Compressed Oblivious Encoding for Homomorphically Encrypted Search*

Seung Geol Choi
United States Naval Academy
choi@usna.edu

Dana Dachman-Soled
University of Maryland
danadach@ece.umd.edu

S. Dov Gordon
George Mason University
gordon.dov@gmail.com

Linsheng Liu
George Washington University
lls@gwu.edu

Arkady Yerukhimovich
George Washington University
arkady@gwu.edu

ABSTRACT

Fully homomorphic encryption (FHE) enables a simple, attractive framework for secure search. Compared to other secure search systems, no costly setup procedure is necessary; it is sufficient for the client merely to upload the encrypted database to the server. Confidentiality is provided because the server works only on the encrypted query and records. While the search functionality is enabled by the full homomorphism of the encryption scheme.

For this reason, researchers have been paying increasing attention to this problem. Since Akavia et al. (CCS 2018) presented a framework for secure search on FHE encrypted data and gave a working implementation called SPiRiT, several more efficient realizations have been proposed.

In this paper, we identify the main bottlenecks of this framework and show how to significantly improve the performance of FHE-base secure search. In particular,

- To retrieve ℓ matching items, the existing framework needs to repeat the protocol ℓ times sequentially. In our new framework, all matching items are retrieved in parallel in a *single protocol execution.*
- The most recent work by Wren et al. (CCS 2020) requires $O(n)$ multiplications to compute the first matching index. Our solution requires *no homomorphic multiplication*, instead using only additions and scalar multiplications to encode all matching indices.
- Our implementation and experiments show that to fetch 16 matching records, our system gives an 1800X speed-up over the state of the art in fetching the query results resulting in a 26X speed-up for the full search functionality.

KEYWORDS

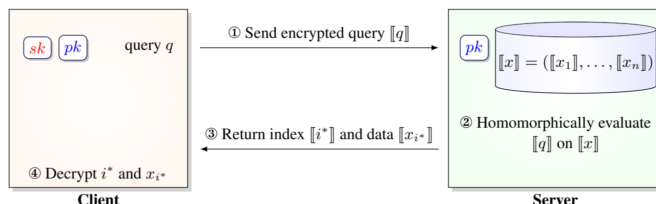
secure search; encrypted database; fully homomorphic encryption

1 INTRODUCTION

As computing paradigms are shifting to cloud-centric technologies, users of these technologies are increasingly concerned with the privacy and confidentiality of the data they upload to the cloud. Specifically, a *client* uploads data to the *server* and expects the following guarantees:

(1) The uploaded data should remain private, even from the server itself;

*A preliminary version of this paper will appear at ACM CCS '21. Authors are named alphabetically, and contributed equally.



In the above, $[\![\cdot]\!]$ denotes an FHE-encrypted ciphertext.

Figure 1: The secure search framework in [1]

(2) The server should be able to perform computations on the uploaded data in response to client queries;

(3) The client should be able to efficiently recover the results of the server's computation with minimal post-processing.

In this work, we will focus on the computational task of secure search. In this application, the client uploads a set of records to the server, and later posts queries to the server. Computation proceeds in two steps called *matching* and *fetching*. In the matching step, the server compares the encrypted search query from the client with all encrypted records in the database, and computes an encrypted 0/1 vector, with 1 indicating that the corresponding record satisfies the query. The fetching step returns all the 1-valued indexes and the corresponding records, to the client for decryption.

While seemingly conflicting goals, the guarantees of (1), (2), (3) can be simultaneously achieved for the secure search setting via techniques such as secure multiparty computation and searchable encryption. Recently, a line of works has focused on Fully Homomorphic Encryption (FHE)-based secure search, which we describe next.

FHE-based secure search. The simplicity of the framework of *secure search on FHE encrypted data* is attractive. Compared to other secure search systems, no costly setup procedure is necessary; it is sufficient for the client merely to upload the encrypted database to the server. Confidentiality is provided because the server works only on the encrypted query and records. The server can still perform the search correctly due to the powerful property of the full homomorphism of the underlying encryption scheme.

For this reason, researchers have been paying increasing attention to this problem. In particular, Akavia et al. [1] introduce a framework of performing secure search on FHE-encrypted data (see Figure 1).

Informally, a secure, homomorphic encrypted search scheme has the following Setup:

(1) (Setup) The client encrypts and uploads n items $x = (x_1, \ldots, x_n)$ to the server. Let $[\![x]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$. denote the encrypted data stored in the server.

Throughout the paper, we let $[\![\cdot]\!]$ denote an FHE-encrypted ciphertext. After the encrypted records have been uploaded, the client can perform a secure search using three algorithms, (Query, Match, Fetch).

(2) (Query) The client sends an encrypted query $[\![q]\!]$ to the server.
(3) (Match) The server homomorphically evaluates the query $[\![q]\!]$ on each record $[\![x_i]\!]$ to obtain the encrypted matching results $[\![b]\!] = ([\![b_1]\!], \ldots, [\![b_n]\!])$. That is, b_i is 1 if item x_i satisfies the given query q; otherwise, b_i is 0.
(4) (Fetch) Given $[\![b]\!]$, the server homomorphically computes $[\![i^*]\!]$, where $i^* = \min\{i \in [n] : b_i = 1\}$ which corresponds to the first matching record index. It fetches $[\![x_{i^*}]\!]$ (obliviously) and sends $([\![i^*]\!], [\![x_{i^*}]\!])$ to the client for decryption.

Multiplications in the fetching step. Akavia et al. also provide a construction that performs the fetching step in $O(n \log^2 n)$ homomorphic multiplications. Subsequently, more efficient algorithms have been presented with $O(n \log n)$ multiplications [2] and $O(n)$ multiplications [45].

1.1 Motivation

Bottleneck: fetching records sequentially. Suppose a client wants to fetch all matching items. Under the above framework, the client would first obtain the first matching index i^* and its corresponding item x_{i^*}. To fetch the second matching item, the framework suggests that the client should slightly change the original query q to a new query q'_{i^*} as follows:

- $q'_{i^*}(i, x_i)$ return true if $q(i, x_i)$ is true and $i > i^*$.

Then, by executing a new instance of the protocol with the encrypted query $[\![q'_{i^*}]\!]$, the client will obtain the second matching item. By repeating this procedure, the client will ultimately obtain all the matching records.

Note that the query q'_{i^*} embeds i^* in itself as a constant, which implies that there is no way for the client to construct this query q'_{i^*} without obtaining i^* first. In other words, the client can construct the query for the second matching item, *only after fetching the first matching item*. In this sense, the framework inherently limits the client to fetch only a single matching record at a time in a sequential manner.

If there are ℓ matching records, the client and server have to execute ℓ instances of the Query, Match, and Fetch algorithms. Since each Match and Search step requires costly homomorphic multiplications, the limitation of sequential protocol execution creates a serious bottleneck with respect to the running time. This leads us to ask the following natural question:

> *Is there a different secure search framework that allows the client to fetch all the matching records by executing a smaller number of protocol executions, possibly avoiding sequential record fetching?*

Reducing homomorphic multiplications. All previous schemes have to perform $\Omega(n)$ homomorphic multiplications in the fetching step. Since homomorphic multiplications are costly operations, it is desirable to reduce such computations, which begs the natural following question:

> *Can you reduce the number of homomorphic multiplications in the fetching step?*

In this paper, we answer both of the above questions affirmatively.

1.2 Our Work

Parallelizing the Fetch procedure. To address the issues, we introduce a new secure search framework where the matching items are retrieved *in parallel* in a constant number of rounds. Our Setup, Query and Match algorithms are the same as in prior work. However, we modify the Fetch procedure, dividing into two steps: Encode and Decode. In the Encode step, the server homomorphically inserts the matching items into a data structure - the particular structure depends on the construction, as we provide 3 different constructions, each using a different encoding. After receiving the encrypted encoding, the client decrypts the encoding and runs the Decode step to recover the items.

Compressed oblivious encoding. The encoding is computed homomorphically, and, most importantly, allows to encode *the full result set*, rather than just a single item. In particular, we introduce a notion of Compressed Oblivious Encoding (COE). A compressed oblivious encoding takes as input a large, but sparse, vector and compresses it to a much smaller encoding from which the non-zero entries of the original vector can be recovered. What makes this encoding *oblivious* is that the encoding procedure is performed on encrypted data. In certain constructions, the encoding includes the data values (CODE, compressed oblivious data encoding), and in others it only includes the indices (COIE, compressed oblivious index encoding). In the latter case, the Decode procedure is interactive, and allows the client to recover the values from the decoded set of indices.

For simplicity, when describing the generic syntax of secure search scheme, we denote the Encode procedure as taking both the indices and the values as input, and we suppress the fact that when the values are not used during Encoding, the Decoding step must be interactive. Recall, we use $[\![b]\!] = ([\![b_1]\!], \ldots, [\![b_n]\!])$ to denote the encrypted bit vector that results from the Match step.

(4) (Encode) Let $S = \{i \in [n] : b_i = 1\}$. Let $V = \{v_i : i \in S\}$. The server homomorphically evaluates an $[\![\mathsf{encoding}(S, V)]\!]$ and send it to the client.
(5) (Decode) The client decrypts $[\![\mathsf{encoding}(S)]\!]$ and runs the decoding procedure to recover (S, V).

We assume that the results set $|S|$ is small (i.e., sublinear in n). We would like the size of the compressed encoding to be sublinear in n to maintain meaningful communication cost.

No multiplications in the Encode step. To ensure minimal computational cost for encoding the results, we also wish to minimize the number of homomorphic multiplications. Recall, the best prior

	rounds	#Match	hmult	hadd	smult	communication	plaintext modulus
LEAF [45]	s	s	$O(ns)$	$O(ns \log n)$	0	$O(s \cdot \log n \cdot \lvert C \rvert)$	2
Protocol w/ BF-COIE	3	1	0	$O(n \log \frac{n}{s})$	0	$O(s^{1+\epsilon} \log \frac{n}{s} \cdot \lvert C \rvert + pir(s))$	prime
Protocol w/ PS-COIE	3	1	0	$n \cdot s$	$n \cdot s$	$O(s \cdot \lvert C \rvert + pir(s))$	prime
Protocol w/ BFS-CODE	2	1	n	$O(\lambda n)$	0	$O(s\lambda \cdot \lvert C \rvert)$	prime

- λ: statistical security parameter.
- n: number of uploaded encrypted records.
- s: number of matching records.
- ϵ: protocol parameter such that $0 < \epsilon < 1$.
- #Match: number of times the matching algorithm is executed.
- hmult: number of homomorphic multiplication operations used in the overall fetching step.
- hadd: number of homomorphic addition operations used in the overall fetching step.
- smult: number of scalar (plain) multiplication operations used in the overall fetching step.
- $|C|$: length of an FHE ciphertext.
- $pir(s)$: communication complexity required to retrieve s records via a PIR protocol.

Figure 2: Performance Comparisons when s records are fetched

work requires $O(n)$ multiplications by the server. Somewhat surprisingly, we demonstrate three encoding algorithms that can be evaluated *without* any homomorphic multiplications!

Using PIR (Private Information Retrieval). The asymptotic complexities and trade-offs of the search protocols are presented in Figure 2.

In some of our protocols (i.e., the search protocols with BF-COIE and PS-COIE; see Sections 4 and 6.3 for more detail), the indices and actual records are fetched in separate steps. This allows us to focus on optimizing the retrieval of the indices after which the values can be fetched using an efficient (setup-free) PIR protocol resulting in overall savings.

However, if reliance on PIR is undesirable, we also offer a variant that fetches the values directly (i.e., the protocol w/ BFS-CODE in Figure 2; see Sections 5 and 6.4 for more detail), as in prior work.

Implementation. We implement all of our proposed schemes and compare their performance with that of prior work. Our experiments show that our schemes outperform the fetching procedure of prior work by a factor of 1800X when fetching 16 records, which results in a 26X speedup for the full search functionality.

2 PRELIMINARIES

Let λ be the security parameter. For a vector a, let $\mathsf{nzx}(a)$ denote the set of all the positions i such that a_i is non-zero, i.e.,

$$\mathsf{nzx}(a) := \{i : a_i \neq 0\}.$$

Chernoff bound. We will use the following version of Chernoff bound.

THEOREM 2.1. *Let $X_1, \ldots, X_n$ be independent random variables taking values in $\{0,1\}$ such that $\Pr[X_i = 1] = p$. Let $\mu := \mathbf{Exp}[\sum X_i] = np$. Then for any $\delta > 0$, it holds*

$$\Pr\left[\sum_{i=1}^{n} X_i \geq (1+\delta)\mu\right] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}.$$

FHE. We use a standard CPA-secure (leveled) fully homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. We refer readers to [2, 45] for a formal definition. We use $[\![x]\!]$ to denote an encryption of x.

We also use + (resp. $\cdot$) to denote homomorphic addition (resp., multiplication). For example, $[\![c]\!] := [\![a]\!] + [\![b]\!]$ means that homomorphic addition of two FHE-ciphertexts $[\![a]\!]$ and $[\![b]\!]$ has been applied, which results in $[\![c]\!]$.

PIR. A PIR protocol allows the client to choose the index i and retrieve the ith record from one (or more) untrusted server(s) while hiding the index value i [18].

Assume that each of the k server has n records $D = (d_1, \ldots, d_n)$ where all items d_i have equal length. A single-round k-server PIR protocol consists of the following algorithms:

- The query algorithms $Q_j(i, r) \to q_j$ for each server $j \in [k]$, which are executed by the client with input index i and randomness r.
- The answer algorithms $A_j(D, q_j) \to a_j$ for each server $j \in [k]$, which is executed by the jth server.
- The reconstruction algorithm $R(i, r, (a_1, \ldots, a_k)) \to d_i$.

The communication complexity of a PIR protocol is defined by the sum of the all query lengths and answer lengths, i.e.,

$$\sum_{j \in [k]} |q_j| + |a_j|.$$

A PIR protocol is correct if for any $D = (d_1, \ldots, d_n)$ with $|d_1| = \cdots = |d_n|$, and for any $i \in [n]$, it holds that

$$\Pr_r\left[R\Big(i, r, \big\{A_j(D, Q_j(i, r))\big\}_{j=1}^{k}\Big) = d_i\right] = 1.$$

A PIR protocol is private if for any $j \in [k]$, for any $i_0, i_1 \in [n]$ with $i_0 \neq i_1$, the following distributions are computationally (or statistically) indistinguishable:

$$\{Q_j(i_0, r)\}_r \approx \{Q_j(i_1, r)\}_r.$$

2.1 Bloom Filter

A Bloom filter [9] is a well-known space-efficient data structure that allows a user to insert arbitrary keywords and later to check whether a certain keyword in the filter.

$\mathsf{BF.Init}()$. The filter B is essentially an ℓ-bit vector, where ℓ is a parameter, which is initialized with all zeros. The filter is also associated with a set of η different hash functions

$$\mathcal{H} = \{h_q : \{0,1\}^* \to [\ell]\}_{q=1}^{\eta}.$$

$\mathsf{BF.Insert}(\mathsf{B}, \alpha)$. To insert a keyword α, the hash results are added to the filter. In particular,

- For $q \in [\eta]$ do the following:
  Compute $j = h_q(\alpha)$ and set $B_j := 1$. Here B_j is the jth bit of B.

$\mathsf{BF.Check}(\mathsf{B}, \beta)$. To check whether a keyword β has been inserted to a BF filter B, one can just check the filter with all hash results. In particular,

- For $q \in [\eta]$ do the following:
  Compute $j = h_q(\beta)$ and check if B_j is set.
- If all checks pass output "yes". Otherwise, output "no".

The main advantage of the filter is that it guarantees there will be no false negatives and allows a tunable rate of false positives:

$$\left(1 - \left(1 - \frac{1}{\ell}\right)^{\eta s}\right)^{\eta} \approx \left(1 - e^{-\frac{\eta s}{\ell}}\right)^{\eta},$$

where s is the number of keywords in a Bloom filter.

Random oracle model for hash functions. We show our analysis in the random oracle model. That is, the hash functions are modelled as random functions.

2.2 Algebraic Bloom Filter

In this work, we leverage a variant of the Bloom filter where, when inserting an item, the bit-wise OR operation is replaced by addition. There have been works using a similar idea of having each cell hold an integer instead of holding a bit [21, 34].

Moreover, we consider a limited scenario *where the upperbound on the number of keywords to be inserted is known beforehand.* In particular, let s denote such an upperbound.

As before, the filter is also associated with a set of η different hash functions $\mathcal{H} = \{h_q : \{0,1\}^* \to [\ell]\}_{q=1}^{\eta}$. However, now the filter B is not an ℓ-bit vector but a vector where each element is in $[s\eta]$ (i.e., $B \in [s\eta]^{\ell}$) [1]. Therefore, the number of bits to encode B is now blown up by a multiplicative factor $\lceil \lg s\eta \rceil$.

The BF operations are described below where differences are marked by framed boxes.

$\mathsf{BF.Insert}(\mathsf{B}, \alpha)$. To insert a keyword α, the hash results are added to the filter. In particular,

- For $q \in [\eta]$ do the following:
  Compute $j = h_q(\alpha)$ and set $\boxed{B_j := B_j + 1}$.

[1] We can reduce $s\eta$ further to $\Theta(\eta \cdot (s/\ell) \cdot \log(s/\ell))$ using a Chernoff bound to bound the number of collisions contributing to the sum, but we will use $s\eta$ for the sake of simplicity of presentation.

$\mathsf{BF.Check}(\mathsf{B}, \beta)$. To check whether a keyword β has been inserted to a BF filter B, one can just check the filter with all hash results. In particular,

- For $q \in [\eta]$ do the following:
  Compute $j = h_q(\beta)$ and check if B_j is $\boxed{\text{greater than } 0}$.
- If all checks pass output "yes". Otherwise, output "no".

It is easy to see that this variant construction enjoys the same properties as the original BF construction.

3 COMPRESSED OBLIVIOUS ENCODING

As our main building block, we introduce a new tool we call Compressed Oblivious Encoding. A compressed oblivious encoding takes as input a large, but sparse, vector and compresses it to a much smaller encoding from which the non-zero entries of the original vector can be recovered. What makes this encoding *oblivious* is that the encoding procedure is oblivious to the original data; in fact, in our constructions the original data will all be encrypted. An efficient encoding must satisfy the following two performance requirements: 1) The size of the encoding must be sublinear in the size of the original array, and 2) constructing the encoding should be computationally cheap. Our constructions only use (homomorphic) addition and multiplication by constant (i.e. plaintext values).

A related notion is that of compaction over encrypted data [5, 8] which aims to put all non-zero entries of a vector to the front of the encoding. Our encoding can be viewed as a form of noisy compaction where, in addition to keeping all the non-zero entries, it allows a small number zero entries to be mixed in with the result. Thus, a compressed encoding trades some inaccuracy in the output for much cheaper construction costs.

We define two variants of compressed oblivious encodings, one that encodes the indices of non-zero entries and one that encodes the actual entries themselves.

3.1 Compressed Oblivious Index Encoding

A compressed oblivious index encoding (COIE) encodes the indices or locations of all the non-zero entries in the input array. We begin by defining the parameters and syntax for a COIE scheme.

Parameters. A COIE scheme is parametrized as follows.

- n: Input size – The dimension of the input vector v.
- s: Sparsity – Bound on the number on non-zero entries in v.
- c: Compactness – The dimension of the output encoding.
- f_p: False positives – The upperbound on the number of false positives returned by the decoding algorithm.

Syntax. A (n, s, c, f_p)-COIE scheme has the following syntax:

- $[\![\gamma_1]\!], \ldots, [\![\gamma_c]\!] \leftarrow \mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])$. The Encode algorithm takes as input a vector of ciphertexts with $v_i \in \{0,1\}$ for all $i \in [n]$. It outputs an encrypted encoding $[\![\gamma_1]\!], \ldots, [\![\gamma_c]\!]$.
- $I \leftarrow \mathsf{Decode}(\gamma_1, \ldots, \gamma_c)$. The Decode algorithm takes the encoding $(\gamma_1, \ldots, \gamma_c)$, in decrypted form, and outputs a set $I \subseteq [n]$

Correctness. Let $(\gamma_1, \ldots, \gamma_c) \leftarrow \mathsf{Dec}([\![\gamma_1]\!], \ldots, [\![\gamma_c]\!])$ denote a correct decryption of the encoding.

Definition 3.1. A (n, s, c, f_p)-COIE scheme is *correct*, if the following conditions are satisfied:

- (No false negatives) For all $v \in \{0,1\}^n$ with at most s non-zero positions, and for all $i \in \mathsf{nzx}(v)$, it should hold
$$i \in \mathsf{Decode}(\mathsf{Dec}(\mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])))$$
with probability at least $1 - \mathsf{negl}(\lambda)$ where the random coins are taken from Encode.
- (Few false positives) For all $v \in D^n$ with at most s non-zero positions, consider the set of false positives
$$E = \{i \in [n] : v_i = 0, \text{ but } i \in I\},$$
where $I = \mathsf{Decode}(\mathsf{Dec}(\mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])))$.
We require that $|E| \le f_p$ with the overwhelming probability over the randomness of Encode.

Efficiency. For efficiency, we look at the following three parameters of a COIE:

- The type and number of operations used by the Encode algorithm.
- The size of the encoding.
- The computation cost of the Decode algorithm.

For an efficient construction, we require that the latter two of these are sublinear in the size of the input vector.

3.2 Compressed Oblivious Data Encoding

A Compressed Oblivious Data Encoding (CODE) scheme is very similar to COIE except, rather than encoding the locations of non-zero entries, it encodes the values of these entries. We give a definition of CODE below where differences are marked by framed boxes.

Parameters. A CODE scheme is parametrized by the same four parameters (n, s, c, f_p) as a COIE.

Syntax. A (n, s, c, f_p)-CODE scheme over $\boxed{\text{domain } D}$ has the following syntax:

- $[\![\gamma_1]\!], \ldots, [\![\gamma_c]\!] \leftarrow \mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])$. The Encode algorithm takes as input a vector of ciphertexts with $v_i \in \boxed{D}$ for all $i \in [n]$. It outputs an encrypted encoding $[\![\gamma_1]\!], \ldots, [\![\gamma_c]\!]$.
- $\boxed{V} \leftarrow \mathsf{Decode}(\gamma_1, \ldots, \gamma_c)$. The Decode algorithm takes the encoding $(\gamma_1, \ldots, \gamma_c)$, in decrypted form, and outputs a set of values $\boxed{V = \{v_i : v_i \neq 0\}}$

Correctness.

Definition 3.2. A (n, s, c, f_p)-CODE scheme over domain D is correct, if the following conditions are satisfied:

- (No false negatives) For all $v \in \{0,1\}^n$ with at most s non-zero positions, and for all $i \in \mathsf{nzx}(v)$, it should hold
$$\boxed{v_i \in \mathsf{Decode}(\mathsf{Dec}(\mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])))}$$
with probability $1 - \mathsf{negl}(\lambda)$ where the random coins are taken from Encode.
- (Few false positives) For all $v \in D^n$ with at most s non-zero positions, consider the set of false-positive values
$$\boxed{E = \{z \in V : z \neq v_i \text{ for any } i \in \mathsf{nzx}(v)\}},$$
where $V = \mathsf{Decode}(\mathsf{Dec}(\mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])))$.
We require $|E| \le f_p$ with the overwhelming probability over the randomness of Encode.

4 COIE SCHEMES

We assume the input index vector $v \in \{0,1\}^n$ is sparse. In particular, throughout the paper, we assume $s = o(n)$.

4.1 A Warm-up construction

Using an algebraic BF, we can create an (n, s, c, f_p)-COIE scheme (the parameters c and f_p will be worked out after the description of the scheme).

$\mathsf{Encode}([\![v_1]\!], \ldots, [\![v_n]\!])$**.** The encoding algorithm works as follows:

(1) Initialize a BF $[\![B]\!] := ([\![B_1]\!], \ldots, [\![B_c]\!])$ with $B_j = 0$ for all j. Let $\mathcal{H} = \{h_q : \{0,1\}^* \to [c]\}_{q=1}^{\eta}$ be the associated hash functions.
(2) For $i = 1, \ldots, n$:
  (a) For $q = 1, \ldots, \eta$, do the following: Compute $j = h_q(i)$ and set $[\![B_j]\!] := [\![B_j]\!] + [\![v_i]\!]$.

Note that at step 2.a in the above, if $v_i = 0$, then B_j stays the same. On the other hand, if $v_i = 1$, then B_j will be increased by 1. This implies that B will exactly store the results of the operations $\{\mathsf{BF.Insert}(B, i) : i \in \mathsf{nzx}(v)\}$.

$\mathsf{Decode}(B_1, \ldots, B_c)$**.** Given the algebraic BF B, we can recover the indices for the nonzero elements as follows:

- Initialize I to be the empty set.
- For $i \in [n]$: if $\mathsf{BF.Check}(B, i) =$ "yes", add i to I.
- return I.

Parameters c and f_p. Since this is a warm-up construction, we perform only a rough estimation on the false positive parameter and the compactness parameter.

For reasons that will become clear later, we wish to keep the upper bound on the number of false positives (f_p) small. In particular, we use a BF with false-positive rate $1/n$. Since there are n operations of BF.Check, the expected number of false positives is 1, and from the Chernoff bound, the number of false positives is bounded by $\Omega(\log \lambda)$ with overwhelming probability in λ. This implies that we have $f_p = \Omega(\log \lambda)$.

The dimension c of the Bloom filter B can be computed using the following equation of BF false positive ratio:

$$\left(1 - e^{-\frac{\eta s}{c}}\right)^{\eta} \le \frac{1}{n},$$

Setting $c = \eta s \cdot n^{\frac{1}{\eta}}$ will satisfy the equation. This can be verified by using an equality $1 - e^{-x} \le x$ for $x \in [0, 1]$; that is, $1 - e^{-\frac{\eta s}{c}} \le \frac{\eta s}{c} = 1/n^{1/\eta}$.

Efficiency.

- The encoding algorithm uses $n\eta$ homomorphic addition operations, and $n\eta$ hash functions.
- The dimension c of the encoding is $\eta s \cdot n^{\frac{1}{\eta}}$. Usually, η is set to between 2 and 32.
- The decoding algorithm uses n operations of BF.Check.

In summary, we have reduced the encoding size c to be sublinear in n as desired. However, we still need to reduce the number

BF.Check operations in Decode to be sub-linear in n. We show how to achieve that in our next construction.

4.2 BF-COIE

We now show how to improve the above construction to achieve decoding in time $o(n)$. The main idea of this improvement is to use Bloom filters to represent a binary search tree, one BF per level of the tree. We can then guide the decoding algorithm to avoid decoding branches that do not contain non-zero entries. As most branches can be truncated well before reaching the leaf-level Bloom filter, this results in sublinear total cost.

Example. Before presenting the formal protocol for this construction we convey our idea through an example. Let $n = 32$, and suppose we wish to encode the indices $I = \{1, 15, 16\}$. Denote

$$I^k = \left\{ \left\lceil \frac{i}{2^k} \right\rceil : i \in I \right\}.$$

Intuitively, an element i in I^k can be thought of a range of length 2^k covering $[(i-1) \cdot 2^k + 1, i \cdot 2^k]$. We have:

- $I^4 = \{1\}$.
- $I^3 = \{1, 2\}$.
- $I^2 = \{1, 4\}$.
- $I^1 = \{1, 8\}$.
- $I^0 = \{1, 15, 16\}$.

Now, assume we insert each set I^k into its own BF. We can traverse these BF's to decode the set I as follows:

(1) Check I^4 for all possible indices. The only possible indices at this level are 1 and 2, since $n = 32$ and I^4 divides the original indices by $2^4 = 16$.
In the above example, When we query the BF for I^4, it only contains the index 1, which means that no values greater than 16 are contained in I. We can thus avoid checking any such indices at the lower levels.
Now consider the BF at the next level (i.e., the BF for I^3). The only possible values at this level are 1,2,3,4, but since we already know that there are no values greater than 16 in I, we only need to check for values 1, 2 (since $3 \cdot 8 > 16$).
(2) Check I^3 for indices 1, 2. The BF will show that indices 1 and 2 are both present, which means that we need to check indices 1, 2 and 3, 4 in I^2.
(3) Check I^2 for indices 1, 2, 3, 4. The BF will show that indices 1 and 4 are present, which means that we only need to check indices 1, 2 and 7, 8 in I^1, all other indices can be skipped.
(4) Check I^1 for indices 1, 2, 7, 8. The BF will show that indices 1 and 8 are present, which means that we need to check indices 1, 2 and 15, 16.
(5) Check I^0 for indices 1, 2, 15, 16, and output the final present indices 1, 15, 16.

Assuming, for now, that there are no false positives, observe that this approach checks at most $2 \cdot |I|$ values at each level, and there are $\lg n$ levels. Therefore, the decoding algorithm will check $O(|I| \cdot \lg n)$ indices, which is sub-linear in n.

BF-COIE. We now describe our BF-COIE construction. As before, we will work out the parameters after describing our construction. The encoding algorithm is described in Algorithm 1.

Algorithm 1 BF-COIE.Encode($[\![v_1]\!], \dots, [\![v_n]\!]$)

For simplicity, n and s are assumed to be powers of 2.
(1) $t := \lg \frac{n}{2s}$
(2) For $k = 0, \dots, t$:
  (a) Initialize $[\![B^k]\!] = ([\![B_1^k]\!], \dots, [\![B_\ell^k]\!]) := (\text{nil}, \dots, \text{nil})$.
  (b) Choose $\mathcal{H}^k = \{h_q^k : \{0,1\}^* \to [\ell]\}_{q=1}^{\eta}$ at random.
  (c) For $i \in [n]$ and for $q \in [\eta]$:
    $i' := \lceil i/2^k \rceil$, $j := h_q^k(i')$,
    If $[\![B_j^k]\!]$ is nil, then $[\![B_j^k]\!] := [\![v_{i'}]\!]$
    Otherwise, $[\![B_j^k]\!] := [\![B_j^k]\!] + [\![v_{i'}]\!]$
(3) Output $[\![B^0]\!], \dots, [\![B^t]\!]$.

Note that in steps (a) to (c) above, the warm-up construction is used to construct BF B^k for indices I^k.

In order to reduce the size of the output encoding, we set t to be $\lg \frac{n}{2s}$ instead of $\lg n$ as described previously. Note that when t is set in this way, I^t contains at most $n/2^t = 2s$ possible values thus maintaining our invariant.

The decoding algorithm is described in Algorithm 2.

Algorithm 2 BF-COIE.Decode($B^0, \dots, B^t$)

(1) Initialize $I, I^0, \dots, I^{t-1} := \emptyset$
(2) Initialize $I^t := \{1, \dots, n/2^t\} = [2s]$
(3) For $k = t, t-1, \dots, 1$, and for $i' \in I^k$:
  If BF.Check(B^k, i') is "yes", add $2i' - 1$, $2i'$ in I^{k-1}
(4) For $i \in I^0$:
  If BF.Check(B^0, i) is "yes", add i to I
(5) Output I

Useful lemma. The following lemma will be useful to analyze the parameters c and f_p.

Lemma 4.1. *Consider a Bloom filter with false positive rate $\frac{1}{m}$, where m is an arbitrary positive integer. Suppose at most m BF.Check operations are performed in the BF. Then, for any $\delta > 0$, we have:*

$$\Pr[\#\text{ false positives} \geq 1 + \delta] \leq \frac{e^\delta}{(1+\delta)^{(1+\delta)}}.$$

The proof, by an application of the Chernoff bound, can be found in Appendix A.

Regarding the above Lemma, we remark that setting $\delta = \Omega(\log \lambda)$, we have

$$\Pr\left[\sum_{i=1}^{m} X_i \geq 1 + \delta\right] = \text{negl}(\lambda).$$

Parameters c and f_p. We set the false positive upperbound $f_p := \Omega(\log \lambda)$ for the BF-COIE scheme. In our experiments, we set $f_p = 16$.

Now, let $m = \max(2s, s + 2f_p)$, we set the BF false positive rate to $1/m$. Recall that in the BF-COIE construction, the topmost BF B^t performs the BF.Check operation with $2s$ times; see line (2) in Algorithm 2. Using the above Lemma, the number of false positives in the top level BF B^t is at most f_p with all but negligible probability in λ. Furthermore, the index i in B^t is expanded into two indices

$2i-1$ and $2i$ in B^{t-1}. This means that the number of false indices to be checked in B^{t-1} due to the false positives in B^t is at most $2f_p$.

Now consider an index i that belongs to B^t. Algorithm 2 will run BF.Check on the values $2i-1$ and $2i$ in B^{t-1}. Since at least one of these values must actually belong to B^{t-1}, this leads to at most one false index being checked. Thus, the maximum number of false indices that would be checked in B^{t-1} is at most $s+2f_p$ (i.e., $2f_p$ from false positives of B^t and s from true positives of B^t).

The above argument applies inductively all the way to the bottom most level, which means that the maximum number of false indices that would be checked in each level BF B^i will be at most $s+2f_p$. In the end, the bottom BF will have at most f_p false positives, and the overall BF-COIE scheme will have at most f_p false positives with all but negligible probability in λ.

For the compactness parameter c, we must determine the dimension ℓ of each BF. Recall that we set the BF false positive rate to $1/m$ for $m = \max(2s, s+2f_p)$:

$$\left(1 - e^{-\frac{\eta s}{\ell}}\right)^{\eta} \leq \frac{1}{m}.$$

Setting $\ell = \eta \cdot s \cdot m^{\frac{1}{\eta}}$ would satisfy the above condition, which can be verified using an inequality $1 - e^{-x} \leq x$ for $x \in [0,1]$; that is, $1 - e^{-\frac{\eta s}{\ell}} \leq \frac{\eta s}{\ell} = (1/m)^{1/\eta}$.

Since the encoding has $t+1$ BFs, the overall compactness parameter is as follows:

$$c = (t+1) \cdot \ell = O\left(\eta \cdot s^{1+\frac{1}{\eta}} \cdot \lg \frac{n}{s}\right).$$

Efficiency.

- The size c of encoding is $O\left(\eta \cdot s^{1+\frac{1}{\eta}} \cdot \lg \frac{n}{s}\right)$. In our experiment, we choose $\eta = 2$.
- The encoding algorithm uses $O(\eta \cdot n \cdot \lg \frac{n}{s})$ homomorphic addition operations and hash functions.
- The decoding algorithm uses BF.Check operations for $O(s \lg \frac{n}{s})$ times.

In summary, assuming $s = o(n)$, we reduced the encoding size c to be sub-linear in n. Moreover, we also reduced the number BF.Check operations to be sub-linear in n.

Remark. Although this scheme has multiple BFs, the size of encoding c is smaller than that of the warm-up scheme! This is because with multiple levels of BFs, we can relax the false positive ratio for each BF. The encoding computation time was increased by a multiplicative factor of $\lg \frac{n}{s}$.

4.3 COIE Scheme Based on Power Sums

Removing false positives using power sums. We offer another encoding scheme using quite different techniques that can eliminate the false positives of the prior construction. To achieve this, we abandon Bloom filters, and instead use a power sum encoding, as has been done in several works using DC-Nets for anonymous broadcast [33, 39].

PS-COIE. We describe a COIE scheme based on power sums, which we call PS-COIE. As before, we will work out the parameters after describing our construction. The encoding algorithm is shown below.

Algorithm 3 PS-COIE.Encode($[\![v_1]\!], \ldots, [\![v_n]\!]$)

(1) For $j = 1, \ldots, s$:
  Compute $[\![w_j]\!] = \sum_{i=1}^{n} i^j \cdot [\![v_i]\!]$
(2) Output $[\![w_1]\!], \ldots, [\![w_s]\!]$.

Note that the values of i^j (modulo the underlying plaintext modulus) are publicly computable, so computing $i^j \cdot [\![v_i]\!]$ only requires scalar multiplication and no homomorphic multiplication.

Recall that $v_i \in \{0,1\}$. If we let $I = \{i : v_i = 1\}$ denote the indices of the nonzero elements, then note that

$$w_j = \sum_{i=1}^{n} i^j \cdot v_i = \sum_{i \in I} i^j.$$

Therefore, this w_j is the jth power sum of the indices. Using the power sums, we present the decoding algorithm in Algorithm 4.

Algorithm 4 PS-COIE.Decode($[\![w_1]\!], \ldots, [\![w_s]\!]$)

(1) Recall that we have $w_j = \sum_{x \in I} x^j$, for $j = 1, \ldots, s$, and we would like to reconstruct all x's in I.
(2) Let $f(x) = a_s x^s + a_{s-1} x^{s-1} + \cdots + a_1 x + a_0$ denote the polynomial whose roots are the indices in I.
(3) Use Newton's identities to compute the coefficients of this polynomial $f(x)$:

$$\begin{aligned} a_s &= 1 \\ a_{s-1} &= w_1 \\ a_{s-2} &= (a_{s-1} w_1 - w_2)/2 \\ a_{s-3} &= (a_{s-2} w_1 - a_{s-1} w_2 + w_3)/3 \\ &\vdots \\ a_0 &= (a_1 w_1 - a_2 w_2 + \cdots w_s)/s \end{aligned}$$

(4) Extract and output the roots of the polynomial $f(x)$.

Parameters c and f_p. This COIE scheme has no false positives; that is, $f_p = 0$. The compactness parameter c is equal to s.

Efficiency.

- The encoding algorithm uses $s \cdot n$ homomorphic addition operations and scalar multiplications[2].
- The encoding consists of s ciphertexts.
- The decoding algorithm computes coefficients in time $O(s^2)$. Roots of degree-s polynomial can be found in time $O(s^3 \log p)$, where p is the plaintext modulus of the underlying FHE, by using the Cantor–Zassenhaus algorithm [13].

5 CODE SCHEME

In the previous section, we showed two constructions of COIE schemes for encoding a vector of indices using sublinear storage. We now turn to the construction of CODE schemes, which, instead of encoding the indices of non-zero entries, encode the actual data values.

[2]We do not count the public multiplications to produce powers of i

Simplified key-value store. To construct our CODE scheme, we first construct an auxiliary data structure that supports the following operations:

- $\mathsf{Init}()$. Initialize the data structure.
- $\mathsf{Insert}(key, value)$. This operation allows the user to insert an item based on its key and value.
- $\mathsf{Values}()$. Returns all values that have been inserted thus far.

This data structure is simpler than a typical key-value store since it doesn't need to find an individual item by key. Note, however, that this is still sufficient to serve our purpose of constructing a CODE scheme.

5.1 BF Set

We now show how to instantiate a simplified key-value store using a data structure we call a Bloom filter set (BFS) that is in turn based on the algebraic Bloom filter presented in Section 2.2. To insert a pair $(key, value)$, the Bloom filter set stores the actual *value* rather than an indicator bit. Items are inserted similar to before, by adding their value to the locations indicated by the hashes of the *key*.

Input data format. For our construction we make an assumption on the format of the inserted data. Specifically, we assume that all inserted values contain a unique checksum (e.g., a cryptographic hash of the value). We assume that this checksum is sufficiently long that a random sum of checksums does not give a valid checksum except with negligible probability (as a function of λ).

Construction. We first describe the construction of the data structure. We show below how to choose parameters in such a way that the client can extract all the matched items from this Bloom filter, with overwhelming probability.

- $\mathsf{BFS.Init}() \rightarrow (B, \mathcal{H})$. Create an ℓ-dimensional vector B where each element can store any possible value in the domain D. Choose a set of η different hash functions $\mathcal{H} = \{h_q : \{0,1\}^* \rightarrow [\ell]\}_{q=1}^{\eta}$. Initialize $B_i := 0$ for $i \in [\ell]$.
- $\mathsf{BFS.Insert}(B, \mathcal{H}, key, \alpha)$. To add (key, α), we add α to the values stored at the locations indicated by the hashes of key. Specifically,
  - For $q \in [\eta]$:
    Compute $j = h_q(key)$ and set $B_j := B_j + \alpha$.
- $\mathsf{BFS.Values}(B)$. Initialize a set V to be the empty set. For $j \in [\ell]$, if B_j has a valid checksum, add B_j to V. Finally, output V.

We note that, as previously proposed by Goodrich [25], it is possible to avoid the checksum by maintaining a counter of the number of values inserted for each location. Then, BFS.Values only returns values at locations with a counter of 1.

Parameters. We show how to set the Bloom filter parameters to guarantee that all values can be recovered with all but negligible probability. We assume that we know the upper bound s on the number of inserted values. We prove the following lemma.

Lemma 5.1. *If at most s values have been inserted in the* BFS *data structure, then by setting η and ℓ such that*

$$\ell \geq 2(s\eta - 1),$$

we can recover all s values with probability at least $1 - s \cdot (1/2)^{\eta}$.

Proof. Consider a (key, value) pair (k_i, α_i). We say that this pair has a *total collision* if every hash position for the pair is also occupied by another inserted key, value pair. In this case, α_i cannot be recovered. On the other hand, if at least one hash position has no collisions, then we can recover the value. Note that the collision depends on the key k_i but not the value α_i.

For a given key k_i, we define the event $\mathsf{TCOL}(k_i)$:

$$\mathsf{TCOL}(k_i) = 1 \text{ if } \forall q \in [\eta], \exists (k', q') \neq (k_i, q) : h_q(k_i) = h_{q'}(k').$$

Here, k' can be the key of any item that has been inserted in the set. Since the set contains at most s items, there are at most s possible keys for k'. Recall also that η hash functions are applied for each item.

Since for each k_i, there are at most $\eta s - 1$ pairs of (k', q')s that are different from (k_i, q), we can bound the collision probability as follows:

$$\Pr[\mathsf{TCOL}(k_i)] \leq \left(\frac{(\eta s - 1)}{\ell}\right)^{\eta}$$

Thus, if we choose η and ℓ such that $\ell \geq 2(s\eta - 1)$, we have

$$\Pr[\mathsf{TCOL}(k_i)] \leq (1/2)^{\eta}$$

Taking a union bound over all s inserted values, we have

$$\Pr[\exists k_i : \mathsf{TCOL}(k_i)] \leq s \cdot (1/2)^{\eta}$$

. □

5.2 CODE Scheme Based on BF Set

In this section, we construct a CODE scheme. Recall that unlike encoding the indices through a COIE scheme, a CODE scheme encodes data in a compressed manner. The main idea of our construction is simulating the operations of BFS; we call our scheme BFS-CODE.

Pre-processing the input data. As mentioned in the description of the BF Set construction, we need to pre-process the input data so that each item is attached with its checksum. Although a data item v is represented as a single number, it is assumed that v can be parsed as $v.val$ for its actual value and $v.tag$ for its checksum. Moreover, we assume that the checksum is long enough, such that a random linear combination of checksums is only negligibly likely to produce a valid checksum (i.e., $|checksum| = \omega(\lambda)$).

We stress that when our CODE scheme is used for secure search, this pre-processing can be performed locally by the client prior to encrypting his data. Moreover, computing checksum adds only a tiny amount of overhead.

BFS-CODE. We now describe our (n, s, c, f_p)-BFS-CODE construction over domain D. As before, we will work out the parameters after describing our construction. The encoding algorithm is shown below.

Note that at step 4 in the above, if v_i is 0, then B_j stays the same. On the other hand, if v_i is not 0, B_j will be increased by v_i. This implies that B will exactly hold the result of operations $\{\mathsf{BFS.Insert}(B, \mathcal{H}, i, v_i) : i \in \mathsf{nzx}(v)\}$.

The decoding algorithm is simple, and it's described in Algorithm 6.

Correctness. This is immediate from the additive homomorphism of the underlying encryption scheme and the parameters for the

Algorithm 5 BFS-CODE.Encode($[\![v_1]\!], \ldots, [\![v_n]\!]$)

(1) $\eta = \lambda + \lg s$; $\ell = 2(\eta s - 1)$
(2) Initialize $[\![B]\!] = ([\![B_1]\!], \ldots, [\![B_\ell]\!]) := ([\![0]\!], \ldots, [\![0]\!])$.
(3) Choose $\mathcal{H} = \{h_q : \{0,1\}^* \to [\ell]\}_{q=1}^{\eta}$ at random.
(4) For $i \in [n]$ and for $q \in [\eta]$:
    $j := h_q(i)$; $[\![B_j]\!] = [\![B_j]\!] + [\![v_i]\!]$
(5) Output $[\![B]\!]$.

Algorithm 6 BFS-CODE.Decode(B)

(1) Output BFS.Values(B)

BFS. In particular, we set $\eta = \lambda + \lg s$ so that the probability of recovery error is at most 2^{λ}.

Parameters c and f_p. The checksums attached to the data items ensure that we have no false positives with overwhelming probability, that is, $f_p = 0$. The compactness parameter c is the dimension ℓ of the BF, which is $O(\eta s)$.

Efficiency.

- The encoding algorithm uses $\ell = O(\eta s)$ encryption operations, $\eta \cdot n$. addition operations, and ηn hash functions.
- The encoding consists of ℓ ciphertexts.
- The decoding algorithm uses ℓ decryption operations.

Since by Lemma 5.1, the size ℓ of the Bloom filter only depends on the number of matches s and the number of hash function η, we get that the communication complexity of the above protocol is independent of the database size n.

6 SECURE SEARCH PROTOCOLS

We implement secure search protocols by using compressed oblivious encoding schemes. We begin by defining a relaxed notion of correctness that allows for false positives, as is needed in some of our constructions. we then define security of secure search.

6.1 (ℓ, f_p)-Relaxed Secure Search

We relax the correctness guarantee to allow the Client to retrieve a superset of the matching records. Specifically, if $\mathcal{S}$ is the set of indexes matching a Client's query q, then at the end of the protocol, we require the Client to obtain a set $\mathcal{S}'$ such that:

- With all but negligible probability, $\mathcal{S} \subseteq \mathcal{S}'$
- With all but negligible probability, $|\mathcal{S}' \setminus \mathcal{S}| \leq f_p$.

We parameterize a secure search scheme by (ℓ, f_p), where ℓ is the *amortized* communication complexity *per* matching record, and f_p is the number of "false positives," as defined above.

6.2 Security of Setup-free Secure Search

To define security of our secure search schemes, we use a game-based security definition similar to that of Akavia et al. [2]. The game is between a challenger and an adversary $\mathcal{A}$ with regard to a setup-free search scheme, sec-search, and an FHE scheme, FHE.

$\mathsf{Game}_{\mathsf{FHE}}^{\mathsf{sec\text{-}search}}(\mathcal{A})$:

(1) The challenger runs a key generation algorithm (with computational security parameter κ) and sends the evaluation key to $\mathcal{A}$ so that $\mathcal{A}$ can perform homomorphic additions and multiplications.
(2) $\mathcal{A}$ chooses either:
    - Two databases $x^0 = (x_1^0, \ldots, x_n^0)$ and $x^1 = (x_1^1, \ldots, x_n^1)$ of the same length, and a query q, or
    - A single database $x = (x_1, \ldots, x_n)$ and two queries q^0, q^1 of the same circuit size.

    In both cases, we require that the sizes of the two result sets (denoted by s) are equal.
(3) The challenger samples $b \leftarrow \{0,1\}$. Then, either
    - Runs Setup on input x^b and the search protocol from sec-search on input q, or
    - Runs Setup on input x, and the search protocol from sec-search on input q^b.
(4) $\mathcal{A}$ outputs a bit b'
(5) We say that $\mathcal{A}$ has advantage

$$\mathsf{Adv}_{\mathsf{FHE}}^{\mathsf{sec\text{-}search}}(\mathcal{A}) = |\Pr[b = b'] - 1/2|.$$

Definition 6.1. A setup-free (ℓ, f_p)-secure search scheme sec-search is *fully secure* if every PPT adversary $\mathcal{A}$ controlling the server has a negligible advantage $\mathsf{Adv}_{\mathsf{FHE}}^{\mathsf{sec\text{-}search}}(\mathcal{A}) \leq \mathsf{negl}(\kappa)$ in the game above.

6.3 From COIE to Secure Search

We next present our framework for obtaining Secure Search from COIE. The intuition is likely already clear from the previous descriptions: the encrypted client query is applied to the dataset, returning an encrypted bit vector indicating where index matches lie. The server homomorphically computes the hamming weight of this vector, and sends it to the client for decryption. This provides the result set size to the Server, allowing it to encode the result vector in the COIE.[3] The encoding is sent to the client for decryption and decoding.

Because the COIE only encodes the indices, and not the data values, we then add a PIR step to fetch the corresponding data. Note that if the COIE scheme admits false positives, it is possible that the number of false positives, and therefore the number of PIR queries, depends on the data, leaking something to the Server. To fix this problem, the client pads the number of PIR queries as follows. It fixes a bound f_p on the number of false positives, and aborts if the actual number of false positives exceeds this bound. Otherwise, the client uses enough dummy queries to pad the number of PIR queries to $s + f_p$.

Theorem 6.2. *Given an FHE scheme, a (n, s, c, f_p)-COIE scheme in the random oracle model, and a PIR scheme in the random oracle model with communication complexity ℓ_p for records in $\{0,1\}^m$, the construction in Algorithm 7 yields a (ℓ, f_p)-secure search scheme for records in $\{0,1\}^m$ in the Random Oracle Model, where $\ell = \frac{c \cdot \ell_c + (s+f_p) \cdot \ell_p}{s}$, ℓ_c is the length of an FHE ciphertext, and s is the number of matching records.*

Proof. We begin by proving that the adversary cannot distinguish between two different queries. The adversary chooses

[3] We note if we don't wish to reveal this to the server, we can use a fixed, global upper bound, or, if it is appropriate to the application, the client can add noise to provide differential privacy. It is also worth pointing out that prior work leaks the result set size as well.

Algorithm 7 Secure search with a (n, s, c, f_p)-COIE scheme.

(1) Client runs the FHE key generation algorithm and encrypts database $x = (x_1, \ldots, x_n)$ with $x_i \in \{0,1\}^m$. It then sends $[\![x]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$ and the evaluation key to Server.
(2) Client sends an encrypted query $[\![q]\!]$.
(3) Server homomorphically evaluates the encrypted query $[\![q]\!]$ on each encrypted record. In particular, let $[\![b]\!] = ([\![b_1]\!], \ldots, [\![b_n]\!])$ where $[\![b_i]\!] = [\![q(x_i)]\!]$. Note that $q(x_i) = 1$ if record i is a match and is equal to 0 otherwise.
(4) Server homomorphically computes $[\![s]\!] = \sum_{i=1}^{n} [\![b_i]\!]$, and sends to Client for decryption.
(5) Client decrypts $[\![s]\!]$ to obtain s, and sends s to Server.
(6) Server calls COIE.Encode($[\![b]\!]$) with sparsity parameter s, to obtain an encrypted encoding $[\![C]\!]$. It sends $[\![C]\!]$ to Client.
(7) Client decrypts $[\![C]\!]$ into C and calls COIE.Decode(C) to obtain a set $\mathcal{S}'$ of size $s+e$ indexes. If $e > f_p$, Client aborts. Otherwise, Client adds $f_p - e$ number of dummy indexes to $\mathcal{S}'$.
(8) Client runs a PIR protocol with the Server to obtain the records corresponding to the indexes in $\mathcal{S}'$.

a database x and two queries q^0 and q^1, with the promise that $s = \sum_{i=1}^{n} q^0(x_i) = \sum_{i=1}^{n} q^1(x_i)$.

The entire view of the adversary during the experiment can be reconstructed efficiently given (1) the encrypted database $[\![x]\!]$ (2) the encrypted query $[\![q]\!]$, (3) $s + f_p$ iterations of the PIR protocol, requesting indexes in $\mathcal{S}'_b$, where s is the number of matching records.

Since the value of s is the same for q^0 and q^1, the two things that change in the view of the adversary when switching from $b = 0$ to $b = 1$ are (1) the encrypted query $[\![q^b]\!]$ (2) the set of indexes $\mathcal{S}'_b$ (but not the number) requested during the PIR step.

We also note that the experiment only aborts when the number of received false positives e is greater than the bound f_p, which only happened with probability $\mathsf{negl}(\lambda)$ for a statistical security parameter λ. Thus, we ignore this possibility in the following.

We can now proceed via a standard hybrid argument:

- We first consider the real experiment with $b = 0$.
- We then switch the encrypted query from q^0 to q^1, but leave the set of indexes in the PIR step as $\mathcal{S}'_0$. Indistinguishability of the adversary's view follows from the IND-CPA security of the FHE scheme.
- Next, we switch the set of indexes in the PIR step from $\mathcal{S}'_0$ to $\mathcal{S}'_1$. Indistinguishability of the adversary's view now follows from the security of the PIR scheme. This is now identical to the real experiment with $b = 1$.

We conclude that the probability the adversary outputs 0 or 1 differs by a negligible amount when $b = 0$ versus $b = 1$. Therefore, the advantage of the adversary in guessing b is negligible.

The proof that the adversary cannot distinguish between the same query applied to two different databases follows nearly identically. □

6.4 From CODE to Secure Search

We next present our framework for obtaining Secure Search from CODE.

Algorithm 8 Secure search with a (n, s, c, f_p)-CODE scheme.

(1) Client runs the FHE key generation algorithm and encrypts database $x = (x_1, \ldots, x_n)$ with $x_i \in D$. It then sends $[\![x]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$ and the evaluation key to Server.
(2) Client sends an encrypted query $[\![q]\!]$.
(3) Server homomorphically evaluates the encrypted query $[\![q]\!]$ on each encrypted record. In particular, let $[\![b]\!] = ([\![b_i]\!], \ldots, [\![b_n]\!])$ where $[\![b_i]\!] = [\![q(x_i)]\!]$. Note that $q(x_i) = 1$ if record i is a match and is equal to 0 otherwise.
(4) Server homomorphically computes $[\![s]\!] = \sum_{i=1}^{n} [\![b_i]\!]$ and sends $[\![s]\!]$ to Client.
(5) Client decrypts $[\![s]\!]$ to obtain s and sends it back to the Server.
(6) Server computes $[\![d_i]\!] = [\![b_i]\!] \cdot [\![x_i]\!]$ for $i \in [n]$. Then, it applies CODE.Encode($[\![d_1]\!], \ldots [\![d_n]\!]$) with sparsity parameter s, to obtain an encrypted encoding $[\![C]\!]$. It sends $[\![C]\!]$ to Client.
(7) Client decrypts $[\![C]\!]$ to C and decodes C to obtain a set $\mathcal{S}$ of size s matching records.

Theorem 6.3. *Given an FHE scheme, and a (n, s, c, f_p)-CODE scheme over domain D in the random oracle model, the construction in Algorithm 8 yields a (ℓ, f_p)-secure search scheme for records in domain D in the random oracle model, where $\ell = \frac{c(s) \cdot \ell_c}{s}$, ℓ_c is the length of an FHE ciphertext with plaintext space D, and s is the number of matching records.*

The proof is similar to the COIE-based scheme and can be found in Appendix B.

On the use of homomorphic multiplication. As described, our CODE-based search scheme uses n homomorphic multiplications to create the vector $[\![d]\!]$. However, it may be the case that this vector is already produced as part of the match step, for example for arithmetic queries. In this case, our CODE scheme requires no further homomorphic multiplications.

On volume attacks. In our secure search schemes, the client sends the number s of matching records to the server so that the server can create an oblivious compress encoding. One recent line of works has developed attacks using volume leakage (e.g., [7, 28, 31]), and these types of attacks can be applied to our scheme in theory.

In our scheme, the volume attacks can be mitigated by hiding s in a differentially private manner. In particular, the client can add a small amount of noise to s before sending it to the server. A similar approach was used in previous work e.g., [37].

7 EVALUATION

7.1 Fetch time

We implemented our search protocols based on BF-COIE, PS-COIE, and BFS-CODE schemes. All protocols were implemented using PySEAL [43], which is a Python wrapper of the Microsoft research SEAL library (version 3.6) [40] using the BFV encryption scheme [20]. We instantiated a single-server PIR protocol in our construction using SealPIR [4]. For the root finding step of the decoding procedure in PS-COIE, we use an implementation based on SageMath 9.2 [42].

Measuring the Fetch step. Our search framework improves the overall search time by executing the Match step only once, while the LEAF protocol must execute the Match step s times. However,

since we do not optimize the Match step itself over prior work, we focus on measuring the cost of the Fetch procedure. That is, our experiments measure the time from when the server holds encrypted query results, i.e., $([\![b_1]\!], \ldots, [\![b_n]\!])$ with $b_i \in \{0, 1\}$, to when the client recovers all s records matching the query. Specifically, we measure the cost of steps 4 and up in Algorithms 7 and 8. Similarly, for LEAF+, we only measure the cost of the Fetch step.

Database. To measure the performance of our protocols, we run experiments with database size n ranging from 1000 to 100,000 data items and the result set size s set to between 8 and 128. As in the LEAF+ experiments [45], all data items are 16-bit integers.

BF-COIE parameters. For the BF-COIE secure search, we set the parameters as indicated in Section 4.2.

- We set the false positive upperbound $f_p = 16$. Recall that the client aborts (without executing the PIR) if the actual number of false positives exceeds this, but this only happens with probability negligible in the security parameter, which we set $\lambda = 40$.
- We set the number of hash function $\eta = 2$ for each Bloom filter, so each BF has size $\ell = 2s \cdot \sqrt{2s}$. (If $2s < s + 2f_p$, we set $\ell = 2s \cdot \sqrt{s + 2f_p}$).

BFS-CODE parameters. For BFS-CODE secure search, with $\lambda = 40$, the number of hash functions η is set to $\lambda + \lg s$, and the Bloom filter size is set to $2(\eta s - 1)$. Additionally, each data item is attached with a 40-bit checksum to guarantee a $2^{-\lambda}$ probability of collision. We used SHA2 to compute a checksum.

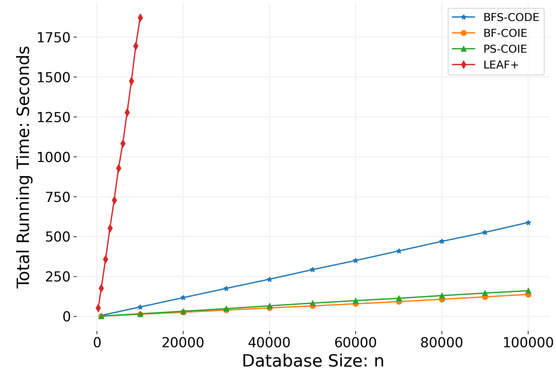
Implementing LEAF+. For a comparison we also implemented the fetch step of the LEAF+ protocol [45], since their implementation is not publicly available.

Their protocol has $O(\log \log n)$ depth of multiplications. Therefore, they have to use bootstrapping techniques to reduce the accumulated noise. However, SEAL doesn't provide a method for bootstrapping, and we suspect that they added a customized implementation of bootstrapping on top of SEAL. Unfortunately, their implementation is not available.

We address this issue by choosing to ignore the time for bootstrapping when we measure the running time of our implementation of LEAF+. Of course, our implementation doesn't output the correct results, but the measured running time will be shorter than the actual running time. Therefore, we believe that this measured time serves as a good baseline.

Experiment environments. All our experiments were performed on an Intel®Core 9900k @4.7GHz with 64GB of memory. For fair comparison, the test was performed on a single thread with no batching optimizations for computation. Networking protocol between server and clients is a 1Gbps LAN.

Results: Fetch time vs. database size. Figure 3 shows the performance of our protocols as a function of database size, while the result set size s is fixed to 16. However, for LEAF+, we plot the time for fetching only a **single** record, since fetching s records takes too long. In our implementation of LEAF+, fetching even a single record when $n = 10,000$ requires 1872 seconds. We note that the authors of LEAF+ report about 60 seconds for a single fetch [45].



For LEAF+, we plot the time for fetching only a **single** record, since fetching s records takes too long.

Figure 3: Fetch time vs. Database size with $s = 16$.

We conjecture that they parallelize the scheme with 32 threads. Here, we only use a single thread.

All three of our protocols greatly outperform LEAF+. Looking at BF-COIE in particular:

- In BF-COIE search, fetching 16 records with $n = 10,000$ takes 16.7 seconds, compared to 1872 seconds for a single record fetch in LEAF+. We believe that the speed up is due to the fact that LEAF+ (with a single-record fetching) needs $O(n \log n)$ homomorphic additions and $O(n)$ homomorphic multiplications, while BF-COIE search needs only $O(n \log \frac{n}{s})$ homomorphic additions with *no homomorphic multiplications*. In addition, as Figure 4 shows, the overhead of the PIR step to retrieve the actual data is small.
- Due to the sequential limitation in LEAF+, fetching 16 records with LEAF+ is extrapolated to take about $16 \cdot 1872 = 29952$ seconds. Overall, BF-COIE search is about **1800 times faster** than LEAF+.

The time for all three of our protocols is dominated by the server's computation during encode, which grows linearly with the DB size.

Since the number of hash functions η is larger in the BFS-CODE protocol than in BF-COIE protocol, the encoding step of this protocol takes longer.

Results: fetch time vs. the result set size. Figure 4 shows the performance of our protocols as a function of the result set size s while n is fixed to $10,000$. Here, again the performance is dominated by the encoding step, but the relative costs have changed. Due to the need to compute more power sums, the PS-COIE protocol performs worse than BS-COIE and BFS-CODE when s becomes moderately large.

The time used for transmitting the data over network (green in Figure 4) increases for larger s. However, it still remains small for all three schemes. In the scenario of having lower network bandwidth, batching is recommended to pack a vector of ciphertexts into a

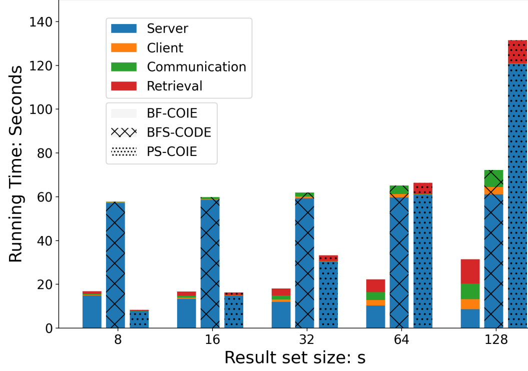


Figure 4: Fetch time vs. Result set size with $n = 10,000$.

single ciphertext with relatively low computation overhead. We discuss communication costs further in Section 7.3.

7.2 Overall Running Time

Although we do not optimize the Match step itself over prior work, we provide an estimated comparison of the running time for the end-to-end flow.

Our search framework improves the overall search time by executing the Match step only once, while the LEAF protocol must execute the Match step s times. Based on this, we can extrapolate the running time as follows:

- The overall running time for LEAF:

$$Time(\mathsf{LEAF}) = s \cdot \mathsf{MT}(\mathsf{LEAF}) + s \cdot \mathsf{FT}(\mathsf{LEAF}).$$

  Here, MT and FT denote the match time and fetch time respectively.
- The overall running time for the BF-COIE scheme:

$$Time(\mathsf{BF\text{-}COIE}) = \mathsf{MT}(\mathsf{BF\text{-}COIE}) + \mathsf{FT}(\mathsf{BF\text{-}COIE})$$

Although the implementation (nor the algorithm) of the matching step of LEAF protocol is not available in [45], we expect that it holds $\mathsf{MT}(\mathsf{LEAF}) \approx \mathsf{MT}(\mathsf{BF\text{-}COIE})$. In the experiment performed in LEAF (see Figure 9 in [45]), we have $m = \frac{\mathsf{MT}(\mathsf{LEAF})}{\mathsf{FT}(\mathsf{LEAF})} \approx 1.5$. For $s = 16$, setting $\mathsf{FT}(\mathsf{LEAF}) = 1800 \cdot \mathsf{FT}(\mathsf{BF\text{-}COIE})$ based on the above discussion, we can estimate the speed-up as follows:

$$\frac{Time(\mathsf{LEAF})}{Time(\mathsf{BF\text{-}COIE})} = \frac{s \cdot (m+1)}{m + 1/1800}.$$

Thus, with $s = 16$, we estimate that our BF-COIE scheme has roughly 26X end-to-end speed-up.

7.3 Communication

We now look at the communication required by each of our schemes and by LEAF+. Figure 5 shows the network cost of the protocols when the result set size s is 16 and the size of the database is $n = 10,000$. In our implementations, the length of an FHE ciphertext is approximately 103KB and the communication cost of PIR is approximately 369KB.

To explain this table, we first need to explain how we determined the costs of LEAF+ and PIR.

	LEAF+	BF-COIE	PS-COIE	BFS-CODE
#ct's	704	1323	17	1321
#PIR	0	32	16	0
#ct's (w/ batching)	32	2	2	2

Figure 5: The communication costs ($n = 10,000$ and $s = 16$).

- *LEAF+.* Since LEAF+ fetches each data item and the corresponding index one by one, LEAF+ needs to 16 rounds of communication to retrieve 16 data items. Worse yet, LEAF+ requires the client to send the index of the previous match (requiring $\lg n$ bits) in his next query to ensure correctness. Finally, LEAF+ uses bitwise encryption requiring a ciphertext for each bit of the encrypted communication. Thus, in a single round, the client must send $\lg n = 14$ ciphertexts and the server returns $16 + \lg n = 30$ ciphertexts – 16 ciphertexts for returning the matching data item, and $\lg n$ ciphertexts to return its index. This amounts to 704 ciphertexts for fetching 16 items (excluding the query).
- *PIR costs.* We reduce the cost of PIR for the COIE-based schemes by making a slight modification. In addition to storing the FHE-encrypted database, the server also stores a copy of each record encrypted using a symmetric-key encryption scheme (resulting in much shorter ciphertexts). Then, in the PIR step, the client fetches this symmetrically encrypted ciphertext instead of the FHE-encrypted one.
  We use SealPIR for our PIR protocol, which requires 368.6 KB per request. We remark that a very recently introduced SealPIR+ takes 80KB per request (see Table 1 in [3]), using which we can reduce the communication further.

We can now compare the communication costs based on rows 1 and 2 of Figure 5. We see that the communication of BF-COIE and BFS-CODE are roughly twice that of LEAF+, while PS-COIE requires almost 10X less communication. The extra communication needed by BF-COIE and BFS-CODE can likely be offset by the much lower round complexity required by our protocol since the latency costs are likely higher than the cost for the extra bandwidth.

Reducing communication using ciphertext batching. We now describe an optimization to significantly reduce the communication of our protocols at the cost of slightly increased server computation. SEAL allows thousands of encrypted values to be packed together into a single ciphertext. This allows us to pack the ciphertexts in all of our protocols into just one a single ciphertext to be sent from the server to the client. However, this does require the server to do some additional computation to pack the ciphertexts prior to sending them. We experimentally measured this packing, and it requires approximately 3 seconds on a single threaded machine.

LEAF+ can also take advantage of packing to reduce the communication of their protocols. However, since the results must be returned one at a time, the best LEAF+ can do is to pack all ciphertexts that are sent in each round, resulting in a total of 32 ciphertexts.

We note that the cost of PIR is unchanged by this modification. Thus, with the packing optimization, the communication of BFS-CODE is roughly 1/16 of the communication needed by LEAF+, but BF-COIE and PS-COIE require approximately 4X and 2X more

communication than LEAF+ respectively when SealPIR is used; however, when SealPIR+ is used, both schemes have slightly less communication than LEAF+.

8 RELATED WORK

8.1 Techniques for Secure Search

Secure pattern matching (SPM) on FHE-encrypted data. In SPM, given an encrypted query $[\![q]\!]$ and n FHE-encrypted data items $([\![x_1]\!], \ldots, [\![x_n]\!])$, it returns a vector of n ciphertexts $[\![b_1]\!], \ldots, [\![b_n]\!]$, where b_i indicates whether the ith data element is a match [16, 17, 32, 47]. Their works focus on optimizing the search circuits to determine whether a data item matches the query, and therefore the communication complexity and client's running time are proportional to the number of data items. Our work focuses on the orthogonal problem of optimizing the retrieval of the matched data items with sublinear communication and client computation.

Searchable encryption (SE). Searchable encryption [12, 41] allows highly efficient search (usually in $o(n)$ time) over encrypted data. Efficient SE schemes have been proposed for a wide variety of queries including equality queries [15, 19], range queries [29, 38], and conjunctive queries [14, 36]. However, to achieve sublinear query performance, SE schemes require significant preprocessing and relax security, allowing some partial information about the queries and data (e.g. access patterns) to leak to the server. For a recent survey on SE constructions and security, see Fuller et al. [22]. In contrast, our work focuses on achieving preprocessing-free secure constructions, leaking nothing about the queries or results other than their sizes.

Property Preserving Encryption (PPE). As a different approach, property-preserving encryption [35] produces ciphertexts that maintain certain relationships (e.g., equality, and order) of the underlying plaintexts. This allows queries to be performed over ciphertexts in the same way that they can be carried out over plaintexts. Examples of PPE include deterministic encryption [6] allowing equality queries, and order-preserving encryption [10, 11] allowing range queries. However, it has been shown [26, 27, 30] that such property-preserving ciphertexts leak a lot of information about the underlying plaintexts. See [22] for a survey of constructions and attacks.

8.2 General Techniques

Private information retrieval (PIR). PIR allows the client to choose the index i and retrieve the ith record from an untrusted server while hiding the index i [18]. However, this protocol by itself provides only a limited search functionality requiring the client to know the index of the data to retrieve. In this work, we aim at protocols supporting any arbitrary search functionality.

Secure multi-party computation (MPC). Secure two-party computation [23, 46] allows players to compute any function of their private inputs without compromising privacy of their inputs. For example, the client and the server can run a protocol for secure two-party computation to solve the secure search problem. While there has been much progress in improving efficiency of MPC protocols, such protocols still require $\Omega(n)$ communication and $\Omega(n)$ client computation per query. In this work, we aim to achieve protocols with sublinear communication and client work.

Oblivious RAM (ORAM) and Oblivious data structure (ODS). ORAM [24] is a protocol which allows a client to store an array of n items on an untrusted server and to access an item obliviously, that is, hiding contents and which item is accessed (i.e., the access pattern). Likewise, ODS [44] allows the client to store and use a data structure obliviously. One could implement secure search by utilizing an ODS for a search tree. However, ODS constructions typically need $\Omega(\log^2 n)$ rounds for each operation. In this work, we aim at achieving a constant round protocol.

9 CONCLUSION

We have presented several new constructions of secure search based on fully homomorphic encryption. Prior constructions were inherently sequential, returning only a single record from the result set, and requiring a new query from the client that depended on the index of the previous match. We have demonstrated several new methods for encoding the entire result set at one time, removing the added rounds, and allowing the server work to be parallelized. Additionally, we have shown that this can be done without homomorphic multiplication, ensuring low computational cost at the server. Finally, we have implemented our constructions, and demonstrated up to three orders of magnitude speed-up over prior work. Additionally, we introduced the notion of compressed oblivious encoding which may be of independent interest.

ACKNOWLEDGEMENTS

Dana Dachman-Soled is supported in part by NSF grants CNS-1933033, CNS-1453045(CAREER), and by financial assistance awards 70NANB15H328 and 70NANB19H126 from the U.S. Department of Commerce, National Institute of Standards and Technology; Seung Geol Choi is supported by ONR N0014-20-1-2745 and NSF grant CNS-1955319; S. Dov Gordon is supported by the NSF Grants CNS-1942575 and CNS-1955264, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-15-C-4070, by the Blavatnik Interdisciplinary Cyber Research Center at Tel-Aviv University and Israel National Cyber Directorate (INCD), and by a Google faculty award; Arkady Yerukhimovich is supported by NSF grant CNS-1955620, and by a Facebook Research Award.

REFERENCES

[1] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search on Encrypted Data via Multi-Ring Sketch. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 985–1001. https://doi.org/10.1145/3243734.3243810

[2] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. 2019. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 87–107. https://doi.org/10.2478/popets-2019-0038

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2021. Communication–Computation Trade-offs in PIR. Usenix Security (To appear). Available at https://ia.cr/2019/1483.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 962–979. https://doi.org/10.1109/SP.2018.00062

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *EUROCRYPT 2020, Part II (LNCS, Vol. 12106)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg,

Germany, Zagreb, Croatia, 403–432. https://doi.org/10.1007/978-3-030-45724-2_14

[6] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and Efficiently Searchable Encryption. In *CRYPTO 2007 (LNCS, Vol. 4622)*, Alfred Menezes (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 535–552. https://doi.org/10.1007/978-3-540-74143-5_30

[7] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *NDSS 2020*. The Internet Society, San Diego, CA, USA.

[8] Marina Blanton and Everaldo Aguiar. 2011. Private and Oblivious Set and Multiset Operations. Cryptology ePrint Archive, Report 2011/464. http://eprint.iacr.org/2011/464.

[9] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[10] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *EUROCRYPT 2009 (LNCS, Vol. 5479)*, Antoine Joux (Ed.). Springer, Heidelberg, Germany, Cologne, Germany, 224–241. https://doi.org/10.1007/978-3-642-01001-9_13

[11] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO 2011 (LNCS, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 578–595. https://doi.org/10.1007/978-3-642-22792-9_33

[12] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. 2004. Public Key Encryption with Keyword Search. In *EUROCRYPT 2004 (LNCS, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, Heidelberg, Germany, Interlaken, Switzerland, 506–522. https://doi.org/10.1007/978-3-540-24676-3_30

[13] D. Cantor and H. Zassenhaus. 1981. A new algorithm for factoring polynomials over finite fields. *Math. Comp.* 36 (1981), 587–592.

[14] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO 2013, Part I (LNCS, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 353–373. https://doi.org/10.1007/978-3-642-40041-4_20

[15] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT 2010 (LNCS, Vol. 6477)*, Masayuki Abe (Ed.). Springer, Heidelberg, Germany, Singapore, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

[16] Jung Hee Cheon, Miran Kim, and Myungsun Kim. 2016. Optimized Search-and-Compute Circuits and Their Application to Query Evaluation on Encrypted Data. *IEEE Trans. Inf. Forensics Secur.* 11, 1 (2016), 188–199. https://doi.org/10.1109/TIFS.2015.2483486

[17] Jung Hee Cheon, Miran Kim, and Kristin E. Lauter. 2015. Homomorphic Computation of Edit Distance. In *FC 2015 Workshops (LNCS, Vol. 8976)*, Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff (Eds.). Springer, Heidelberg, Germany, San Juan, Puerto Rico, 194–212. https://doi.org/10.1007/978-3-662-48051-9_15

[18] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (1998), 965–981. https://doi.org/10.1145/293347.293350

[19] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006*, Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati (Eds.). ACM Press, Alexandria, Virginia, USA, 79–88. https://doi.org/10.1145/1180405.1180417

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144. http://eprint.iacr.org/2012/144

[21] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* 8, 3 (2000), 281–293. https://doi.org/10.1109/90.851975

[22] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. 2017. SoK: Cryptographically Protected Database Search. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 172–191. https://doi.org/10.1109/SP.2017.10

[23] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[24] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[25] Michael T. Goodrich. 2011. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*. 379–388. https://doi.org/10.1145/1989493.1989555

[26] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1353–1364. https://doi.org/10.1145/2976749.2978351

[27] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 655–672. https://doi.org/10.1109/SP.2017.44

[28] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 361–378. https://doi.org/10.1145/3319535.3363210

[29] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2016. Private Large-Scale Databases with Distributed Searchable Symmetric Encryption. In *CT-RSA 2016 (LNCS, Vol. 9610)*, Kazue Sako (Ed.). Springer, Heidelberg, Germany, San Francisco, CA, USA, 90–107. https://doi.org/10.1007/978-3-319-29485-8_6

[30] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS 2012*. The Internet Society, San Diego, CA, USA.

[31] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1329–1340. https://doi.org/10.1145/2976749.2978386

[32] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. 2019. Private Compound Wildcard Queries Using Fully Homomorphic Encryption. *IEEE Trans. Dependable Secur. Comput.* 16, 5 (2019), 743–756. https://doi.org/10.1109/TDSC.2017.2763593

[33] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 887–903. https://doi.org/10.1145/3319535.3354238

[34] Michael Mitzenmacher. 2001. Compressed bloom filters. In *20th ACM PODC*, Ajay D. Kshemkalyani and Nir Shavit (Eds.). ACM, Newport, Rhode Island, USA, 144–150. https://doi.org/10.1145/383962.384004

[35] Omkant Pandey and Yannis Rouselakis. 2012. Property Preserving Symmetric Encryption. In *EUROCRYPT 2012 (LNCS, Vol. 7237)*, David Pointcheval and Thomas Johansson (Eds.). Springer, Heidelberg, Germany, Cambridge, UK, 375–391. https://doi.org/10.1007/978-3-642-29011-4_23

[36] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Berkeley, CA, USA, 359–374. https://doi.org/10.1109/SP.2014.30

[37] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 79–93. https://doi.org/10.1145/3319535.3354213

[38] Daniel S. Roche, Daniel Apon, Seung Geol Choi, and Arkady Yerukhimovich. 2016. POPE: Partial Order Preserving Encoding. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1131–1142. https://doi.org/10.1145/2976749.2978345

[39] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2017. P2P Mixing and Unlinkable Bitcoin Transactions. In *NDSS 2017*. The Internet Society, San Diego, CA, USA.

[40] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[41] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Oakland, CA, USA, 44–55. https://doi.org/10.1109/SECPRI.2000.848445

[42] W. A. Stein et al. 2020. *Sage Mathematics Software (Version 9.2)*. The Sage Development Team. `http://www.sagemath.org`.

[43] Alexander J. Titus, Shashwat Kishore, Todd Stavish, Stephanie M. Rogers, and Karl Ni. 2018. PySEAL: A Python wrapper implementation of the SEAL homomorphic encryption library. arXiv:1803.01891 [q-bio.QM]

[44] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *ACM CCS 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM Press, Scottsdale, AZ, USA, 215–226. https://doi.org/10.1145/2660267.2660314

[45] Rui Wen, Yu Yu, Xiang Xie, and Yang Zhang. 2020. LEAF: A Faster Secure Search Algorithm via Localization, Extraction, and Reconstruction. In *ACM CCS 20*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 1219–1232. https://doi.org/10.1145/3372297.3417237

- [46] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167. https://doi.org/10.1109/SFCS.1986.25
- [47] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. 2013. Secure pattern matching using somewhat homomorphic encryption. In *CCSW'13, Proceedings of the 2013 ACM Cloud Computing Security Workshop, Co-located with CCS 2013, Berlin, Germany, November 4, 2013*, Ari Juels and Bryan Parno (Eds.). ACM, 65–76. https://doi.org/10.1145/2517488.2517497

A PROOF OF LEMMA 4.1

Lemma A.1 (4.1). *Consider a Bloom filter with false positive rate $\frac{1}{m}$, where m is an arbitrary positive integer. Suppose at most m $\mathsf{BF.Check}$ operations are performed in the BF. Then, for any $\delta > 0$, we have:*

$$\Pr[\#\text{ false positives} \geq 1 + \delta] \leq \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}.$$

Proof. Let α_i be the ith item that is checked through $\mathsf{BF.Check}$. That is, we consider a sequence of

$$\mathsf{BF.Check}(\alpha_1), \ldots, \mathsf{BF.Check}(\alpha_m),$$

where α_i is an arbitrary item. Since we wish to upper bound the false positives (i.e., we don't care about true positives), it suffices to consider the case that for every i, $\alpha_i \notin \mathsf{BF}$ (i.e, α_i has not been inserted in the BF) as this maximizes the number of possible false positives.

Let $X_1, \ldots, X_m$ be independent Bernoulli random variables with $\Pr[X_i = 1] = 1/m$. Since the BF false positive rate is assumed to be $1/m$, we have for all i,

$$\Pr[\mathsf{BF.Check}(\alpha_i) = 1] = \Pr[\text{query } i \text{ is a false positive}] \leq 1/m.$$

Thus, we can bound the number of false positives by $\sum_{i=1}^{m} X_i$.

Now, let $\mu := \mathbf{Exp}[\sum X_i] = m \cdot \frac{1}{m} = 1$. By applying the Chernoff bound with $\mu = 1$, we have:

$$\Pr\left[\sum_{i=1}^{m} X_i \geq 1 + \delta\right] \leq \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}.$$

□

B PROOF OF THEOREM 6.3

Theorem B.1 (6.3). *Given an FHE scheme, and a (n, s, c, f_p)-CODE scheme over domain D in the random oracle model, the construction in Algorithm 8 yields a (ℓ, f_p)-secure search scheme for records in domain D in the random oracle model, where $\ell = \frac{c(s) \cdot \ell_c}{s}$, ℓ_c is the length of an FHE ciphertext with plaintext space D, and s is the number of matching records.*

Proof. We begin by proving that the adversary cannot distinguish between two different queries. The adversary chooses a database x and two queries q^0, q^1, with the promise that $s = \sum_{i=1}^{n} q^0(x_i) = \sum_{i=1}^{n} q^1(x_i)$.

The entire view of the adversary during the experiment can be reconstructed efficiently given (1) the encrypted database $[\![x]\!]$, (2) the encrypted query $[\![q]\!]$, (3) the decrypted value of s.

We note that the CODE scheme may return either more than s values to the client (in case of a false positive) or less than s values (in case decoding fails), but both of these occur with probability at most $\mathsf{negl}(\lambda)$ and thus we can ignore them in the following.

Since the value of s is the same for q_0 and q_1, the only thing that changes in the view of the adversary when switching from $b = 0$ to $b = 1$ is the encrypted query $\tilde{q_b}$. Therefore, the adversary guesses b with negligible advantage by the IND-CPA security of the FHE scheme.

The proof that the adversary cannot distinguish between the same query applied to two different databases follows nearly identically.

□