public parameter pp.

• Encrypt($\mathsf{ke}, x_1, \ldots, x_p$): A probabilistic algorithm executed by the client. It takes the client's secret key ke and a p-bit input $\{x_1, \ldots, x_p\}$. The output is a ciphertext $c = \{c_1, \ldots, c_p\}$.

• Evaluate($f_{\mathsf{desc}}, c_1, \ldots, c_p, \mathsf{pp}$): A deterministic algorithm executed by the server that takes as input a description of a circuit f_{desc} that is to be evaluated, the encrypted input bits $c = \{c_1, \ldots, c_p\}$ and the public parameter pp. The server returns the encrypted evaluation of the circuit eval to the client.

• Decrypt($\mathsf{ke}, \mathsf{eval}$): A determinisitic algorithm executed by the client with its secret-key ke and an encrypted evaluation eval as input, which outputs the decrypted result.

Correctness. SEC is said to be functionally correct if for security parameter λ, and for the following sequence of operations:

$$\begin{aligned} \mathsf{sk}, \mathsf{pp} &\leftarrow \text{SEC.KeyGen}(\lambda) \\ \{c_1, \ldots, c_p\} &\leftarrow \text{SEC.Encrypt}(\mathsf{ke}, x_1, \ldots, x_p) \\ \mathsf{eval} &= \text{SEC.Evaluate}(f_{\mathsf{desc}}, c_1, \ldots, c_p, \mathsf{pp}) \\ \mathsf{result} &= \text{SEC.Decrypt}(\mathsf{ke}, \mathsf{eval}), \end{aligned}$$

The following holds with certainty:

$$\Pr[\mathsf{result} = f_{\mathsf{desc}}(x_1, \ldots, x_p)] = 1,$$

Security. SEC guarantees privacy of inputs to a function and output of the evaluation by encrypting them using an IND-CPA secure symmetric-key encryption. Formally, SEC is said to be adaptively secure with respect to a leakage function $\mathcal{L}_{\text{SEC}} = \{\mathcal{L}_{\text{SEC}}^{\text{KeyGen}}, \mathcal{L}_{\text{SEC}}^{\text{Encrypt}}, \mathcal{L}_{\text{SEC}}^{\text{Evaluate}}\}$ if for any PPT adversary $\mathcal{A}$, for a p-bit input $x_1, \ldots, x_p$, there exists a PPT simulator $\text{SIM} = \{\text{SIM}_{\text{KeyGen}}, \text{SIM}_{\text{Encrypt}}, \text{SIM}_{\text{Evaluate}}\}$ such that the following holds:

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

where the "real" experiment $\mathbf{Real}_{\mathcal{A}}^{\text{SEC}}$ and the "ideal" experiment $\mathbf{Ideal}_{\mathcal{A}}^{\text{SEC}}$ are as described in Algorithm 3 and Algorithm 4. $\mathcal{L}_{\text{SEC}}^{\text{KeyGen}}$ captures the leakage from SEC.KeyGen,

$\mathcal{L}^{\text{Encrypt}}_{\text{SEC}}$ captures the leakage from SEC.Encrypt, and $\mathcal{L}^{\text{Evaluate}}_{\text{SEC}}$ captures the leakage from SEC.Evaluate.

Algorithm 3 Experiment $\mathbf{Real}^{\text{SEC}}_{\mathcal{A}}(\lambda)$

function $\mathbf{Real}^{\text{SEC}}_{\mathcal{A}}(\lambda)$
  $(\mathsf{sk}, \mathsf{pp}) \leftarrow \text{KeyGen}(\lambda)$
  for $k \leftarrow 1$ to y ▷ $y = poly(\lambda)$
  do
    Let $[x_1, \ldots, x_p]_k \leftarrow \mathcal{A}(\lambda, \mathsf{pp}, \tau_1, \ldots, \tau_{k-1})$
    Let $[c_1, \ldots, c_p]_k \leftarrow \text{SEC.Encrypt}(\mathsf{sk}, [x_1, \ldots, x_p]_k)$
    Let $f_{\mathsf{desc}}([c_1, \ldots, c_p]_k) \leftarrow \text{SEC.Evaluate}(f_{\mathsf{desc}},$
      $\mathsf{pp}, [c_1, \ldots, c_p]_k)$
    (τ_k denote $\mathcal{A}$'s view after the k^{th} evaluation)
  $b \leftarrow \mathcal{A}(\lambda, \mathsf{pp}, f_{\mathsf{desc}}, \tau_1, \ldots, \tau_y)$
  return b

Algorithm 4 Experiment $\mathbf{Ideal}^{\text{SEC}}_{\mathcal{A},\text{SIM}}(\lambda, \mathcal{L})$

function $\mathbf{Ideal}^{\text{SEC}}_{\mathcal{A},\text{SIM}}(\lambda, \mathcal{L})$
  Parse the leakage function $\mathcal{L}$ as:
  $\mathcal{L} = \left(\mathcal{L}^{\text{KeyGen}}_{\text{SEC}}, \mathcal{L}^{\text{Encrypt}}_{\text{SEC}}, \mathcal{L}^{\text{Evaluate}}_{\text{SEC}}\right)$.
  $(\mathsf{s}_{\text{SIM}}, \mathsf{pp}) \leftarrow \text{SIM}_{\text{KeyGen}}(\mathcal{L}^{\text{KeyGen}}(\lambda))$
  for $k \leftarrow 1$ to y ▷ $y = poly(\lambda)$
  do
    Let $[x_1, \ldots, x_p]_k \leftarrow \mathcal{A}(\lambda, \mathsf{pp}, \tau_1, \ldots, \tau_{k-1})$
    Let $(\mathsf{s}_{\text{SIM}}, [c_1, \ldots, c_p]_k) \leftarrow \text{SIM}_{\text{Encrypt}}(\mathsf{s}_{\text{SIM}},$
      $\mathcal{L}^{\text{Encrypt}}_{\text{SEC}}([x_1, \ldots, x_p]_k); \mathsf{pp})$
    Let $f_{\mathsf{desc}}([c_1, \ldots, c_p]_k) \leftarrow \text{SIM}_{\text{Evaluate}}(\mathsf{s}_{\text{SIM}},$
      $\mathcal{L}^{\text{Evaluate}}_{\text{SEC}}(f_{\mathsf{desc}}, \mathsf{pp}, [c_1, \ldots, c_p]_k))$
    (τ_k denote $\mathcal{A}$'s view after the k^{th} evaluation)
  $b \leftarrow \mathcal{A}(\lambda, \mathsf{pp}, f_{\mathsf{desc}}, \tau_1, \ldots, \tau_y)$
  return b

3.2 SEC Construction

The fundamental goal of SEC is to compute arbitrary sized Boolean circuits using encrypted lookup tables while ensuring data-privacy, in a single round of communication between the client and server. The three universal Boolean function set supported by SEC are $f = \{f_{\texttt{XOR}}, f_{\texttt{AND}}, f_{\texttt{OR}}\}$. SEC incorporates a black-box conjunctive SSE construction to *compute* Boolean functions on encrypted inputs by performing searches over encrypted lookup tables. We refer the reader to Section 1.2 for an overview of SEC, and proceed with the construction here.

SEC.KeyGen. Algorithm 3 formally explains the KeyGen routine. The client executes it and is responsible for creating the client secret key sk and a public parameter pp. pp constitutes ① encrypted lookup tables (i.e. encrypted search indices) for $\{f_{\texttt{XOR}}, f_{\texttt{AND}}, f_{\texttt{OR}}\}$ and ② encrypted search tokens st (i.e. the token set). pp is offloaded to the server.

Algorithm 5 GenDB

Input: Security parameter λ
Output: **DB**

function GenDB(λ)
  $f = \{f_{\texttt{XOR}}, f_{\texttt{AND}}, f_{\texttt{OR}}\}$
  $\texttt{Input\_set} = \{(x, y) \in \{0,1\}^2\}$
  for $i \in 1$ *to* $|f|$ **do**
    for $j \in 0$ *to* $|\texttt{Input\_set}| - 1$ **do**
      $b = f_i(\texttt{Input\_set[j]})$
      $D_j \leftarrow \Pi_{\mathsf{sym}}.\text{Encrypt}_{\mathsf{ke}}(b)$
      $\mathbf{DB}_{f_i} = \mathbf{DB}_{f_i} \cup \{\ \texttt{MAP}(D_j, \texttt{Input\_set[j]}\ )\}$
                                ▷ *MAP is used as an abstraction of Table 4*
    $\mathbf{DB} = \mathbf{DB} \cup \{\mathbf{DB}_{f_i}\}$
  return DB

Concretely, first, a helper function GenDB generates **DB** (see Algorithm 5), which essentially comprises the plaintext mappings between keywords and encrypted documents (Table 3 and Table 5). **DB** is then encrypted into SSE specific data structures using CSSE.Encrypt (See Section 2) to generate the encrypted search index or **EDB**[15], following the discussion established in Section 1.2. Thereafter, the specific search tokens st are generated by invoking CSSE.GenToken (See Section 2). The size of $\mathsf{TokenSet}$ is therefore $\mathcal{O}(n)$, assuming a set of n distinct "special" terms $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$ for a circuit with n gates[16]. Recall from Section 1.2 that `ConstructQuery` routine maps the output of one (inner) SEC evaluation to obtain search tokens for the *next* query (outer function evaluation). Thus, SEC thereby stores all pairwise mappings in $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$ in the table[17] for `ConstructQuery`. Concretely, *any* "special" term $\mathbf{w}_d^k$ (i.e. $\texttt{sval}_{\mathbf{w}_d^k, D_j}$) can be used to obtain search tokens belonging to any other "special" term $\mathbf{w}_d^{k'}$: $\texttt{Token}_{\mathbf{w}_d^{k'}, \mathbf{w}_i, D_j}$ without requiring exponential storage. Finally, all the SEC specific data structures are then offloaded to the server.

Note that KeyGen *is a one-time routine, executed only once at the beginning. We emphasize that by trading off storage,* SEC *achieves the capability of randomizing the order of "special" terms* $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$ *across different executions of circuits, thereby achieving reusability. The client-controlled permutation of these "special" occurs during* SEC.Evaluate.

SEC.Encrypt. This routine is executed on the client's end and is responsible for encrypting and offloading the actual data. As shown in Algorithm 7, the client uses its secret key ke to encrypt plaintext bits $\{x_1, \ldots, x_p\}$ using an IND-CPA secure symmetric-key encryption scheme ($\Pi_{\mathsf{sym}} = (\text{KeyGen}, \text{Encrypt}_{\mathsf{ke}}, \text{Decrypt}_{\mathsf{ke}})$), and returns a ciphertext $\{c_1, \ldots, c_p\}$.

SEC.Evaluate. SEC.Evaluate is executed by the server and is mainly responsible for computing a circuit (whose description is provided by the client) on encrypted inputs. This is summarized in Algorithm 8. We assume the client wants to compute an arbitrary Boolean

[15]Refer to Appendeix 2.1 for conjunctive SSE syntax and security model.
[16]The analysis subsumes $\mathcal{O}(1)$ number of keywords for input combinations (i.e. $\mathbf{w}_1$ and $\mathbf{w}_2$), as well as $\mathcal{O}(1)$ number of documents ($D_j : j \in \{0, 1, 2, 3\}$)
[17]Implemented either as a membership test (thereby allowing use of efficient Bloom Filters), or through a two-dimensional dictionary, as done in state-of-the-art SSE schemes.

Algorithm 6 SEC.KeyGen

Input: Security parameter λ
Output: Client's secret key sk and a public parameter pp

function SEC.KeyGen(λ)
  Samples a uniformly random key ke for an IND-CPA Symmetric-key Encryption scheme: $\Pi_{\mathsf{sym}} = (\text{KeyGen}, \text{Encrypt}_{\mathsf{ke}}, \text{Decrypt}_{\mathsf{ke}})$
  $\mathbf{DB} \leftarrow \text{GenDB}(\lambda)$
  $(\mathbf{EDB}, \mathsf{sk}) \leftarrow \text{CSSE.Encrypt}(\mathbf{DB})$
  for all conjunctive query q of keywords in $\mathbf{DB}$ **do**
    $\{\mathsf{st}_q\} = \text{CSSE.GenToken}(\mathsf{sk}, q)$
  $\mathsf{TokenSet} = \mathsf{TokenSet} \cup \{\mathsf{st}_q\}$
  return $\mathsf{sk}, \mathsf{pp} = \{\mathbf{EDB} \cup \mathsf{TokenSet}\}$

Algorithm 7 SEC.Encrypt

Input: $\mathsf{ke}, x_1, \ldots, x_p$
Output: $c_1, \ldots, c_p$

function SEC.Encrypt($\mathsf{ke}, x_1, \ldots, x_p$)
  Encrypt input bits $\{x_1, \ldots, x_p\}$ using an IND-CPA secure symmetric-key encryption scheme Π_{sym} with client secret-key ke
  $\{c_1, \ldots, c_p\} \leftarrow \Pi_{\mathsf{sym}}.\text{Encrypt}_{\mathsf{ke}}(x_1, \ldots, x_p)$
  return $\{c_1, \ldots, c_p\}$

circuit:

$$f_{\mathsf{cir\_depth}}(f_{\mathsf{cir\_depth}-1}(x_{\mathsf{cir\_depth}-1}, y_{\mathsf{cir\_depth}-1}), \ldots, f_1(x_1, y_1))$$

where $f_j \in \{f_{\texttt{XOR}}, f_{\texttt{AND}}, f_{\texttt{OR}}\}$ and (x_j, y_j) are inputs to the j-th function f_j (where $1 \leq j \leq \mathsf{cir\_width}$; $\mathsf{cir\_width}$ being the maximal width of the circuit) at depth i (for $1 \leq i \leq \mathsf{cir\_depth}$; $\mathsf{cir\_depth}$ being the depth of the entire circuit). The server runs the SEC.Evaluate algorithm that takes as input a circuit description f_{desc}, encrypted inputs $\{c_1, \ldots, c_p\}$, and the public parameter pp (already offloaded to the server at the end of SEC.KeyGen). It parses pp as $\{\mathbf{EDB}, \mathsf{TokenSet}\}$ and then uses the encrypted inputs $\{c_1, \ldots, c_p\}$ to retrieve the corresponding *search tokens* (st_{q_f}) for all j functions at depth 1 of the circuit. Next, for each of the j-th function at depth i ($1 \leq i \leq \mathsf{cir\_depth}$), it invokes the CSSE.Search algorithm using a specific search token, sk and the $\mathbf{EDB}$ as input. The output obtained (i.e. $\mathsf{eval}_{j,i}$) is in turn used to retrieve the search tokens for the functions at depth $(i+1)$ of the circuit by calling $\texttt{ConstructQuery}(\mathsf{eval}_{j,i})$. It is to be noted that $\mathsf{eval}_{j,i}$ encapsulates ($\mathbf{w}_d^k$, $\mathtt{sval}_{\mathbf{w}_d^l, D_j}, b$) and retrieves the search token st_{q_f} (which is of the form $\mathtt{Token}_{\mathbf{w}_d^k, \mathbf{w}_i, D_j}$). Bit b is used to determine whether to use the retrieved search token as the first or second input to the outer function. Again CSSE.Search is called with the new search token query and $\mathbf{EDB}$ as the input. This process continues till the last level of the circuit. Note, that each call to CSSE.Search can be made in parallel for all j functions at a certain depth of the circuit, since all functions at depth/level i are pairwise independent wrt. required inputs. The final encrypted evaluation eval returned by the last level of the circuit contains the encrypted bit corresponding to the actual output of the entire circuit evaluation. eval is returned to the client as the final encrypted output of the circuit f_{desc}.

SEC.Decrypt. The client runs the SEC.Decrypt algorithm to decrypt the encrypted evaluation eval. The decrypted value result is equal to the evaluation of the circuit f_{desc} on

Algorithm 8 SEC.EVALUATE

Input: $f_{\mathsf{desc}}, \{c_1, \ldots, c_p\}, \mathsf{pp}$
Output: eval

function SEC.EVALUATE($f_{\mathsf{desc}}, \{c_1, \ldots, c_p\}, \mathsf{pp}$)
  Parse $\mathsf{pp} = \{\mathbf{EDB}, \mathsf{TokenSet}\}$
  Retrieve *search tokens* st_{q_f} from $\mathsf{TokenSet}$ using $\{c_1, \ldots, c_p\}$ ▷ $\mathtt{ConstructQuery}(\mathsf{TokenSet}, \{c_1, \ldots, c_n\})$
  for i= 1 to cir_depth-1 **do** ▷ cir_depth *is depth of the circuit* f_{desc}
    $\mathsf{eval}_{j,i}$ = CSSE.SEARCH($\mathsf{sk}, \mathbf{EDB}, \mathsf{st}_{q_f}$) ▷ *for a circuit of width* j *at depth* i*, run* j *instances of* CSSE.SEARCH *in parallel*
    Retrieve *search tokens* $\mathsf{st}_{q_f} \leftarrow \mathtt{ConstructQuery}(\mathsf{eval}_{j,i})$
  eval = CSSE.SEARCH($\mathsf{sk}, \mathbf{EDB}, \mathsf{st}_{q_f}$)
  return eval to the client at the end of the protocol

Algorithm 9 SEC.DECRYPT

Input: $\mathsf{ke}, \mathsf{eval}$
Output: result

function SEC.DECRYPT($\mathsf{ke}, \mathsf{eval}$)
  $\mathsf{result} = \Pi_{\mathsf{sym}}.\text{DECRYPT}_{\mathsf{ke}}(\mathsf{eval})$
  return result

$(x_1, \ldots, x_p)$.

3.3 Proof of Correctness of SEC

The proof of correctness for SEC follows from the correctness of CSSE. The correctness of CSSE ensures that a conjunctive query $q = \mathbf{w}_1 \wedge \ldots \wedge \mathbf{w}_\mathsf{n}$ over an encrypted database satisfies the following relations (we refer to Section 2.1 for generic conjunctive SSE syntax):

$$\begin{aligned} \mathsf{sk} &\leftarrow \text{CSSE.KEYGEN}(\lambda) \\ \mathbf{EDB} &\leftarrow \text{CSSE.ENCRYPT}(\mathsf{sk}, \mathbf{DB}) \\ \mathsf{st}_q &= \text{CSSE.GENTOKEN}(\mathsf{sk}, q) \\ \mathbf{DB}(\mathbf{w}_1) \cap \ldots \cap \mathbf{DB}(\mathbf{w}_\mathsf{n}) &= \text{CSSE.SEARCH}(\mathbf{EDB}, \mathsf{st}_q) \end{aligned}$$

Proof. By deploying CSSE as a black-box, SEC generates the encrypted search index **EDB** specific to the function set $f = \{f_{\mathtt{XOR}}, f_{\mathtt{AND}}, f_{\mathtt{OR}}\}$ supported by SEC. The encrypted search index **EDB** consists of keywords corresponding to encrypted input bits and documents that encapsulate the encrypted output of the function evaluation. It also invokes the CSSE.GENTOKEN algorithm that generates search tokens corresponding to the keywords (that map to encrypted input bits of a function). The search tokens are stored in a $\mathsf{TokenSet}$ and then $\mathsf{pp} = \{\mathbf{EDB} \cup \mathsf{TokenSet}\}$ is offloaded to the server. For evaluating a function $f(x, y)$, the function description f_{desc} (which is essentially a single binary function in this case) and encrypted inputs $\{c_1, c_2\}$ (corresponding to plaintext bits x and y respectively) are sent to the server. The server retrieves the search tokens from $\mathsf{TokenSet}$ and invokes CSSE.SEARCH protocol with the search token query st_{qf} and **EDB** as input. It retrieves a document from **EDB** that consists of the encrypted output of the computation, (denoted as eval) and sends it to the client. The client decrypts eval locally using the

SEC.Decrypt function and obtains the output bit result which is equal to $f(x, y)$. Correctness is hence a combination of correctness of CSSE, along with SEC specific mappings (Table 4 and Table 5) that ensure *exactly* one document being returned by CSSE.Search, which is the *correct* result of respective gate evaluation.

Following the functionally correct evaluation of a single binary gate $f(\cdot, \cdot)$, the correct encrypted evaluation eval returned by SEC.Evaluate for any arbitrary circuit f_{desc} over encrypted inputs $\{c_1, \ldots, c_p\}$ can be asserted transitively. This is because f_{desc} is essentially viewable as a collection of several binary gates $f(\cdot, \cdot)$ which are functionally *correct* as established above, and thereby ensures that the output eval returned by SEC.Evaluate on f_{desc} on decryption is equal to $f_{\mathsf{desc}}(x_1, \ldots, x_p)$ for any circuit f_{desc} and unencrypted input set $\{x_1, \ldots, x_p\}$. Thereby, we conclude that *for a functionally correct and exact conjunctive SSE scheme* CSSE*, a set of functions* $f = \{f_{\mathtt{XOR}}, f_{\mathtt{AND}}, f_{\mathtt{OR}}\}$*, and encrypted values* $\{c_1, \ldots, c_p\}$ *of input* $\{x_1, \ldots, x_p\}$, SEC *is functionally correct, since for the following sequence of operations:*

$$\begin{aligned} \mathsf{sk}, \mathsf{pp} &\leftarrow \text{SEC.KeyGen}(\lambda) \\ \{c_1, \ldots, c_p\} &\leftarrow \text{SEC.Encrypt}(\mathsf{ke}, x_1, \ldots, x_p) \\ \mathsf{eval} &= \text{SEC.Evaluate}(f_{\mathsf{desc}}, c_1, \ldots, c_p, \mathsf{pp}) \\ \mathsf{result} &= \text{SEC.Decrypt}(\mathsf{ke}, \mathsf{eval}) \end{aligned}$$

result is the decrypted output of the evaluation of a circuit as specified by f_{desc} on encrypted inputs $c_1, \ldots, c_p$, such that $\mathsf{result} = f_{\mathsf{desc}}(x_1, \ldots, x_p)$ (where, $x_1, \ldots, x_p$ is the unencrypted input), i.e.,

$$\Pr[\mathsf{result} = f_{\mathsf{desc}}(x_1, \ldots, x_p)] = 1,$$

3.4 Correctness

For a functionally correct and exact conjunctive SSE scheme CSSE, a function description f_{desc} derived from a set of functions $f = \{f_{\mathtt{XOR}}, f_{\mathtt{AND}}, f_{\mathtt{OR}}\}$, ciphertexts $\{c_1, \ldots, c_p\}$ of input $\{x_1, \ldots, x_p\}$, SEC is functionally correct for the following sequence of operations:

$$\begin{aligned} \mathsf{sk}, \mathsf{pp} &\leftarrow \text{SEC.KeyGen}(\lambda) \\ \{c_1, \ldots, c_p\} &\leftarrow \text{SEC.Encrypt}(\mathsf{ke}, x_1, \ldots, x_p) \\ \mathsf{eval} &= \text{SEC.Evaluate}(f_{\mathsf{desc}}, c_1, \ldots, c_p, \mathsf{pp}) \\ \mathsf{result} &= \text{SEC.Decrypt}(\mathsf{ke}, \mathsf{eval}) \end{aligned}$$

if and only if the following holds:

$$\Pr[\mathsf{result} = f_{\mathsf{desc}}(x_1, \ldots, x_p)] = 1$$

3.5 Practical Instantiation of SEC

Our generic privacy-preserving computation framework SEC can be practically implemented by deploying any conjunctive SSE scheme as a black-box. We provide concrete constructions of SEC using two conjunctive SSE schemes:

SEC$_{\mathsf{OXT}}$. We analyze a concrete instantiation of SEC based on the OXT protocol [16] (See Section 2.3 for a brief overview), abbreviated SEC$_{\mathsf{OXT}}$. Our analysis fundamentally covers the complexity of SEC$_{\mathsf{OXT}}$ in terms of performance and storage overhead, and a formal security analysis based on a well-defined leakage profile. We provide a detailed complexity analysis of SEC$_{\mathsf{OXT}}$ based on our experimental results in Section 6. The leakage profile and security proof of SEC$_{\mathsf{OXT}}$ are elaborated in Section 4.

SEC$_{\text{ConjFilter}}$. We also demonstrate the scalability and complexity overhead of a second instantiation of SEC by deploying a purely symmetric-key based (plausibly quantum-safe) conjunctive SSE, ConjFilter [45]. We provide performance and storage overhead analysis of SEC$_{\text{ConjFilter}}$ in Section 6.

3.6 Complexity Analysis of SEC

Storage Overhead. The storage requirement of SEC depends upon the number of functions it supports along with the number of search tokens (input combinations) for every function. The final **DB** is collectively composed of three sub-databases (search indices) $\mathbf{DB} = \{\mathbf{DB}_{\mathtt{AND}}, \mathbf{DB}_{\mathtt{OR}}, \mathbf{DB}_{\mathtt{XOR}}\}$ encrypted and offloaded to the server. As discussed in Section 1.2 and Section 3.2, each function-specific sub-database contains exactly *four* keywords[18], which are mapped to exactly *two* documents (out of four possible documents). The bulk of storage overhead comes from the set of n distinct "special" terms $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$ (for a circuit with n binary gates), for which SSE specific data structures occupy $\mathcal{O}(n)$ space. Some constant number of *dummy* documents are also added to maintain a uniform frequency of each keyword in the final encrypted database. Concretely, there are exactly four keywords for one binary function, four documents (including dummy documents) per keyword, and n "special" terms. Total storage for three gates can be calculated as -

$$\mathsf{Total\ Storage} = [(((4+n)\times 4)\times 3)\cdot b] \text{ bytes} = O(n)$$

where b is a constant.

Computation and Communication Overhead. The evaluation time of SEC for computing arbitrary sized Boolean circuit over encrypted data scales linearly with the search time complexity of the underlying CSSE scheme, which in turn depends upon the depth of the circuit. The crux of SEC is to bypass explicit circuit evaluation as done in state-of-the-art encrypted computation schemes like FHE and leverage the extremely efficient encrypted search capability of a CSSE scheme to evaluate functions on encrypted inputs.

By constructing the mapping for all pairwise combinations in $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$, multiple circuits can be executed securely without the need to refresh SEC's data structures. This is achieved by ensuring ① no two gates in the *same* circuit share a "special" term, and

[18] We consider this as $\mathcal{O}(1)$ overhead in our analysis.

② two isomorphic gates in *different* circuits also do not share corresponding "special" terms. Hence, the client's communication overhead is majorly because of the *one-time* setup, which is dominated by "special terms" (see storage analysis above; consequently, randomly client-generated permutations of $\{\mathbf{w}_d^1, \mathbf{w}_d^2, ..., \mathbf{w}_d^n\}$ are used to evaluate subsequent circuits. Overall, the communication complexity is bounded by $\mathcal{O}(n)$.

4 Security and Leakage Profile Analysis of SEC

We analyze the security of SEC in this section. We follow a semi-honest adversarial setting for our security analysis where the remote server is assumed to be honest-but-curious. This implies the untrusted server follows the algorithmic specification exactly, but can also observe and record additional information for analysis. Using the ideal/real world paradigm, we establish formally that a probabilistic polynomial-time (PPT) simulator can simulate the view of the adversarial server in an indistinguishable manner given only the leakage profile of SEC.

CSSE Leakage Profile. We now detail how SEC inherits security properties and leakage profile from the underlying conjunctive SSE (CSSE) construction (Refer to Section 2.1,2.3 for more details). We note that CSSE is an adaptively secure conjunctive SSE scheme against a semi-honest adversary $\mathcal{A}$. The leakage of CSSE is characterized by the leakage function $\mathcal{L}_{\text{CSSE}}$ which is an ensemble of the leakage functions for Encrypt, GenToken and Search individually, expressed as:

$$\mathcal{L}_{\text{CSSE}} = \{\mathcal{L}_{\text{CSSE}}^{\text{Encrypt}}, \mathcal{L}_{\text{CSSE}}^{\text{GenToken}}, \mathcal{L}_{\text{CSSE}}^{\text{Search}}\}$$

SEC Leakage Profile. Given the above CSSE leakage functions, security of SEC can be analyzed using SEC leakage function $\mathcal{L}_{\text{SEC}}$ in the same adaptive semi-honest adversarial model. $\mathcal{L}_{\text{SEC}}$ is composed of two separate leakage functions for KeyGen and Encrypt (as expressed below), that capture the leakage from SEC.KeyGen and SEC.Encrypt execution respectively.

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

Informally, note that $\mathcal{L}_{\text{SEC}}$ subsumes $\mathcal{L}_{\text{CSSE}}$ plus leakages from SEC specific operations. Our security analysis thereby considers the security of SEC in presence of $\mathcal{L}_{\text{CSSE}}$ plus any leakage from any SEC specific operations.

Concrete Security of $\text{SEC}_{\mathsf{OXT}}$. Note that $\mathcal{L}_{\text{CSSE}}$ is CSSE specific. For a concrete security analysis thereby, we provide the security analysis of $\text{SEC}_{\mathsf{OXT}}$ (i.e. SEC instantiated with OXT as the CSSE (Section 2.3)), which follows from the security notions of generic SEC and underlying OXT protocol.[19]

[19] Analysis for SEC_{Conj} follows suit, since the leakage profile of SEC is agnostic of the CSSE used.

5 Security Analysis and Discussion on Leakage Profile of $\mathsf{SEC}_{\mathsf{OXT}}$

We formally explain the leakage profile for the specific instantiation of SEC based on the OXT scheme, namely $\mathrm{SEC}_{\mathsf{OXT}}$.

5.1 Leakage Profile of $\mathsf{SEC}_{\mathsf{OXT}}$

The significance of each component of the leakage function in $\mathrm{SEC}_{\mathsf{OXT}}$ is comparable to that of OXT. 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 a trivial leakage in the literature of SSE.

• 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 $\mathsf{SP}[i] = |\mathbf{DB}(\mathsf{sterm}[i])|$. It leaks the number of documents satisfying the sterm in a query. In SEC this is always constant (equal to 2), since each keyword (encrypted input bit) maps to exactly four documents (encrypted output of function evaluated on the particular encrypted input) in **EDB**.

• RP - result pattern of the queries or the indices of documents matching the entire conjunction. Formally, RP is vector of size n with $\mathsf{RP}[i] = \mathbf{DB}(\mathsf{sterm}[i]) \cap \mathbf{DB}(\mathsf{xterm}[i])$ for each $i = 1, \ldots, n$ where xterm refers to all the other keywords in the conjunctive query other than the sterm. It is the final output of the search query and is not considered a real leakage in the context of SSE. This is always a single document in SEC.

• IP - conditional intersection pattern, a $n \times n$ table defined as -

$$\mathsf{IP}[i,j] = \begin{cases} \mathbf{DB}(s[i]) \cap \mathbf{DB}(s[j]), \\ \quad if\ i \neq j\ and\ x[i] = x[j] \\ \phi, \quad otherwise \end{cases}$$

IP captures a leakage which occurs due to the specialised computation of xtags and storing a unique xtag value in the XSet for each ($\mathsf{w\text{-}id}$) pair. Fundamentally in SEC IP would capture the leakage which occurs when two distinct function evaluation have a common xterm (or second input) but different sterm (or first input) 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 leaks nothing significant in SEC because in our construction sterms are nothing but dummy keywords that are mapped to all the documents present in the database.

5.2 Analysis of Potential Leakages in $\mathsf{SEC}_{\mathsf{OXT}}$

The database (search index) generation process and algorithmic design of $\mathrm{SEC}_{\mathsf{OXT}}$, renders certain non-trivial information leakage insignificant or redundant, which is otherwise considered crucial by the underlying OXT scheme and could lead to potential correlation inference

by the server between two encrypted function computation. Due to the uniform keyword frequency and selection of random "special" term for each query by the client, $\text{SEC}_{\mathsf{OXT}}$ restricts certain non-trivial leakages like *size-pattern*, *result-pattern*, *equality-pattern*, *conditional intersection pattern* leakages that analyze the frequency pattern of the "special" terms and "cross" terms in OXT over multiple conjunctive queries. This makes OXT vulnerable to certain state-of-the-art leakage-abuse attacks [53, 8].

• *Size Pattern Leakage (SP).* It leaks the number of documents satisfying the "special" term in a query. In $\text{SEC}_{\mathsf{OXT}}$ since every keyword (input bit) maps to exactly four documents (encapsulating encrypted output bits), this leakage reveals no significant information.

• *Result Pattern Leakage (RP).* It is the final output of the search query i.e. indices of documents matching the entire conjunction. By the design of $\text{SEC}_{\mathsf{OXT}}$ the result of a conjunctive query (binary function evaluation) is always a single document (single encrypted output bit) and hence this does not reveal any significant information to the server.

• *Equality pattern (EP).* It indicates which queries have the equal "special" terms. This occurs due to the optimization technique devised in OXT in order to ensure sub-linear search complexity by filtering out the least frequent term (sterm) during the search. In $\text{SEC}_{\mathsf{OXT}}$, since the frequency of all keywords is the same, the client chooses a different "special" term from a pool of dummy keywords ($\mathbf{w}_d^k$: $k \in \{1, \ldots, n\}$) for different queries (gate evaluation). Hence, the adversary will not be able to infer any correlation for multiple gate evaluation over multiple (repeated/non-repeated) inputs.

• *Conditional Intersection Pattern Leakage (IP).* It is a subtle leakage in OXT that occurs when two distinct queries have a common "cross" term but a different "special" term and there exists a document that satisfies both the "special" terms. In such a scenario the set of document indices matching both "special" terms is leaked (if no document matching both "special" terms exists then nothing is leaked). In $\text{SEC}_{\mathsf{OXT}}$ this leakage will not reveal any significant information about the underlying input bits to a function or the output of a function evaluation, since all the input/output bits are encrypted using an IND-CPA secure symmetric-key encryption scheme. The server cannot gain any high entropy information from this leakage, as all the sterms are "special" terms in SEC that are mapped to every documents in the database.

All these leakages except N (total size of $\mathbf{EDB}$) are essentially encapsulated by $\mathcal{L}_{\mathsf{OXT}}^{\text{Search}}$, and hence are leaked by the SEC.Evaluate algorithm (encapsulated by $\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}^{\text{Evaluate}}$). As explained above none of these leakages have a significant impact on the data privacy guarantees of SEC.

Theorem 1 *Given that* OXT *is an adaptively secure* CSSE *scheme with respect to the leakage function* $\mathcal{L}_{\mathsf{OXT}} = \{\mathcal{L}_{\mathsf{OXT}}^{\text{Encrypt}}, \mathcal{L}_{\mathsf{OXT}}^{\text{GenToken}}, \mathcal{L}_{\mathsf{OXT}}^{\text{Search}}\}$ *against a polynomially-bounded adaptive adversary,* $\text{SEC}_{\mathsf{OXT}}$ *is also an adaptively secure encrypted computation framework with respect to the leakage function* $\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}} = \{\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}^{\text{KeyGen}}, \mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}^{\text{Encrypt}}, \mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}^{\text{Evaluate}}\}$*, where the* $\text{SEC}_{\mathsf{OXT}}$ *instantiation encrypts an input* $\{x_1, \ldots, x_p\}$ *using an IND-CPA secure symmetric-key encryption scheme to obtain corresponding encrypted bits* $\{c_1, \ldots, c_p\}$ *over which a (publicly known) function* f *is evaluated, where* f *is composed of functions from* $\{f_{\texttt{XOR}}, f_{\texttt{AND}}, f_{\texttt{OR}}\}$.

Proof 1 *We give an extensive security analysis of* $\text{SEC}_{\mathsf{OXT}}$ *through formal proof of Theo-*

rem 1 next.

The security analysis of $\text{SEC}_{\mathsf{OXT}}$ (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 adaptive semantic security of OXT implies adaptive security guarantees of $\text{SEC}_{\mathsf{OXT}}$.

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

Leakage Cover. We briefly describe why each of the individual leakage components ($\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}} = \{\mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}}, \mathcal{L}^{\text{Encrypt}}_{\text{SEC}_{\mathsf{OXT}}}, \mathcal{L}^{\text{Evaluate}}_{\text{SEC}_{\mathsf{OXT}}}\}$) are necessary for a simulator to produce correct results. To simulate $\text{SEC}_{\mathsf{OXT}}$ correctly each of the leakage components are critically analyzed and their significance is justified. N or the total number of appearances of keywords in the database gives the size of the **EDB**, which is encapsulated by the public parameter pp along with the size of search token set $|\mathsf{TokenSet}|$.

Simulating $\text{SEC}_{\mathsf{OXT}}$ KeyGen and Encrypt. In OXT the **EDB** comprises of two data structures $\textbf{EDB} = \{\mathsf{TSet}, \mathsf{XSet}\}$. The main crux of our adaptive security proof is that the simulator for $\text{SEC}_{\mathsf{OXT}}$ initializes the XSet and $\mathsf{TokenSet}$ to consist entirely of uniformly random elements from a *discrete log* hard group initially (while relying on the DDH assumption for indistinguishability of the real and simulated XSet and $\mathsf{TokenSet}$ entries). Additionally, the simulator for $\text{SEC}_{\mathsf{OXT}}$ can directly invoke the simulator for the adaptively secure TSet to simulate the TSet entries at Encrypt. Overall $\text{SIM} = \{\text{SIM}_{\text{KeyGen}}, \text{SIM}_{\text{Encrypt}}, \text{SIM}_{\text{Evaluate}}\}$ takes as input the leakage components as defined by $\mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}}$. N and $|\mathsf{TokenSet}|$ is essentially learned from the public parameter pp, hence, $\mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}} = \perp$, $\mathcal{L}^{\text{Encrypt}}_{\text{SEC}_{\mathsf{OXT}}}$ that reveals the length n of the ciphertext $c_1, \ldots, c_n$, and $\mathcal{L}^{\text{Evaluate}}_{\text{SEC}_{\mathsf{OXT}}}$ reveals $\{\texttt{SP, RP, EP, CIP}\}$. Using the leakages and the public parameter the SIM then produces the ciphertext $c_1, \ldots, c_n$ which is indistinguishable from the encrypted output returned by the original SEC.Encrypt algorithm on an input $x_1, \ldots, x_n$.

Simulating $\text{SEC}_{\mathsf{OXT}}$.KeyGen. We observe that, $\text{SEC}_{\mathsf{OXT}}$.KeyGen comprises of the GenDB, OXT.Encrypt (CSSE.Encrypt), OXT.GenToken (CSSE.GenToken) (this phase is encapsulated in the OXT.Search protocol in [16], and is entirely executed by the client). Without loss of generality, we extract the search token generation phase (OXT.GenToken) and store all possible search tokens in $\mathsf{TokenSet}$ during $\text{SEC}_{\mathsf{OXT}}$.KeyGen. A number of values generated by pseudo-random functions (PRF) and group operations, are inserted into TSet using TSet.Setup [16] and XSet respectively during OXT.Encrypt. 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.

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

Thus, the simulator $\text{SIM}_{\text{KeyGen}}$ can exactly simulate GenDB execution straightforwardly.

Subsequently, the OXT.Encrypt is invoked with the plain **DB** generated by GenDB. Since the OXT.Encrypt algorithm is executed in a black-box way, the leakage from $\text{SEC}_{\mathsf{OXT}}$.KeyGen is same as the OXT.Encrypt executed over **DB**. Also, OXT.GenToken phase is invoked on all possible queries q of keywords (alphanumeric translations of encrypted input bits) in **DB**. This algorithm is also used a black-box and is entirely executed by the client. Thus, the leakage can be expressed as below.

$$\mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}} = \{\mathcal{L}^{\text{Encrypt}}_{\mathsf{OXT}}(\mathbf{DB}), \mathcal{L}^{\text{GenToken}}_{\mathsf{OXT}}(q)\},$$

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

$$\mathcal{L}^{\text{KeyGen},\mathsf{TSet}}_{\text{SEC}_{\mathsf{OXT}}} = N = \perp \text{ as this is a public information}$$

$\text{SIM}_{\text{KeyGen}}$ can run the TSet simulator (as discussed in the original paper [17]). Combined all, the simulator for KeyGen ($\text{SIM}_{\text{KeyGen}}$) simulates $\text{SEC}_{\mathsf{OXT}}$.KeyGen with access to following the leakage.

$$\mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}} = \{\mathcal{L}^{\text{GenDB}}, \mathcal{L}^{\text{KeyGen}}_{\text{SEC}_{\mathsf{OXT}}}, \mathcal{L}^{\text{KeyGen},\mathsf{TSet}}_{\text{SEC}_{\mathsf{OXT}}}\},$$

Simulating $\text{SEC}_{\mathsf{OXT}}$.Encrypt. For simulating $\text{SEC}_{\mathsf{OXT}}$.Encrypt the simulator $\text{SIM}_{\text{Encrypt}}$ observes $\mathcal{L}^{\text{Encrypt}}_{\text{SEC}_{\mathsf{OXT}}}$ which is equal to the length of the ciphertext $c_1, \ldots, c_n$.

$$\mathcal{L}^{\text{Encrypt}}_{\text{SEC}_{\mathsf{OXT}}} = |c_1, \ldots, c_n|,$$

The $\text{SEC}_{\mathsf{OXT}}$.Encrypt algorithm invokes an IND-CPA secure symmetric-key encryption scheme which is used to encrypt an input bit $x_1, \ldots, x_n$. The $\text{SIM}_{\text{Encrypt}}$ produces a ciphertext $c1, \ldots, c_n$ corresponding to the input only with the information from $\mathcal{L}^{\text{Encrypt}}_{\text{SEC}_{\mathsf{OXT}}}$ and its state (s_{SIM}). The ciphertext thus produced by $\text{SIM}_{\text{Encrypt}}$ is indistinguishable from the ciphertext produced by $\text{SEC}_{\mathsf{OXT}}$.Encrypt in the real scheme.

Simulating $\text{SEC}_{\mathsf{OXT}}$.Evaluate. The Evaluate function takes as input encrypted bits $c_1, \ldots, c_n$ and retrieves *search tokens* from the $\mathsf{TokenSet}$ using the encrypted input bits (as input to the $\mathsf{ConstructQuery}$ subroutine). It then invokes the CSSE.Search function using the search tokens and gets an encrypted bit as output. This process is repeated for all gates at every depth of the circuit being evaluated. The $\mathsf{ConstructQuery}$ subroutine leaks essentially no information to the server. This is because the input bits are encrypted using an IND-CPA symmetric-key encryption algorithm and the search tokens to be searched are dependent on the "special" term for that particular function/query (which can be selected at random by the client and the number of possible permutation of "special" terms for a circuit with n gates is upper bounded by $O(n!)$) and also the function to be evaluated. Therefore, the search pattern in the $\mathsf{TokenSet}$ for any repeated input bits cannot be correlated by the server. This proves that -

$$\mathcal{L}^{\mathsf{ConstructQuery}} = \perp,$$

The leakage $\mathcal{L}^{\text{Evaluate}}_{\text{SEC}_{\mathsf{OXT}}}$ is therefore exactly similar to $\mathcal{L}^{\text{Search}}_{\mathsf{OXT}}$. Hence, we can write the following.

$$\mathcal{L}^{\text{Evaluate}}_{\text{SEC}_{\mathsf{OXT}}} = \{\mathcal{L}^{\text{Search}}_{\mathsf{OXT}}(\mathbf{EDB}, st_q)\},$$

By the simulation security guarantee of OXT, $\text{SEC}_{\mathsf{OXT}}$ is secure against these leakages. We show that the leakages from OXT.Search phase do not have any detrimental effect on the encrypted function evaluations in SEC.

Let a client evaluate a circuit of the form $f_3(f_2(\mathbf{x}_1, \mathbf{x}_2), f_1(\mathbf{x}_3, \mathbf{x}_4))$. The client sends the encrypted input bits to SEC.Evaluate. A two-input function evaluation is translated to a three-keyword conjunctive query, where the first keyword is a "special" term (chosen randomly by the client), the second keyword corresponds to the first input bit, and the third keyword to the second input bit. The client maintains a state of record of all "special" terms used. For a circuit with n gates, the client chooses n "special" terms, which can be permuted and selected in $n!$ different ways (upper bounded by $O(n!)$). The client sends a set of "special" terms to be used at each gate in a circuit. Since the "special" terms are used to fetch the records from memory, permuting "special" terms for consecutive (same) gate evaluation ensures no repetition of the same "special" term hence, the same memory location is not accessed twice.

Case-I: If the same circuit is evaluated twice, the "special" terms are permuted (which is upper bounded by $O(n!)$). Hence the server cannot distinguish between two identical gate evaluations.

Case-II: The search tokens generated corresponding to encrypted input bits are a function of the "special" term and the function being evaluated. Thus, for each gate, the search tokens depend on "special" terms ($n!$ possible combinations). This ensures that an adversary cannot distinguish between two isomorphic gate evaluation.

This proves that the leakages incurred by the underlying CSSE.Search algorithm does not compromise the security of SEC.Evaluate. The output of SEC.Evaluate is indistinguishable from random by the security guarantees of an IND-CPA secure symmteric-key encryption scheme. Since, OXT is proven simulation secure it follows from the simulation security guarantee that $\mathcal{A}$ no additional advantage over the real experiment. This implies the **Real** experiment of SEC (Algorithm 3) is indistinguishable from the **Ideal** experiment (Algorithm 4), and proves Theorem 1.

5.3 $\mathcal{L}_{\mathbf{SEC}}$ and Reusability of SEC specific data structures

The design rationale of SEC guarantees data privacy: privacy of the input bits to a function and the output bit returned after function evaluation. Note that SEC leverages the encrypted search capability of an efficient CSSE scheme to perform encrypted computation. We now detail how $\mathcal{L}_{\text{SEC}}$ evolves from $\mathcal{L}_{\text{CSSE}}$.

Details on $\mathcal{L}_{\mathbf{CSSE}}$. The most generic notion of SSE with optimal guarantees on security is achievable through Oblivious RAM [35], which allows evaluation of search queries without leaking *anything* to the server[20]. However, such *ideal* security guarantees come at immense computational/communication overheads. Hence, most modern SSE constructions trade-off security for efficiency by allowing calculated, acceptable leakages. Some usual leakages:

[20] As with state-of-the-art FHE and SSE constructions, we assume semi-honest server. That is, the server acts as a passive adversary which does not deviate from the protocol.

- **Access Pattern of "special" term**: Two queries having the *same* "special" term can be correlated by the server by the same set of $\texttt{sval}_{\mathbf{w},D_j}$ returned. We emphasize that the server learns not the plaintext alphanumeric value of $\mathbf{w}$, but rather the fact that two queries share the same "special" term. Data privacy of $\mathbf{w}$ thus still remains intact.
- **Access Pattern of "actual" keywords**: Two queries having a common document matched to their "special" term and the same cross term tuples (for example, $(\mathbf{w}, \mathbf{w}_i)$) are leaked since the same $\texttt{Token}_{\mathbf{w},\mathbf{w}_i,D_j}$ is generated for both queries. As before, the server can not learn the underlying plaintext alphanumeric value of $\mathbf{w}_i$; it can simply correlate same cross-terms across two queries. Data privacy of $\mathbf{w}_i$ thus still remains intact.
- **Query Result Pattern**: Two queries having the same result (i.e. $\texttt{sval}_{\mathbf{w},D_j}$) can be correlated by the server. Data privacy of D_j still remains intact.

Evolving $\mathcal{L}_{\mathbf{SEC}}$ from $\mathcal{L}_{\mathbf{CSSE}}$. To the best of our knowledge, state-of-the-art SSE constructions *tolerate* such correlations made by the semi-honest server. However, when we use the search capabilities of SSE to *compute*, leaking these correlations essentially allows a server to learn: ① when inputs of two different gates evaluations are same, and ② when the output of two different gates is same. We stress that while the *exact* plaintext input/output bit can not be leaked (thereby not violating data privacy guaranteed by the IND-CPA symmetric-key encryption scheme used to encrypt the data); still, such correlations between different computations are undesirable non-trivial leakages.

As such, we focus not on plugging these leakages, but rather on *unlinking* the computation from such leakages. In other words, we allow the server to learn these leakages; but embed no critical information in such leakages. To do so, we first observe the aforementioned leakages: *the "special" term is present in all leakage functions*, be it whether the leakage occurs through $\texttt{sval}_{\mathbf{w},D_j}$ or through $\texttt{Token}_{\mathbf{w},\mathbf{w}_i,D_j}$. Hence, in our construction, we *do not embed any computation-related information in the "special" term.* From Table 3 it is observed that the actual bits participating in computing `XOR` are independent of the choice of the "special term" $\mathbf{w}$. This design choice allows SEC to change the "special" term across multiple queries. The server still learns the aforementioned leakages; however, no useful correlations as to the underlying computation are revealed.

Example. Re-consider the problem of computing `XOR(1,1)` using SEC (from Section 1.2), but now the computations happen twice. As such, two queries $q_1 = (\mathbf{w}_d^1 \wedge \mathbf{w}_1 \wedge \mathbf{w}_2)$ and $q_2 = (\mathbf{w}_d^2 \wedge \mathbf{w}_1 \wedge \mathbf{w}_2)$ are issued by a black-box CSSE scheme. We enumerate the leakages visible across these queries:

- **Access Pattern of "special" term**: q_1 leaks $[\texttt{sval}_{\mathbf{w}_d^1,D_j} : j \in \{0,1,2,3\}]$. Likewise, q_2 leaks $[\texttt{sval}_{\mathbf{w}_d^2,D_j} : j \in \{0,1,2,3\}]$
- **Access Pattern of "actual" keywords**: q_1 leaks accesses made by $\texttt{Token}_{\mathbf{w}_d^1,\mathbf{w}_1,D_j}$ and by $\texttt{Token}_{\mathbf{w}_d^1,\mathbf{w}_2,D_j}$. Likewise, q_2 leaks accesses make by $\texttt{Token}_{\mathbf{w}_d^2,\mathbf{w}_1,D_j}$ and by $\texttt{Token}_{\mathbf{w}_d^2,\mathbf{w}_2,D_j}$. In all cases, $j \in \{0,1,2,3\}$.
- **Query Result Pattern Leakage**: q_1 leaks $\texttt{sval}_{\mathbf{w}_d^1,D_3}$, while q_2 leaks $\texttt{sval}_{\mathbf{w}_d^2,D_3}$ to the server.

Hence, even though the same gate with same inputs is being evaluated, by unlinking the "special" term from the actual computation, the view of the server is *different* in both cases. $\mathcal{L}_{\text{SEC}}$ still contains these leakages (as with $\mathcal{L}_{\text{CSSE}}$), but the server can still not correlate multiple evaluations of the same gate. We formalize this idea in Theorem 2.

Reusability of SEC specific data structures. The unique selection of "special" terms by a client for each gate evaluation guarantees resistance of SEC to any statistical analysis of input bits by restricting the adversary's advantage of reverse-engineering inputs to negligible. This design choice also ensures that the same function invocation across different input sets has a non-identical access pattern. Concretely, for an adversary to correlate the encrypted input bits in two same function evaluations of the form $f_i(x, y)$ (where, x, y are all encrypted), it requires correlating the "special" term used for each query (evaluation). Let in this case the client chooses two different "special" terms for each gate evaluation, the corresponding conjunctive query translates to - $q_{f_i} = \mathbf{w}_d^1 \wedge \mathbf{w}_x \wedge \mathbf{w}_y$ for first evaluation and $q_{f_i'} = \mathbf{w}_d^2 \wedge \mathbf{w}_x \wedge \mathbf{w}_y$ for second evaluation (where, $\mathbf{w}_x, \mathbf{w}_y$ are keywords that map to the corresponding input bits). By the design of SEC, the access pattern leakage is dependent on the "special" term because the memory location of the documents corresponding to the "special" term is accessed and only those documents are fetched during a conjunctive search. In the example above the probability of a "special" term being repeated is upper bounded by $O(n!)$ (n is the number of gates in a circuit) because the client chooses unique permutation of "special" terms for every query (function evaluation). This is unlike the CSSE schemes where the "special" term is determined based on the least frequent keyword in the query and hence for the example above it would be the same for both functions. Since the frequency of all keywords in SEC is equal, we can leverage the unique selection of a "special" term thereby preventing an adversary from potentially guessing the encrypted input bits to a function by observing the memory access pattern. We re-iterate that both queries q_{f_i} and $q_{f_i'}$ evaluate the *same* function $f_i(x, y)$, hence the output of both evaluations will be equal. However, because of *permuting* the "special" term across two queries, the adversarial view of the server is indistinguishable from uniform for both queries. Thus, *using the same look-up table that is encrypted and offloaded to the server once during* SEC.KeyGen *phase,* SEC *can evaluate any arbitrary Boolean circuit multiple times.* This is guaranteed both in a single circuit across multiple gate evaluation as well as across multiple circuit evaluation. Hence, even though the generic SSE leakages still occur, nothing significant to the underlying computation is compromised. We provide a formal proof of reusability of lookup tables in Theorem 2.

Theorem 2 (Reusability of Lookup Table) *Given that* $\text{SEC}_{\mathsf{OXT}}$ *encrypts an input* $x_1, \ldots, x_p$ *using an IND-CPA secure symmetric-key encryption scheme to obtain corresponding encrypted bits* $c_1, \ldots, c_p$ *that is used as an input to a (publicly known) function* f*, and all information leaked from the underlying* CSSE.Search *(*OXT.Search*) phase is encapsulated by* $\mathcal{L}_{\mathsf{OXT}}^{\text{Search}}$*,* $\text{SEC}_{\mathsf{OXT}}$ *ensures reusability of the same look-up table for multiple (similar/different) gate evaluations without leaking any extra information than that encapsulated by* $\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}$*, while guaranteeing input and output data privacy from semantic security guarantees of an IND-CPA secure encryption scheme.*

Proof 2 *We prove Theorem 2 via a sequence of games between a challenger and an adversary, where the first game* (G_0) *is identical to the real experiment* $\mathbf{Real}_{\mathcal{A}}^{\text{SEC}}$ *and the final game (Simulator) is identical to the simulation experiment* $\mathbf{Ideal}_{\text{SIM},\mathcal{A}}^{\text{SEC}}$*. We establish*

formally that the view of the adversary $\mathcal{A}$ in each pair of consecutive experiments is computationally indistinguishable. For ease of exposition, we consider the client computes a binary function f on encrypted input bits $\{c_1, c_2\}$.

Game G_0. This game is identical to $\mathbf{Real}_{\mathcal{A}}^{\text{SEC}}$, where the challenger generates transcripts for the encrypted input bits $\{c_1, c_2\}$ by invoking SEC.Encrypt *and transcripts for the output $f(c_1, c_2)$ by invoking* SEC.Evaluate.

$$\Pr[G_0 = 1] \leq \Pr[\mathbf{Real}_{\mathcal{A}}^{\text{SEC}}(\lambda) = 1] - \mathsf{negl}(\lambda),$$

Game G_1. This game is identical to G_0 except for the fact that the challenger changes the encrypted input of function f to $\{c_3, c_4\}$. The evaluation of the function is done by invoking SEC.Evaluate *in the same way as done in G_0, i.e. lookup is performed on the same encrypted lookup table. The adversary cannot distinguish between G_0 and G_1 due to the use of different "special" terms for both queries.*

$$f(c_1, c_2) \xrightarrow{translated} \mathsf{OXT}.\textsc{Search}(\mathbf{EDB}, \{\mathbf{w}_d^1 \wedge \mathbf{w}_1 \wedge \mathbf{w}_2\})$$
$$f(c_3, c_4) \xrightarrow{translated} \mathsf{OXT}.\textsc{Search}(\mathbf{EDB}, \{\mathbf{w}_d^2 \wedge \mathbf{w}_3 \wedge \mathbf{w}_4\})$$

where, $\{\mathbf{w}_d^1, \mathbf{w}_d^2\}$ are randomly chosen "special" terms, and $\{\mathbf{w}_1, \ldots, \mathbf{w}_4\}$ are translated keywords from input bits $\{c_1, \ldots, c_4\}$. We say that by IND-CPA security guarantees the output of both G_0 and G_1 are indistinguishable from random.

$$|\Pr[G_1 = 1] - \Pr[G_0 = 1]| \leq \mathsf{negl}(\lambda),$$

Game G_2. This game is identical to G_1 except for the fact that the function being evaluated is changed to f' by the challenger. Since the search tokens generated depend on the "special" term and the function being evaluated, and since for every function evaluation the "special" term is chosen randomly by the client, for every function (same/different) evaluation the search tokens generated are different. The adversary can therefore not distinguish between two isomorphic gate evaluations.

$$|\Pr[G_2 = 1] - \Pr[G_1 = 1]| \leq \mathsf{negl}(\lambda),$$

Simulator. The simulator SIM *generates similar transcripts for* SEC.Evaluate *using the leakages from $\mathcal{L}_{\text{SEC}_{\mathsf{OXT}}}$ and this experiment is similar to $\mathbf{Ideal}_{\text{SIM},\mathcal{A}}^{\text{SEC}}$. How the transcripts are generated from each leakage component is discussed in detail above. We state here, that the* SIM *generates the transcripts from the corresponding leakages correctly, and the output of $\mathbf{Ideal}_{\text{SIM},\mathcal{A}}^{\text{SEC}}$ is indistinguishable from G_2.*

5.4 Statistical Analysis of Leakage Due to Reusability

We demonstrated a proof of computational indistinguishability (from an adversarial perspective) that establishes the secure reusability of SEC's lookup tables in Section 4. In this

section, we provide statistical analysis of the leakage from SEC's lookup tables to validate the same. For this, we closely follow the non-interference security notion well established in several side-channel analysis paradigms [4, 23].

Abstractly, the *non-interference property* ensures no sensitive information flow to the output of a system, given the system's inputs. In context of SEC, non-interference between inputs to SEC.Evaluate and $\mathcal{L}_{\text{SEC}}^{\text{Evaluate}}$ (i.e. the observable leakage) translates directly to the server's inability to infer (with statistical significance) anything about the inputs purely from $\mathcal{L}_{\text{SEC}}^{\text{Evaluate}}$. Concretely, non-interference can be defined as [4, 23]:

Definition 1 (Non-Interference) *For a probabilistic program* $\mathtt{P}$*, consider the set of secret ("high") inputs as* $\mathtt{H}$*, the set of public ("low") inputs as* $\mathtt{L}$*, and the output as* $\mathcal{L}$*. Then,* $\mathtt{P}$ *is said to be non-interfering if and only if the mutual information* $\mathcal{I}(\mathcal{L}; \mathtt{H} \mid \mathtt{L}) = 0$*.*

In other words, this information-theoretic definition captures the mutual information (or the mutual dependence) of $\mathcal{L}$ and $\mathtt{H}$ (or the critical, secret input to $\mathtt{P}$), given knowledge of non-secret $\mathtt{L}$. $\mathtt{P}$ is considered to be non-interfering if variations in $\mathtt{H}$ do not (statistically) affect $\mathcal{L}$ (given knowledge of $\mathtt{L}$). This is captured by the mutual information (conditioned on $\mathtt{L}$) being 0. However, estimating conditional mutual information in an information-theoretic setting is a difficult problem in general. Thus, a slightly "relaxed" definition for non-interference can be used instead [23]:

Definition 2 ("Relaxed" Non-Interference) *For a probabilistic program* $\mathtt{P}$*, consider the set of secret ("high") inputs as* $\mathtt{H}$*, the set of public ("low") inputs as* $\mathtt{L}$*, and the output as* $\mathcal{L}$*. Then,* $\mathtt{P}$ *is said to be non-interfering if the marginal distribution of* $\mathcal{L}$ *is independent of the distribution of* $\mathtt{H}$*.*

Concretely, from the point of view of reusability in SEC (as in Theorem 2), the random choice of "special" terms by the client leads to a computationally indistinguishable view of the server for repeated evaluations. From the perspective of non-interference definitions established, given a query executed in SEC.Evaluate, the "high" inputs $\mathtt{H}$ correspond to the actual tokens that map inputs to the function being evaluated, while the "low" input is the "special" term (that does not participate in actual functional evaluation). Evidently, $\mathcal{L}$ is then essentially the leakage observed from SEC's evaluate phase (i.e. $\mathcal{L}_{\text{SEC}}^{\text{Evaluate}}$).

To establish statistical independence between $\mathtt{H}$ and $\mathcal{L}$, we rely on Welch's t-test (that is naturally applied when two populations have unequal variances). In our experiments, we allow the server to learn memory access patterns wrt. the aforementioned leakage profile of SEC.Evaluate. The null hypothesis (for a two-tailed test) then tests whether the population means for two evaluations of SEC (while reusing SEC's data structures) are indistinguishable. We initialize the "special" terms during SEC.KeyGen as usual, and the client controls the permutations of the same over the execution of 1 million queries (i.e. a circuit consisting of 1 million gates). Our α value (i.e. the probability of incorrectly rejecting the null hypothesis when it is instead true) is 1%. Empirically, we observe a t-statistic of -1.7321 and a p-value of 0.0832. We hence conclude that there is not sufficient evidence to reject the null hypothesis. In other words, statistically, the server's view (given SEC.Evaluate's leakage profile) of reusable executions in SEC is statistically indistinguishable from the ex-

ecution of a randomly sampled circuit of the same size (for client-controlled permutation of "special" terms). This statistically establishes non-interference of $\mathtt{H}$ in the leakage $\mathcal{L}_{\mathrm{SEC}}^{\textsc{Evaluate}}$.

6 Experimental Results

In this section, we report on a prototype implementation of $\mathrm{SEC}_{\mathsf{OXT}}$ and $\mathrm{SEC}_{\textsc{ConjFilter}}$, and compare it with a prototype implementation of the TFHE library [22], which implements an efficient and fast gate-by-gate bootstrapping [21].

Implementation Details. Our prototype implementations are developed in C++ and we use 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 over the elliptic curve Curve25519 [7].

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 3.6, the storage required for SEC scales with the number of keyword-document pairs and the number of search tokens for a particular function. In our implementations, the server storage required for $\mathrm{SEC}_{\mathsf{OXT}}$ to store TSet[21] and XSet[22] and the $\mathsf{TokenSet}$ is around 43 KB while that for $\mathrm{SEC}_{\textsc{ConjFilter}}$ is 26 KB. We thus note that SEC is highly optimized and scalable with significantly fewer storage requirements than state-of-the- art FHE schemes. Table 7 offers a quantifiable comparison.

Table 7: Storage Overhead comparison (in MB) of SEC with existing FHE schemes in literature. Storage overhead of FHE scheme typically indicates the bootstrapping key size whereas for $\mathrm{SEC}_{\mathsf{OXT}}$ and $\mathrm{SEC}_{\textsc{ConjFilter}}$ it implies the size of the encrypted search index stored at the cloud server.

Scheme	Storage Overhead (in MB)
Gentry et. al[33]	3700
Gentry et. al[29]	2300
Halevi et. al.[40]	1600
Ducas et. al[26]	1000
Chillotti et. al.[21]	24
$\mathrm{SEC}_{\textsc{ConjFilter}}$ (**This work**)	0.449
$\mathrm{SEC}_{\mathsf{OXT}}$ (**This work**)	0.098

Evaluation of Computation time. The evaluation time of $\mathrm{SEC}_{\mathsf{OXT}}$ for computing arbitrary sized Boolean circuit over encrypted data scales linearly with the search time complexity of OXT times some constant which depends upon the depth of the circuit. The time required to retrieve the documents corresponding to a conjunctive query scales with the least frequent keyword in the query in OXT. Since the database (search index) in SEC is extremely small, the time taken by $\mathsf{OXT}.\textsc{Search}$ is significantly less. Hence, the average time required by $\mathrm{SEC}_{\mathsf{OXT}}$ is around 10 milliseconds for one binary function evaluation

[21] OXT specific data structure to store inverted index for the "special" term.
[22] OXT specific data structure to check presence of "cross" terms in respective document identifiers.

which is remarkably fast. On a similar note $\text{SEC}_{\text{ConjFilter}}$ scales with the search complexity of ConjFilter, which is dependent on the least frequent conjunct in the query. Notably $\text{SEC}_{\text{ConjFilter}}$ exhibits even faster performance with an average evaluation time of 40 microseconds for one binary function evaluation. Our experimental results validate that SEC is highly efficient and extremely fast while evaluating arbitrary Boolean functions over encrypted data. Figure 4 compares the execution time of SEC_{OXT} and $\text{SEC}_{\text{ConjFilter}}$ with different TFHE backends for varying depth of circuits.

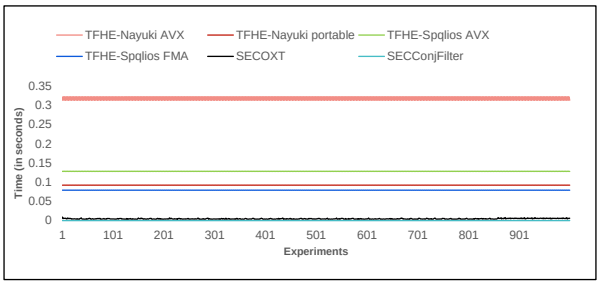

Figure 3: Time taken (in seconds) for 1000 invocations of SEC_{OXT} and $\text{SEC}_{\text{ConjFilter}}$ against different TFHE backends.

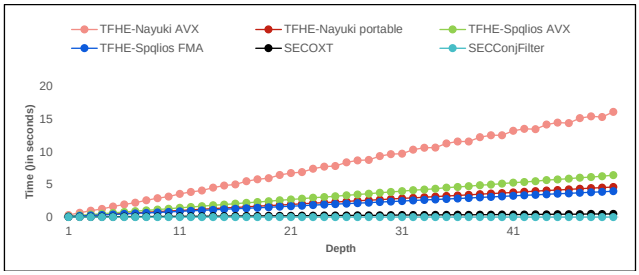

Figure 4: Time taken (in seconds) for different circuit depths of SEC_{OXT} and $\text{SEC}_{\text{ConjFilter}}$ against different TFHE backends.

Table 8: Time taken (in minutes) and Storage overhead (in MB) for evaluation of AES-128 circuit and Maxpool function (AlexNet) by SEC_{OXT} and $\text{SEC}_{\text{ConjFilter}}$ against different TFHE backends.

Scheme	Time (in minutes)		Storage (in MB)
	AES-128	Maxpool Function	
TFHE-Nayuki Portable	336.03	2920.52	24
TFHE-Nayuki AVX	179.02	1441.18	24
TFHE-Spqlios AVX	57.37	523.62	24
TFHE-Spqlios FMA	41.87	349.30	24
$\text{SEC}_{\text{ConjFilter}}$ **(This work)**	1.02	6.18	0.449
SEC_{OXT} **(This work)**	6.57	48.17	0.098

Comparison With FHE. We compare SEC_{OXT} and $\text{SEC}_{\text{ConjFilter}}$ with different variations of TFHE in Figure 3. 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 3 that $\mathsf{SEC}_{\mathsf{OXT}}$ is $10^3\times$ and six to seven times faster; $\mathsf{SEC}_{\textsc{ConjFilter}}$ is $10^6\times$ and $10^3\times$ faster than the portable TFHE backend and the fastest (non-portable) TFHE backend Spqlios AVX, respectively. Figure 4 compares the increase in execution time with an increase in the depth of the circuit. Both instantiations of SEC outperforms the fastest TFHE backend using Spqlios AVX optimization for function evaluation of arbitrary depth.

We showcase SEC's scalability for functions with multi-bit inputs by using it for encrypted evaluation of (i) the entire AES-128 circuit (with `XOR`/`AND`/`NOT`-gate count of 25124/6800/1692) and (ii) three max-pooling layers of AlexNet architecture[23] (a circuit with `OR`-gate count of 289060). This requires no extra storage (since we still only require storage for three extra gates), and the performance figures (as well as a comparison with Torus-FHE) are described in Table 8. For both circuits, a (non-parallelized) implementation of $\mathsf{SEC}_{\mathsf{OXT}}$ outperforms a (non-parallelized) implementation of Torus-FHE by six to seven orders of magnitude, while a (non-parallelized) implementation of $\mathsf{SEC}_{\textsc{ConjFilter}}$ shows an improvement of $10^3\times$ in computation time (we expect the relative comparison to remaining unchanged with parallelization and additional hardware/software-level optimizations). $\mathsf{SEC}_{\mathsf{OXT}}$ requires around $250\times$ less storage while $\mathsf{SEC}_{\textsc{ConjFilter}}$ requires $50\times$ times less storage, which are remarkably less. These results clearly showcase the efficiency and scalability of SEC for circuits with multi-bit inputs.

7 Discussion

We conclude with a brief discussion comparing SEC with traditional FHE. The core technical difference between SEC and FHE is as follows: SEC models each Boolean gate as a truth table, and leverages encrypted look-ups for evaluating this truth table on an encrypted input, while FHE models each Boolean gate as an algebraic operation over some appropriate algebraically structured mathematical object (e.g., polynomial rings [29, 11, 34] or the Torus [21, 22]), and exploits the algebraic structure underlying each encrypted input to evaluate the gate. This offers an efficiency vs functionality tradeoff. As demonstrated empirically, SEC outperforms traditional FHE schemes significantly, both in terms of computation time and storage requirements, when operating over symmetrically encrypted data. On the other hand, the algebraic structure underlying FHE allows it to operate over publicly encrypted data, and we leave it as an interesting open question to extend the lookup-based approach underlying SEC to computing over publicly encrypted data.

We note, however, that in many practical applications (e.g., querying over outsourced encrypted databases), it suffices to support evaluation of arbitrary Boolean circuits over symmetrically encrypted data, since the data owner is also the primary entity querying the (encrypted) data after outsourcing it to an untrusted server for storage and processing. Indeed, this setting motivates the entire literature on SSE [48, 25, 19, 16], albeit for restricted classes of functions. To the best of our knowledge, our work is the first to establish the possibility of supporting arbitrary Boolean circuit evaluation efficiently over encrypted data using purely symmetric-key encryption techniques on top of lookup-based gate evaluation.

[23] KSH17 Imagenet classification with deep convolutional neural networks

Indeed, as demonstrated by our theoretical analysis and practical evaluation, the usage of purely symmetric-key primitives is what enables the highly desirable efficiency and compactness guarantees of SEC, allowing it to scale over extremely large symmetrically encrypted datasets while outperforming FHE.

References

[1] Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., et al.: Openfhe: Open-source fully homomorphic encryption library. In: WAHC (2022)

[2] Alperin-Sheriff, J., Peikert, C.: 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 (2013)

[3] Alperin-Sheriff, J., Peikert, C.: 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 (2014)

[4] Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y.: Verified proofs of higher-order masking. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 457–485. Springer (2015)

[5] Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA (2012)

[6] Ben-Efraim, A., Lindell, Y., Omri, E.: Efficient scalable constant-round MPC via garbled circuits. In: Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Proceedings, Part II (2017)

[7] Bernstein, D.J.: Curve25519: New diffie-hellman speed records. In: Public Key Cryptography - PKC. Lecture Notes in Computer Science (2006)

[8] Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA (2020)

[9] Boemer, F., Kim, S., Seifu, G., DM de Souza, F., Gopal, V.: Intel hexl: accelerating homomorphic encryption with intel avx512-ifma52. In: WAHC (2021)

[10] Boneh, D., Sahai, A., Waters, B.: 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 (2011)

[11] Brakerski, Z., Gentry, C., Halevi, S.: Packed ciphertexts in lwe-based homomorphic encryption. In: Public-Key Cryptography–PKC 2013 (2013)

[12] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) (2014)

[13] Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011 (2011)

[14] Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) lwe. SIAM Journal on computing (2014)

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

[16] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO (2013)

[17] Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO 2013 (2013)

[18] Chang, Z., Xie, D., Li, F.: Oblivious ram: A dissection and experimental evaluation. Proceedings of the VLDB Endowment (2016)

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

[20] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: ASIACRYPT 2017 (2017)

[21] Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: ASIACRYPT 2016 (2016)

[22] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: Fast fully homomorphic encryption library (2019)

[23] Clark, D., Hunt, S., Malacaria, P.: Quantified interference: Information theory and information flow. In: Workshop on Issues in the Theory of Security (WITS'04) (2004)

[24] Clearinghouse., P.R.: Chronology of data breaches. `https://privacyrights.org/data-breaches` (2024)

[25] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM CCS (2006)

[26] Ducas, L., Micciancio, D.: Fhew: bootstrapping homomorphic encryption in less than a second. In: EUROCRYPT 2015 (2015)

[27] El-Yahyaoui, A., Kettani, M.D.E.E.: A verifiable fully homomorphic encryption scheme for cloud computing security. CoRR (2018)

[28] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing (2009)

[29] Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: EUROCRYPT 2011 (2011)

[30] Gentry, C., Halevi, S., Peikert, C., Smart, N.P.: Ring switching in bgv-style homomorphic encryption. In: SCN (2012)

[31] Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: PKC 2012 (2012)

[32] Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Eurocrypt (2012)

[33] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: CRYPTO 2012 (2012)

[34] Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I (2013)

[35] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) (1996)

[36] Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA. ACM (2013)

[37] Gorbunov, S., Vaikuntanathan, V., Wichs, D.: Leveled fully homomorphic signatures from standard lattices. In: Proceedings of the forty-seventh annual ACM symposium on Theory of computing (2015)

[38] Goyal, V., Li, H., Ostrovsky, R., Polychroniadou, A., Song, Y.: ATLAS: efficient and scalable MPC in the honest majority setting. In: Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, Proceedings, Part II (2021)

[39] Halevi, S., Polyakov, Y., Shoup, V.: An improved RNS variant of the BFV homomorphic encryption scheme. In: Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference (2019)

[40] Halevi, S., Shoup, V.: Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive (2020)

[41] Iliashenko, I., Zucca, V.: Faster homomorphic comparison operations for BGV and BFV. Proc. Priv. Enhancing Technol. (2021)

[42] Kamara, S., Wei, L.: Garbled circuits via structured encryption. In: Financial Cryptography and Data Security - FC 2013 Workshops, USEC and WAHC 2013, Okinawa, Japan (2013)

[43] Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A systematic evaluation of compact hardware implementations for the rijndael s-box. In: Topics in Cryptology–CT-RSA 2005 (2005)

[44] Okamoto, T., Takashima, K.: Fully secure functional encryption with general relations from the decisional linear assumption. In: Annual cryptology conference (2010)

[45] Patel, S., Persiano, G., Seo, J.Y., Yeo, K.: Efficient boolean search over encrypted data with reduced leakage. In: ASIACRYPT 2021 (2021)

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

[47] Smart, N.P.: Practical and efficient fhe-based MPC. In: Quaglia, E.A. (ed.) Cryptography and Coding - 19th IMA International Conference, IMACC 2023, London, UK, Proceedings (2023)

[48] Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceeding 2000 IEEE symposium on security and privacy. S&P 2000 (2000)

[49] Yang, K., Wang, X., Zhang, J.: More efficient MPC from improved triple generation and authenticated garbling. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, 2020 (2020)

[50] Yang, Q., Peng, G., Gasti, P., Balagani, K.S., Li, Y., Zhou, G.: MEG: memory and energy efficient garbled circuit evaluation on smartphones. IEEE Trans. Inf. Forensics Secur. (2019)

[51] Yao, A.C.C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986) (1986)

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

[53] Zhang, Y., Katz, J., Papamanthou, C.: Queries are belong to us: The power of file-injection attacks on searchable encryption (2016)

[54] Zhou, W., Jia, Y., Yao, Y., Zhu, L., Guan, L., Mao, Y., Liu, P., Zhang, Y.: 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)