Secure Search on Encrypted Data via Multi-Ring Sketch

Extended Abstract

Adi Akavia
Cybersecurity Research Center
Academic College of Tel-Aviv Jaffa
adi.akavia@gmail.com

Dan Feldman
Robotics & Big Data Lab
University of Haifa
dannyf.post@gmail.com

Hayim Shaul
Robotics & Big Data Lab
University of Haifa
hayim.shaul@gmail.com

ABSTRACT

We consider the *secure search* problem of retrieving from an unsorted data $array = (x_1, \dots, x_m)$ an item (i, x_i) matching a given lookup value ℓ (for a generic matching criterion either hardcoded or given as part of the query), where both input and output are encrypted by a Fully Homomorphic Encryption (FHE). The secure search problem is central in applications of secure outsourcing to an untrusted party ("the cloud"). Prior secure search algorithms on FHE encrypted data are realized by polynomials of degree $\Omega(m)$, evaluated in $\Omega(\log m)$ sequential homomorphic multiplication steps (i.e., multiplicative depth) even with unboundedly many parallel processors. This is too slow with current FHE implementations even for moderate array sizes m such as a few thousands.

We present the first secure search algorithm that is realized by a polynomial of *logarithmic degree*, evaluated in $O(\log\log m)$ sequential homomorphic multiplication steps (i.e., multiplicative depth) using m parallel processors. We implemented our algorithm in an open source library based on HElib and ran experiments on Amazon's EC2 cloud with up to 100 processors. Our experiments show that we can securely search in $m =$ *millions of entries in less than an hour* on a standard EC2 64-cores machine.

We achieve our result by: (1) Employing modern data summarization techniques known as sketching for returning as output (the encryption of) a short sketch C from which the matching item (i, x_i) can be decoded in time proportional to computing poly$(\log m)$ decryption/encryption operations. (2) Designing for this purpose a novel sketch that returns the first strictly-positive entry in a (not necessarily sparse) array of non-negative real numbers; this sketch may be of independent interest. (3) Suggesting a multi-ring evaluation of FHE for degree reduction from linear to logarithmic.

CCS CONCEPTS

• **Security and privacy** → *Public key encryption*;

KEYWORDS

secure search, fully homomorphic encryption, homomorphic encryption, sketching, low degree polynomials, first positive sketch

ACM Reference Format:
Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search on Encrypted Data via Multi-Ring Sketch: Extended Abstract. In *Proceedings of 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243810

1 INTRODUCTION

Storage and computation are rapidly becoming a commodity with an increasing trend of organizations and individuals (client) to outsource storage and computation to large third-party systems often called "the cloud" (server). Usually this requires the client to reveal its private records to the server so that the server would be able to run the computations for the client. With e-mail, medical, financial and other personal information transferring to the cloud, it is paramount to guarantee *privacy* on top of data availability while keeping the correctness of the computations.

Fully Homomorphic Encryption (FHE) [16, 17, 38] is an encryption scheme that facilitates secure outsourcing of computation due to its special property of enabling computing on encrypted data; see a survey in [25]. Specifically, FHE allows computing any algorithm on encrypted input (ciphertexts), with no decryption or access to the secret key that would compromise secrecy, yet succeeding in returning the encryption of the desired outcome.

Secure outsourcing of computation using FHE is conceptually simple: the client sends the ciphertext $[\![x]\!]$ encrypting the input x and receives the ciphertext $[\![y]\!]$ encrypting the output $y = f(x)$, where the computation is done on the server's side requiring no further interaction with the client. This gives a *single round* protocol and with *low communication*; specifically the communication complexity is proportional only to the sizes of the input and output ciphertexts (in contrast to communication proportional to the running time of computing $f()$ when using prior secure multi-party computation (MPC) techniques [19, 44]). The semantic security of the underlying FHE encryption ensures that the server learns no new information on the plaintext input and output from seeing and processing the ciphertexts.

A main challenge for designing algorithms that run on data encrypted with fully (or leveled) homomorphic encryption (FHE) is to present their computation as a *low degree polynomial* $f()$, so that on inputs x the algorithm's output is $f(x)$ (see examples in [11, 15, 21, 30, 32, 33, 47]). Otherwise, a naive conversion resulting in a high degree polynomial $f()$ would typically be highly impractical for the current state-of-the-art FHE implementations, where running time is rapidly growing with degree and the multiplicative depth of the corresponding circuit.

Secure search using FHE has been a leading example for FHE applications since Gentry's breakthrough result construction the first FHE candidate [16]. We consider the *secure search* problem of retrieving from an unsorted data $array = (x_1, \ldots, x_m)$ an item (i, x_i) matching a given lookup value ℓ (with a generic matching criterion, e.g., equality or similarity test), where both input and output are encrypted with Fully Homomorphic Encryption (FHE). We remark that in the context of secure retrieval from a database, we think of *array* as the database table (or specific column), and ℓ the query.

In the context of secure outsourcing, we focus on *single round* protocols and with *low communication*, where the server requires no interaction with the client beyond receiving the encrypted input and returning the encrypted output.

Definition 1 (Secure Search). The server holds an unsorted array of encrypted values (previously uploaded to the server, and where the server has no access to the secret decryption key):

$$[\![array]\!] = ([\![x_1]\!], \ldots, [\![x_m]\!])$$

(here and throughout this work, $[\![msg]\!]$ denotes the ciphertext encrypting message *msg*; the encryption can be any fully, or leveled, homomorphic encryption (FHE) scheme, e.g. [6]). The client sends to the server an encrypted lookup value $[\![\ell]\!]$. The server returns to the client an encrypted index and value

$$[\![y]\!] = ([\![i]\!], [\![x_i]\!])$$

satisfying the condition:

$$is\textsc{Match}(x_i, \ell) = 1$$

for *is*MATCH() a predicate specifying the search condition (see discussion below on using generic predicates). More generally, y may be a value (sketch) from which the client can compute (i, x_i) (decode).

We say the client / server / protocol is efficient if the following holds:

- The *client is efficient* if its running time is polynomial in the time to compute $|\ell|$ encryptions and $|(i, x_i)|$ decryptions of the underlying FHE (for $|\ell|$ and $|(i, x_i)| = \log m + |x_i|$ the bit representation length of the input and output respectively).
- The *server is efficient* if the polynomial $f([\![array]\!], [\![\ell]\!])$ the server evaluates to obtain $[\![y]\!]$ is of degree polynomial in $\log m$ and the degree of *is*MATCH(), and of size (i.e., the overall number of addition and multiplication operations for computing f) polynomial in m and the size of *is*MATCH.
- The *protocol is efficient* if both client and server are efficient.

We call the client / server / protocol *inefficient* if the running time / degree / either is at least $\Omega(m)$.

Generic *is*MATCH predicate. The predicate *is*MATCH() in Definition 1 is a generic predicate that can be instantiated to any desired functionality (with complexity affected accordingly, see Theorem 2). Moreover, a concise specification of *is*MATCH() can typically be used, e.g., by the client providing the function's name. For example, most generally, *is*MATCH() can be a *universal circuit* and ℓ a full specification of the predicate defining the matching values. Alternatively, giving a more concrete instantiation, we can extend the search query to provide the name for a particular *is*MATCH() circuit to be used, chosen from a commonly known set of options (for example, equality operator, conjunction/disjunction query, range query, similarity condition, and so forth). Even more concretely, we can fix a particular predicate in advance, say, the equality condition $is\textsc{Match}(x_i, \ell) = 1$ if-and-only-if $x_i = \ell$. Looking ahead, our experiments are for the latter case; nonetheless, our results are general and apply to any generic *is*MATCH() condition (see Theorem 2).

A toolbox of concrete instantiations of *is*MATCH predicate on FHE encrypted data has been provided by prior works [10, 12, 23, 26, 27, 31, 46], including for example efficient implementations for computing Hamming and edit distances. These works focus on efficient computing the *is*MATCH predicate given the two values x_i and ℓ. This is then employed to either give a YES/NO answer on whether a match exists in the data array (but without returning the matching record or its index), or returning the length m indicator vector $indicator = (is\textsc{Match}(x_1, \ell), \ldots, is\textsc{Match}(x_m, \ell))$ indicating for each record x_i whether or not it is a match to the lookup value ℓ. In this work we focus on the complementary problem of retrieving the matched entry (i, x_i), while using as a black-box the provided *is*MATCH criterion.

Threat Model. The functionality that our protocols implement has client's input $(array, \ell)$; client's output the retrieved record (i, x_i); and the server has no input or output beyond the shared parameters, including most notably, the number of data records and the sizes of records and lookup values (see Figure 4)

The adversaries we address are computationally-bounded semi-honest adversaries controlling the server. Namely, the adversary follows the protocol, but may try to learn additional information. We point out that there is no need to consider adversaries controlling the client because the server has no input/output.

Our security requirement is that the adversary learn no information from participating in the protocol beyond what is explicitly leaked in the shared parameters. In particular, if the client issues the protocols with one of two adversarially-chosen equal size lookup values (similarly, data arrays), the adversary controlling the server cannot distinguish between them. Looking ahead, security follows immediately from the semantic security of the FHE scheme.

Applications of secure search on FHE encrypted data are abundant in the context of secure outsourcing of computation to an untrusted party ("the cloud"/ the server), examples including:

- Secure retrieval from a database
- Secure search in images corpus' tags
- Secure search in text documents
- Secure search in genomic data

In all applications, the lookup value and data (database, images/tags, text documents, genomic data) are encrypted by the client prior to being sent to the server, to ensure their secrecy.

The most relevant applications arise in settings where data pre-sorting or pre-indexing is infeasible, and linear scan of the data is used regardless of security.[1] These are particularly appealing scenarios due to known lower bounds necessitating a linear scan for securely searching on FHE encrypted data. Such applications arise for example in settings with:

[1]Note that securely sorting an encrypted array is believed to be harder than secure search; furthermore, in-order insertion to an encrypted array seems to require secure search as a prerequisite.

- A-priori unknown matching criterion, as in ad-hoc SQL queries and our generic *is*Match predicate, where indexing is impossible;
- Versatile matching criteria requiring indexes of size exponential in the number of attributes, as in range queries on high dimensional data;
- Streaming data with each element discarded by the client immediately after being encrypted and uploaded to the server, making pre-sorting the entire clear-text data impossible;
- Low capacity client that is too weak to store/sort the clear-text data, as in Internet-of-Things (IoT) devices;
- Fragmented data uploaded to the server from multiple distinct clients (e.g., agents/users/devices) with no single entity that holds and can pre-sort the entire clear-text data.

Our main motivation in this paper is to answer affirmatively the following question: **Is there an *efficient* secure search protocol?**

1.1 Related Works

Secure search is a fundamental computational problem that has been widely studied. We survey here the works most relevant to our secure search formulation (Definition 1) of searching on FHE encrypted data, in a 1-round low communication protocol, and with efficient communication, client and server.

Secure two-party computation (2PC) [20, 45] enables two parties to compute any (polynomial-time computable) function of their private inputs via an interactive protocol that reveals no information beyond what can be inferred from the function's output. However, *the communication complexity* in these solutions *grows with the complexity for computing the function*, in contrast to being proportional only to the size of input and output.

Computing on FHE encrypted data [16, 38] resolves the aforementioned high communication caveat, showing that secure two-party computation with 1-round and low communication theoretically feasible (assuming semantic security of the FHE). This FHE approach however incurs a high computational overhead (albeit polynomial). Designing such protocols attaining reasonable concrete efficiency for specific problems has been studied in many prior works; we discuss below the ones most relevant to our secure search problem.[2]

The natural (folklore) solution for secure search on FHE encrypted data suffers from an inefficient server that evaluates polynomial of degree $\Omega(m)$ for m the number of records (instead of poly-logarithmic in m as we require). This is too slow with current FHE candidates and implementations, even for moderate size m such as a few thousands.

Secure pattern matching on FHE encrypted data [10, 12, 23, 26, 27, 31, 46] is a problem related to secure search. In this problem, the server is given encrypted data and encrypted lookup value. It then computes an array of encrypted indicators specifying for each data element x_i whether it is a match for the lookup value ℓ:

$$[\![indicator]\!] \leftarrow ([\![is\textsc{Match}(x_1,\ell)]\!],\dots,[\![is\textsc{Match}(x_m,\ell)]\!])$$

(where the matching criterion *is*Match vary from work to work). The server then sends the array *indicator* to the client (or, in some of these works, returns a YES/NO answer of whether a match exists). The client can then decrypt and scan this length m array of indicators to find the indices of the matching records. To retrieve the records themselves, the parties can now engage in a Private Information Retrieval (PIR) protocol (see below), requiring a second round of interaction.

In these works, while the server is efficient (computes polynomials of degree deg(*is*Match)), the client and communication are not: they are at least linear in the number of records $\Omega(m)$ (instead of logarithmic, as we require).

Search with UNIQUE constraint via FHE based PIR. In Private Information Retrieval (PIR) [13] the client queries for an index i, and the server sends the corresponding record x_i, with the requirement that the server does not learn i or x_i. Much work has been done on PIR protocols (see surveys [34, 35]), including obtaining 1-round low communication PIR protocols using FHE [7, 14, 16]. In these FHE based PIR protocols the server evaluates a polynomial of degree $O(\log m)$ on the encrypted index and data that returns the encrypted record.

For the restricted search functionality of searching with a lookup value that is constraint to be a *unique record identifier* (as in SQL UNIQUE), the aforementioned FHE based PIR protocols can be extended to provide a solution. Here, the assumption of the model is that there is at most one match in *array* for the lookup value ℓ. Specifically, the client queries for a unique record identifier and the server sends the corresponding record (where both the identifier and record are encrypted so that the server does not learn what they are). The server's computation here can be realized by a polynomial of degree proportional to the degree of the matching criterion *is*Match, resulting in a protocol with efficient communication, client and server.

The unique identifier constraint however may be a major obstacle in applications, most notably when the encrypted data is not indexed and unsorted as is the focus of our work. Specifically, uniqueness means that the lookup value ℓ must match at most a single record in *array*, i.e., $|\{\, i \in [m] \mid is\textsc{Match}(x_i,\ell) = 1\}| \le 1$, which we do not expect to generally hold. For example, uniqueness is not likely to hold when searching for a common name ("John", "Mohamed") in a streamed array of names. We point out that the unique identifier constraint cannot be alleviated by batching queries $i_1,\dots,i_t$ to return multiple records $x_{i_1},\dots,x_{i_t}$ (as in [22]), because each of these batched queries must still be a unique identifier for at most a single record in the data.

When the data held by the server is non-encrypted (unlike our settings where both data and query are encrypted), pre-indexing can be used to enforce uniqueness [8, 39]. Specifically, the server can transform the data into a table with a unique row for each lookup value ℓ holding the list of all matching records. This transformation however may incur considerable time and memory overhead, as it produces a table of size $m \cdot |\mathcal{L}|$ for $\mathcal{L}$ the set of possible lookup

[2]We remark that, while we focus on secure search in the two parties settings (client and server), promising results have been shown for settings where the server can be partitioned into several non-colluding entities that secret share the data, e.g., in [4, 42] for settings where data is known to the server and requires protection from the client in the former and is public in the latter; such works are beyond the scope of this literature survey.

values (compared to size m of the original data). Moreover, this transformation is infeasible when $\mathcal{L}$ is a-priori unknown or very large.

Private set intersection (PSI) is another search related problem where FHE based solutions were proposed [9]. In PSI Alice has a set X, Bob has a set Y, and they wish to compute the intersection $X \cap Y$ while revealing no additional information on their sets. This PSI formulation has similarities to search as we can identify X with the lookup value (for $|X| = 1$) and Y with the data array. However, the output $X \cap Y$ only indicates whether the lookup value appears in the data, without returning the record's index i (which is necessary for the client to issue further data management instructions on this record). Moreover, the lookup value X is constrained to be a unique identifier for a record in Y, as discussed in the context of PIR, due to modeling X, Y as sets rather than multi-sets.

The searchable encryption approach takes a different route of exploring the efficiency versus security tradeoff. The approach is to deliberately *leak information* on the underlying data, which is then employed in vital ways to enable fast search and data retrieval. This approach has been extensively studied starting with the pioneering work of Song, Wagner and Perrig, IEEE S&P 2000 [41], with famous examples such as CryptDB [37] and subsequent works [28, 29, 36] giving highly practical search solutions albeit with information leakage; see a survey in [5].

1.2 Our Contribution

In this work we provide the first secure search on FHE encrypted realized by a polynomial of degree logarithmic (rather than linear) in the number m of array entries data. Our contributions are as follow.

Contribution 1: The first efficient protocol for secure search (see Definition 1 and Figure 9) that is applicable to large datasets with unrestricted search functionality and full security. Specifically, our protocol provides:

- *Efficient client:* The client's running time is proportional to the time to compute $k = 1 + \log^2 m / \log\log m = o(\log^2 m)$ encryptions for the lookup value ℓ, and k decryptions of retrieved index and record (i, x_i).
- *Efficient server:* The server evaluates polynomials of degree $O(\log^3) \cdot d$, overall number of multiplications $O(m(s + \log m))$, and size polynomial in m and s, for d, s the degree and size of the polynomial realizing the *is*Match predicate.
- *Efficient Communication*: Single round protocol with communication complexity proportional to k encryptions of the lookup value ℓ and the search outcome (i, x_i).
- *Unrestricted search functionality:* The protocol is applicable to any data *array* and lookup value ℓ, with no restrictions on number of records in *array* that match the lookup value ℓ. Moreover, the protocol is modular and can be used with generic matching criteria *is*Match.
- *Full security:* The input data *array* and lookup value ℓ are encrypted with fully, or leveled, homomorphic encryption (FHE) achieving the strong property of semantic security both for data at rest and at use for search.

This is informally summarized in Theorem 2 below (for formal statements and proofs see section 4).

Theorem 2 (Secure Search). *There exists an efficient secure search protocol (see Definition 1 and Figure 9).*

In contrast, prior works either have an inefficient client with running time is $\Omega(m)$ (see pattern matching solutions); or an inefficient server evaluating a polynomial of degree $\Omega(m)$ (see the natural folklore solution); or restrict the search functionality by requiring the lookup value is a unique record identifier as in SQL UNIQUE constraint (see PIR and PSI solutions); or require higher communication and/or interaction (see 2PC solutions); or compromises security (see searchable encryption solutions). A summary of the comparison to prior single round low communication protocols appears in Tables 1-2.

In summary, our protocol is the first single round, low communication, efficient protocol for secure search with full security and unrestricted functionality. In settings where leakage or high communication/interaction is unacceptable, this provides the first applicable secure search solution.

Contribution 2: System and experimental results for secure search. We implemented our protocol into a system that runs on Amazon's EC2 cloud on 1-100 processors (cores). Our experiments demonstrating, in support of our analysis, that on a standard EC2 64-cores machine (x1.large) we can answer search queries on database with millions of entries in less than an hour; namely, we achieve a searching rate of millions of records per hour per machine; See Figure 1 and Section 5.

Contribution 3: Fast parallel computation. With m parallel processors, our algorithm requires only $O(\log\log m)$ sequential multiplication steps (in contrast to $\Omega(\log m)$ in the folklore polynomial, even with unbounded number of parallel processors). Our experimental results on up to 100 cores on Amazon's EC2 cloud indeed show that performance scales almost linearly with the number of computers.

Contribution 4: High accuracy formulas for estimating running time that allow potential users and researchers to estimate the practical efficiency of our system for their own cloud and databases; See Section 5.3 and Figure 1.

Contribution 5: Open Source Library for Secure-Search with FHE, based on HElib [24], that is provided for the community [1], to reproduce our experiments, to extend our results for real-world applications, and for practitioners at industry or academy that wish to use these results for their future papers or products.

1.3 Techniques Overview

We give the high level ideas of our secure search protocol for returning the first record in *array* matching the lookup value ℓ , when given encrypted *array*, ℓ. To ease the reading of the overview of the server's computation, we specify the computation as if it were executed on plaintext data. In the actual protocol, the server computes on encrypted data. This is by evaluating the homomorphic version of the computation specified here.

As our first attempt for solving secure search on FHE encrypted data, suppose the server homomorphically evaluates the following

	Client's time	Server's degree	Protocol
Secure Pattern Matching	$T \cdot \Omega(m)$	d	inefficient
Folklore Secure Search	$T \cdot \Theta(\log m)$	$\Omega(m)$	inefficient
This Work: Secure Search	$T \cdot o(\log^2 m)$	$O(\log^3 m) \cdot d$	efficient

Table 1: Comparison of client, server, and protocol complexity (see Definition 1) in works supporting unrestricted search functionality, index retrieval, single round low communication protocol, and full security. Here T is the time to encrypt/decrypt a single element with the underlying FHE; d is the degree of the underlying pattern matching polynomial isMatch (e.g., $d = O(|x_i|)$ is the bit representation length of data records, when testing for equality).

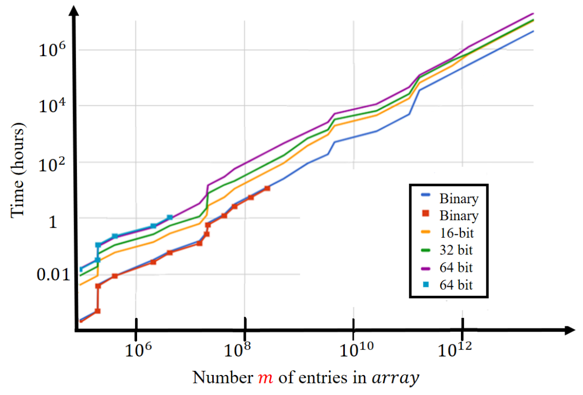

Figure 1: Server's running time in secure search protocol of Figure 9 (y-axis) as a function of the number of records m in database $array$ (x-axis), where each entry $array(i)$ is represented by 1 bit, 16 bits, 32 bits, 64 bits (curves). The graph depicts both measured running times (squares), and estimated running times (curves). Measured times are in executions on a single machine on Amazon's cloud; see Section 5. Estimated running times are based on Formula 1, Section 5.3.

steps on data *array* and lookup value ℓ that are encrypted using homomorphic evaluation.

(1) First, compute an array of indicator values $indicator \in \{0,1\}^m$ where $indicator_i = is\textsc{Match}(x_i, \ell)$ is 1 if-and-only-if the ith record x_i is a match to the lookup value ℓ. Note that the index of the first matching record in *array* is the index of the first positive entry in *indicator*.
(2) Second, compute an array of *prefix-sums* for this *indicator* array. Note that these prefix sum are zero on all prefices not reaching the index of the first match i^*, and are positive otherwise.
(3) Third, transform the prefix-sums array to a binary *step-function* array with value 0 at every zero prefix-sum and 1 otherwise. Note that this step-function transitions from 0 to 1 precisely on the index i^* of the first match.
(4) Fourth, transform the step-function array to a *selector* array where only the index of the first match contains 1 (and all other indices are 0). This can be easily achieved by computing the discrete derivative of the step-function array.
(5) Finally, utilize this selector array to calculate and return this first match (index and element). This is done similarly to prior-art such as PIR protocols.

Realizing this approach however is challenging: Step 2 (computing prefix-sums) has high degree if working with binary plaintext space and using standard addition circuits such as full-adders. Step 3 (zero-testing) has high degree if working over plaintext spaces larger than the array size and utilizing Fermat's Little Theorem for the zero test.

To address this challenge we propose to replace Steps (2)-(3) above with a new component we design, and to repeat our modified computation of Steps (1)-(5) in $k = o(\log^2 m)$ distinct rings GF(p) with distinct small plaintext moduli $p = O(\log^2 m)$. This results in a list of candidate solutions, one for each ring, which we call a "multi-ring sketch". The server sends the multi-ring sketch to the client, who decrypts and identifies the correct solution. Our analysis shows that the multi-ring sketch is guaranteed to contain the correct solution, and that the client can efficiently identify it.

We note that to enable the server, given only encrypted data, to execute computations for each plaintext modulus p, the client produces at the initiation of the protocol k ciphertexts for each data record and for the lookup value: one ciphertext for each plaintext modulus p. The server computes in each ring using the ciphertexts generated for that ring. We note further that we instruct the server to send the multi-ring sketch rather than the final search outcome, because we do not known how to efficiently combine ciphertexts for distinct rings on the server's side.

Elaborating on the above, the advantage of working with the small moduli p is that both the native ring summation and zero-testing are of low degree. The problem however is that the (unmodified) Steps (2)-(3) give incorrect output when computed modulo such small p. Specifically, instead of producing the aforementioned step-function, we'd get many false zeros: a zero in each entry where the prefix-sum is a multiple of p.

To explain our new component replacing Steps (2)-(3) first observe that to compute the step-function over the integers, we can first compute a tree with leaves labeled by entries of *indicator* and internal nodes labeled by sums of their children's labels, then compute each prefix-sum to be the sum of $\leq \log m$ appropriate internal nodes' labels, and obtain the step-function by applying the zero-test on the prefix-sums. Our modified Steps (2)-(3) essentially compute the same procedure as specified for the integers, but with computation modulo p. This results in a candidate for the step-function that is not necessarily correct.

We show in our analysis that the above candidate is the correct step-function as long as the following sufficient condition holds: *The tree-labels of all ancestors of the first non-zero leaf, when computed over the integers, are not multiples of p.* Moreover we show that there must exists a p satisfying the above condition in any set of $k = 1 + \log^2(m)/\log\log(m)$ distinct moduli $p_1, \ldots, p_k$ (by Pigeonhole Principle), and that we can generate k small primes $p_i = O(\log^2 m)$ (by the Prime Numbers Theorem). We conclude that executing the modified Steps (1)-(5) produces a short list of candidate solutions guaranteed to contain the correct solution.

Protocols and Papers	Efficient Client	Efficient Server	Supports unrestricted search functionality	Retrieves record	Full security	Records per hour per machine
Searchable Encryption [5, 41]	✓	N/A	✓	✓	×	$\sim 10^9$
PIR [7, 8, 14, 16, 39]	✓	✓	×	✓	✓	$\sim 10^6$
PSI [9]	✓	✓	×	×	✓	$\sim 10^6$
Secure Pattern Matching [10, 12, 23, 27, 31, 46]	×	✓	✓	~ ✓	✓	$\sim 10^3$
Folklore Secure Search	✓	×	✓	✓	✓	$\sim 10^3$
This work: Secure Search	✓	✓	✓	✓	✓	$\sim 10^6$

Table 2: *Comparison of single-round low-communication secure search protocols.* 1st column lists the compared works, followed by indications to whether: client and server are efficient (✓) or inefficient (×) in columns 2-3; the scheme supports unrestricted search functionality (✓) or requires a unique identifier (×) in column 4; the client's output is both index and record (✓), only an index i (~ ✓), or only a YES/NO answer to whether the record exists (×), in column 5; the scheme is fully secure (✓) in the sense of attaining semantic security for the data and lookup value both at rest and during search, as well as hiding the access pattern to the database, in column 6. Last column gives an order of magnitude of the number of processed records per hours per machine in reported experiments.

2 DATA SETUP AND UPLOAD

In this section we set notations (below); specify the requirements we have from the underlying FHE scheme (Section 2.1); specify our input and output representation (Section 2.2); and present the data upload protocol (Section 2.3).

Notations. For z a data record or a lookup value, we denote by $|z|$ the binary representation length of z. For integers m, we denote $[m] = \{1, \ldots, m\}$. We follow the convention that array indices start from 1, with entry i denoted $array(i)$. We use also the notation $array(b)$, for b the binary representation of i, to specify the ith element of $array$. Vectors are column vectors unless stated otherwise. We assume the array size m is a power of two (otherwise pad with zero). Logarithms are in base 2 unless explicitly stated otherwise.

2.1 Our Black Box use of FHE

Fully (or, leveled) homomorphic encryption (FHE) is used in this work in a black-box fashion: we require black-box use of the standard algorithms for FHE (key generation, encryption, decryption, and evaluation). The only requirement we make on the scheme is that we can choose as a parameter the *plaintext modulus* to be a prime number p of our choice, so that the homomorphic operations are addition and multiplications modulo p. This is the case in many of the FHE candidates, for example, [6]. For security of our scheme we require that the scheme is semantically secure.

Notations. To emphasize the plaintext modulus p we use the following notations for the standard algorithms specifying an FHE scheme E = (Gen, Enc, Dec, Eval):

- Gen is a randomized algorithm that takes a security parameter λ as input and a prime p, and outputs a public key $pk_p = (p, pk)$ and a secret key $sk_p = (p, sk)$ for plaintext modulus p, denoted:
$$(pk_p = (p, pk), sk_p = (p, sk)) \leftarrow \mathsf{Gen}(1^\lambda; p).$$
- Enc is a randomized algorithm that takes pk_p and a plaintext message msg, and outputs a ciphertext $[\![msg]\!]_p$ for plaintext modulus p, denoted:
$$[\![msg]\!]_p \leftarrow \mathsf{Enc}_{pk_p}(msg).$$
- Dec is an algorithm that takes sk_p and a ciphertext $[\![msg]\!]_p$ as input, and outputs a plaintext msg', denoted:
$$msg' \leftarrow \mathsf{Dec}_{sk_p}([\![msg]\!]_p).$$
Correctness is the standard requirement that $msg' = msg$.
- Eval is a (possibly randomized) algorithm takes pk_p, a polynomial $f(x_1, \ldots, x_t)$, and a tuple of ciphertexts $([\![m_1]\!]_p, \ldots, [\![m_t]\!]_p)$, and outputs a ciphertext c, denoted:
$$c \leftarrow \mathsf{Eval}_{pk_p}(f, [\![m_1]\!]_p, \ldots, [\![m_t]\!]_p).$$
Correctness is the requirement that decryption would return the message resulting from evaluating (modulo p) the polynomial $f()$ on inputs $m_1, \ldots, m_t$,
$\mathsf{Dec}_{sk_p}(\mathsf{Eval}_{pk_p}(f, [\![m_1]\!]_p, \ldots, [\![m_t]\!]_p)) = f(m_1, \ldots, m_t) \mod p$.
Semantic security implies that the resulting ciphertext c is computationally indistinguishable from a fresh ciphertext $[\![f(m_1, \ldots, m_t)]\!]_p$.

To encrypt messages $msg = (m_1, \ldots, m_t) \in \{0,1\}^t$, given in binary representation, we use use bit-by-bit encryption, producing a tuple of ciphertexts $[\![msg]\!]_p = ([\![m_1]\!]_p, \ldots, [\![m_t]\!]_p)$ for $[\![msg(i)]\!]_p \leftarrow \mathsf{Enc}_{pk_p}(m_i)$. We abuse notation and denote this as $[\![msg]\!]_p \leftarrow \mathsf{Enc}_{pk_p}(msg)$. Likewise, we encrypt $array = (array(1), \ldots, array(m))$ entry-by-entry, $[\![array(1)]\!]_p \leftarrow \mathsf{Enc}_{pk_p}(array(i))$. We abuse notation and denote $[\![array]\!]_p = ([\![array(1)]\!]_p, \ldots, [\![array(m)]\!]_p)$.

We note that we do not know how to efficiently combine ciphertexts generated for different moduli, say, add or multiply them. This is the reason why our server sends a multi-ring sketch rather than the search solution.

2.2 Input and Output Representation

In this section we specify the representation we use for the client's input and output (the server has no input/output).

The client's input is a data $array = (x_1, \ldots, x_m)$ and a lookup value ℓ. For simplicity of the presentation we assume that data records x_i and lookup values ℓ are given in binary representation.[3] Encrypting records and lookup values is done bit-by-bit (with a ciphertext for each bit); encrypting arrays is done element-by-element. Denote by $\mathcal{M}$ and $\mathcal{Q}$ the spaces of data records and lookup values respectively. We require that $\mathcal{M}$ consists of equal length records and $\mathcal{Q}$ consists of equal length lookup values, because our protocols leak length information; use padding as needed. The matching criterion should be compatible with the chosen input representation, namely, $isMatch\colon \mathcal{M} \times \mathcal{Q} \to \{0,1\}$. For example, when the matching criterion is a binary equality operator (see Section 5.2), we set $\mathcal{M} = \mathcal{Q} = \{0,1\}^t$ for t the bit representation length of data records.

The client's output is the first matching record (i, x_i) where

$$i = \min\{\, i \in [m] \mid isMatch(x_i, \ell) = 1\}$$

The output index i is given by its binary representation $b \in \{0,1\}^{1+\log m}$. We point out that when working modulo $p \ll m$, as in this work, we cannot return the index i as a native ring element because we'd get only its residue modulo p.

We note that despite using binary representation for input/output, we still require computing over multiple rings GF(p); See discussion in Section 1.3.

2.3 Data Upload Protocol

In the upload protocol the client generates keys, encrypts and uploads to the server its data array together with the corresponding public keys.

The client generates $k = 1 + \log^2 m / \log\log m$ keys for E, and k respective ciphertexts: a key and a ciphertext for each of the plaintext moduli $p_1, \ldots, p_k$. The primes are set to be the first k primes larger than $\log m$. See Figure 9 and the full version of this work [3].

Remarks and extensions. For simplicity of the presentation we assume the entire data *array* is uploaded in a single execution of the protocol, together with the keys generation. This can be easily modified. The keys can be generated at a separate time, possibly by a separate entity distributing the public key to the clients who will upload the data (data-sources) and to the server, and distributing the secret key to the clients who will issue search queries (search-clients). The upload can occur over time and from multiple data-sources, gradually uploading new encrypted data records for the server to append to its array of ciphertexts.

3 OUR SECURE SEARCH PROTOCOL

In this section we present our secure search protocol: A high level overview in Section 3.1; The heart of our secure search protocol, our SPiRiT sketch for first positive, in Section 3.2; Optimizations in Section 3.3; and a randomized variant of our protocol in Section 3.4. The protocol is summarized in Figure 9.

The functionality computed by our secure search protocol is as follows. The client's input is the lookup value ℓ and the tuples of secret and public keys (pk, sk) generated during the upload protocol. Here $pk = (pk_{p_1}, \ldots, pk_{p_k})$ and $sk = (sk_{p_1}, \ldots, sk_{p_k})$ correspond to the $k = o(\log^2 m)$ plaintext moduli $p_1, \ldots, p_k$. The server input is the encrypted data $[\![array]\!]$ previously uploaded by the client together with the tuple of public keys pk. The client's output is the first data record (i, x_i) matching the lookup value, i.e., $i = \min\{\, i \in [m] \mid isMatch(x_i, \ell) = 1\}$, where $i \in [m]$ is specified by its binary representation $b \in \{0,1\}^{1+\log m}$. The server has no output.

We remark that we may view the upload and search functionalities as two steps of a single reactive functionality; in this case the keys and encrypted array are states maintained by the parties.

3.1 Overview

At a high-level, the secure search protocol consists of a first step where the client sends the encrypted lookup value $[\![\ell]\!]$; a second step where the server computes and sends the encrypted muti-ring sketch (see below); and a final step where the client decrypts and identifies the correct solution.

Elaborating on the above, the encrypted lookup value is

$$[\![\ell]\!] = ([\![\ell]\!]_{p_1}, \ldots, [\![\ell]\!]_{p_k})$$

for $[\![\ell]\!]_{p_j} \leftarrow \mathsf{Enc}_{pk_j}(\ell)$. The multi-ring sketch is a list of encrypted candidate solutions:

$$\left([\![b_{p_j}]\!]_{p_j}, [\![indicator(b_{p_j})]\!]_{p_j}, [\![array(b_{p_j})]\!]_{p_j}\right) \text{ for all } j \in [k]$$

The client decrypts and outputs $(b_{p_j}, array(b_{p_j}))$ for the smallest index b_{p_j} for which $indicator(b_{p_j}) = 1$.

We elaborate on the steps executed by the server (the second step in the above high level description).

For each $j \in [k]$, the server executes the following three steps on the encrypted data $[\![array]\!]_{p_j} = ([\![x_1]\!]_{p_j}, \ldots, [\![x_m]\!]_{p_j})$ and lookup value $[\![\ell]\!]_{p_j}$ (see Figure 2):

(1) First, the server computes an encrypted array $[\![indicator]\!]_{p_j} = ([\![indicator_1]\!]_{p_j}, \ldots, [\![indicator_m]\!]_{p_j})$ of binary indicators so that

$$indicator_i = isMatch(x_i, \ell).$$

That is, $indicator_i$ is 1 if-and-only-if the data record x_i matches the lookup value ℓ, and it is computed by homomorphically evaluating the matching criterion $isMatch$:

$$[\![indicator_i]\!]_{p_j} \leftarrow \mathsf{Eval}_{pk_{p_j}}\left(isMatch, [\![x_i]\!]_{p_j}, [\![\ell]\!]_{p_j}\right).$$

(2) Second, the server applies our first positive sketch, named SPiRiT, on the encrypted array of indicators $[\![indicator]\!]_{p_j}$ and obtains a candidate for the binary representation $b_{p_j} \in \{0,1\}^{1+\log m}$ of the index $i \in [m]$ of the first positive entry of *indicator*:

$$[\![b_{p_j}]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(\mathrm{SPiRiT}_{m,p_j}, [\![indicator]\!]_{p_j}).$$

(3) Third, the server feeds the encrypted index b_{p_j} as input to a polynomial realizing the Private Information Retrieval (PIR) functionality, to obtain the encrypted record $array(b_{p_j})$ and $indicator(b_{p_j})$:

$$[\![array(b_{p_j})]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(PIR, [\![array]\!]_{p_j}, [\![b_{p_j}]\!]_{p_j}),$$

$$[\![indicator(b_{p_j})]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(PIR, [\![indicator]\!]_{p_j}, [\![b_{p_j}]\!]_{p_j}),$$

The server then sends to the client the list of candidate solutions:

$$\left([\![b_{p_j}]\!]_{p_j}, [\![indicator(b_{p_j})]\!]_{p_j}, [\![array(b_{p_j})]\!]_{p_j}\right)_{j \in [k]}.$$

[3] More generally, input can be represented in any base smaller than $\log m$. This guarantees that the digits of that representation reside in the FHE message space GF(p_i) for all moduli we use p_i, because in our protocol we set all p_i to be larger than $\log m$.

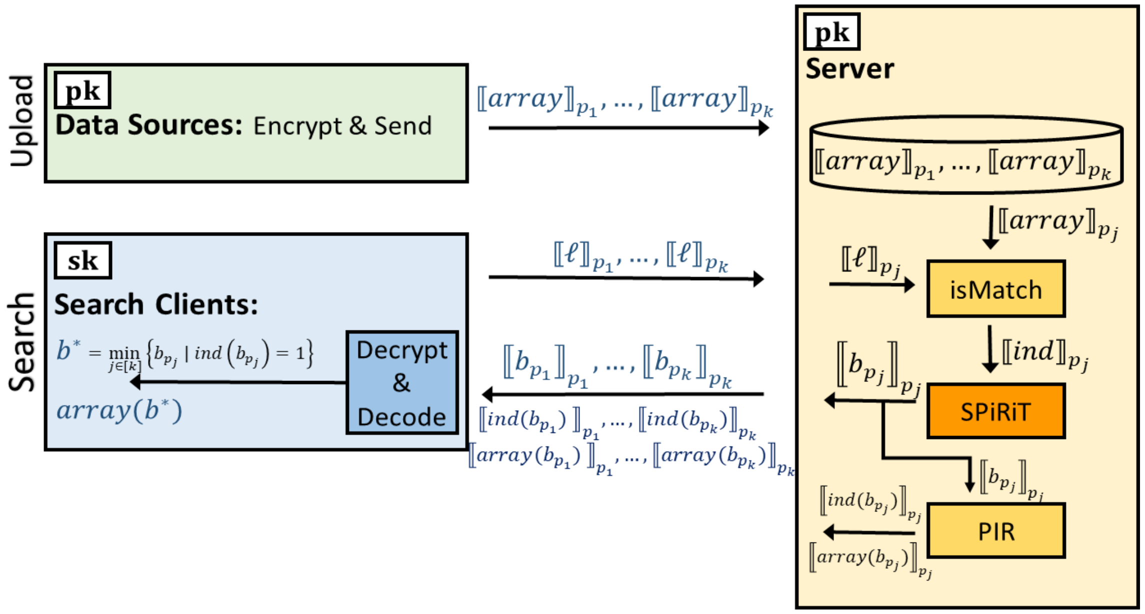
Upload
pk
Data Sources: Encrypt & Send
$[\![array]\!]_{p_1}, \dots, [\![array]\!]_{p_k}$
pk
Server
$[\![array]\!]_{p_1}, \dots, [\![array]\!]_{p_k}$
$[\![array]\!]_{p_j}$
Search
sk
Search Clients:
$b^* = \min_{j \in [k]} \{b_{p_j} \mid ind(b_{p_j}) = 1\}$
$array(b^*)$
Decrypt & Decode
$[\![\ell]\!]_{p_1}, \dots, [\![\ell]\!]_{p_k}$
$[\![\ell]\!]_{p_j}$
isMatch
$[\![ind]\!]_{p_j}$
$[\![b_{p_1}]\!]_{p_1}, \dots, [\![b_{p_k}]\!]_{p_k}$
$[\![ind(b_{p_1})]\!]_{p_1}, \dots, [\![ind(b_{p_k})]\!]_{p_k}$
$[\![array(b_{p_1})]\!]_{p_1}, \dots, [\![array(b_{p_k})]\!]_{p_k}$
$[\![b_{p_j}]\!]_{p_j}$
SPiRiT
$[\![b_{p_j}]\!]_{p_j}$
$[\![ind(b_{p_j})]\!]_{p_j}$
PIR
$[\![array(b_{p_j})]\!]_{p_j}$

Figure 2: *Data upload and secure search protocols' components (see Figure 9). The data source client holds the public key and data array; the search client holds the secret key and the lookup value ℓ; the server holds the public key. In the data-upload protocol the data-source client, encrypts the data array with respect to each modulus p_j ($j \in [k]$), and sends the resulting ciphertexts $[\![array]\!]_{p_j}$ to the server. In the secure search protocol, the client encrypts the lookup value ℓ with respect to each modulus p_j and sends resulting ciphertexts $[\![\ell]\!]_{p_j}$ to the server; The server, computes and sends to the client the encrypted multi-ring sketch ($[\![b_{p_j}]\!]_{p_j}, [\![ind(b_{p_j})]\!]_{p_j}, [\![array(b_{p_j})]\!]_{p_j}$); The client decrypts and outputs ($b_j, array(b_j)$) for the smallest candidate b_j s.t. $ind(b_j) = 1$. Elaborating on the server's computation, for each modulus p_j, the server does the following. First homomorphically evaluate the pattern matching polynomial isMatch on the encrypted lookup value $[\![\ell]\!]_{p_j}$ and each data record $[\![array(i)]\!]_{p_j}$ to obtain an encrypted indicator value $[\![ind(i)]\!]_{p_j}$. Second, homomorphically evaluate our* SPiRiT$_{m,p_j}$ *sketch for first positive on the array ind of encrypted indicators to obtain a candidate $[\![b_{p_j}]\!]_{p_j}$ for its first positive index. Third, homomorphically evaluate a PIR polynomial on the encrypted array and b_{p_j} (similarly, encrypted ind and b_{p_j}) to obtain the encryption $[\![array(b_{p_j})]\!]_{p_j}$ of the b_{p_j} entry in array (similarly, $[\![ind(b_{p_j})]\!]_{p_j}$).*

The heart of our secure search protocol is our sketch for first positive, SPiRiT, presented in Section 3.2

3.2 Details of SPiRiT Sketch for First Positive

The heart of our secure search protocol is evaluating on a binary vector $x \in \{0,1\}^m$ (specifically, x is the array *indicator*, see Section 3.1) our sketch for first positive named SPiRiT. This sketch is defined to be the composition of matrices S, P, R, T and the isPositive operator (denoted by the letter i):

$$\text{SPiRiT}_{m,p} = S \circ P \circ i \circ R \circ i \circ T \mod p$$

with operations computed modulo p. In what follows we specify the components S, P, R, T and i of SPiRiT$_{m,p}$. The constructions are with respect to computation over the integers (that we later employ for modulo p computation). Without loss of generality, we assume that m is a power of 2 (otherwise we pad the input by zero entries).

The Tree matrix $T \in \{0,1\}^{(2m-1)\times m}$ is a binary matrix that, after right multiplication by a length m vector $x = (x_1, \dots, x_m)$, returns the length $2m-1$ array data structure representation $w = (w_1, \dots, w_{2m-1})$ for the tree-representation $\mathcal{T}(x)$ of x, as defined next.

Definition 3 (Tree-representation). The *tree-representation* $\mathcal{T}(x)$ *of* $x = (x_1, \dots, x_m)$ is the full binary tree of depth $\log_2 m$ with labels assigned to nodes as follows: The label of the ith leftmost leaf is $x(i)$, for every $i \in [m]$; The label of each inner node of $\mathcal{T}(x)$ is defined recursively as the sum of the labels of its two children. We stress that summation here is over the integers (and not modulo p).

The *array data structure* for the tree-representation $\mathcal{T}(x)$ is the vector $w = (w(1), \cdots, w(2m-1))$, where $w(1)$ is the label of the root of $\mathcal{T}(x)$, and for every $j \in [m-1]$ we define $w(2j)$, $w(2j+1)$ respectively to be the labels of the left and right children of the node whose label is $w(j)$; see Figure 3. Observe that the value $w(i)$ is the sum of the labels of the leaves of the subtree rooted in the tree node corresponding to array entry $w(i)$.

The *p-flattened tree-representation of* x is obtained from $\mathcal{T}(x)$ by applying the isPositive$_p$ operator on each of its labels to transform labels that are not multiples of p to 1 (0 otherwise).

The matrix T that satisfies $w = Tx$ can be constructed by letting each row k of T corresponds to the node u in $\mathcal{T}(x)$ represented by $w(k)$, and setting this row to have 1 in every column j so that u is an ancestor of the jth leaf (0 otherwise). We point out that T is of course constructed obliviously to the input and is independent of x. Note that the last m entries of w are the entries of $x = (w(m), \cdots, w(2m-1))$.

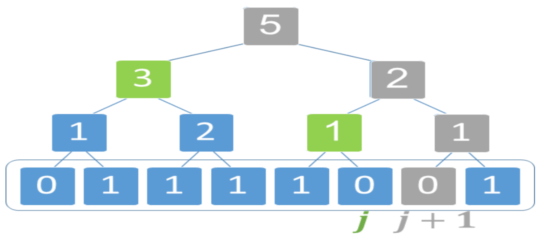

Figure 3: *The tree-representation for* $x = (0, 1, 1, 1, 1, 0, 0, 1)$ *has leaves labeled by entries of* x*, and internal nodes labeled by the sums of their children's labels. The prefix-sum of* x *up to its* $j = 6$*th entry* $(0 + 1 + 1 + 1 + 1 + 0 = 4)$ *can be computed by summing the labels of the left-siblings (depicted in green) of the ancestors (depicted in gray) of the* $(j + 1)$*th leaf* $(3 + 1 = 4)$*. The corresponding flattened prefix-sum is computed by taking the* $OR(1, 1) = 1$ *of these flattened labels* isPositive$(3) = 1$, isPositive$(1) = 3$ *(assuming here* $p \neq 3$*).*

The Roots matrix $R \in \{0, 1\}^{m\times(2m-1)}$ is a binary matrix with each row having at most $\log m$ non-zero entries that satisfies the following: its right multiplications by the tree-representation $w = (w(1), \ldots, w(2m - 1))$ of a vector $x = (x(1), \ldots, x(m))$ returns the vector $v = Rw$ of prefix-sums for x, namely, $v(j)$ is the sum of entries $x(1), x(2), \ldots, x(j)$ of x.

A naive implementation can produce these prefix-sums v using $R' \in \{0, 1\}^{m\times m}$ whose ith row $(1, \ldots, 1, 0, \ldots, 0)$ consists of i ones followed by $m - i$ zeros for every $i \in [m]$. This R' however does not satisfy our requirement for up to $\log m$ non-zero entries in each row, implying that even when applied on binary vectors the result may consist of values up to m; this in turn ruins the success of SPiRiT$_{m,p}$ when using small primes $p \ll m$.

To solve this issue, we implement R by summing labels of only $O(\log m)$ internal nodes of the tree-representation w of x. We next set up terminology to facilitate specifying these internal nodes. Consider a full binary tree with m leaves $\mathcal{T}$. We identify indices $j \in [2m - 1]$ with nodes of the tree, where the mapping is according to the standard array data structure defined above. For each node $j \in [2m - 1]$,

- Ancestors$(j) \subseteq [2m - 1]$ is the set of indices corresponding to the ancestors in the tree of node j (including j itself).
- Siblings$(j) \subseteq [2m - 1]$ is the set of indices corresponding to the left-siblings of Ancestors(j).

We are now ready to define R: Each row i of the matrix R has values 1 in all entries $j \in$ Siblings$(i + 1)$ (0 otherwise); see Figure 3. That is,

$$R(i, j) = \begin{cases} 1 & \text{if } i \in [m] \text{ and } j \in \text{Siblings}(i + 1) \\ 0 & \text{otherwise} \end{cases}$$

This matrix R satisfies the properties we required above from the roots matrix: (1) each row having at most $\log m$ non-zero entries, and (2) $v = RTx$ being the vector of prefix-sums of x, as formally stated in the lemma below.

Lemma 4 (Roots sketch). *Let* $T \in \{0, 1\}^{2m-1\times m}$ *and* $R \in \{0, 1\}^{m\times(2m-1)}$ *be the matrices defined above. Then each row of* R *has at most* $\log m$ *non-zero entries, and for every* $x = (x(1), \ldots, x(m))$*, the vector* $v = RTx$ *is a length* m *vector so that for every* $j \in [m]$*,*

$$v(j) = \sum_{k=1}^{j} x(k).$$

The Pairwise matrix $P \in \{-1, 0, 1\}^{m\times m}$ is a matrix whose right multiplication by a length m vector $u = (u(1), \ldots, u(m))$ yields the vector $t = Pu$ of pairwise differences between consecutive entries in u, i.e., $t(j) = u(j) - u(j - 1)$ for every $j \in \{2, \ldots, m\}$ and $t(1) = u(1)$. For this purpose every row $i \in \{2, \ldots, m\}$ of P has the form $(0, \ldots, -1, 1, \cdots, 0)$ with 1 appearing at its i-th entry; and the first row is $(1, 0, \ldots, 0)$. For example, if $m = 8$ and $u = (0, 0, 0, 1, 1, 1, 1, 1)$ then $t = Pu = (0, 0, 0, 1, 0, 0, 0, 0)$. More generally, if u is a binary vector that represents a step-function, then t has a single non-zero entry at the step location. Indeed, this is the usage of the Pairwise matrix in SPiRiT.

The Sketch matrix $S \in \{0, 1\}^{(1+\log m)\times m}$ is a matrix whose right multiplication by a binary vector $t = (0, \cdots, 0, 1, 0, \cdots, 0) \in \{0, 1\}^m$ with a single non-zero entry in its jth coordinate yields the binary representation $y = St \in \{0, 1\}^{1+\log m}$ of $j \in [m]$ (and $y = 0^{1+\log m}$ if t is the all zeros vector). (Note that $1 + \log m$ output bits are necessary to represent the said $m + 1$ distinct events.) This sketch matrix S can be easily implemented by setting each column $j = \{1, \ldots, m\}$ to be the binary representation of j.

The isPositive operator $i()$ for a prime p, denoted isPositive$_p(\cdot)$ (or $i(\cdot)$ in short, when p is clear from the context), gets as input an integer vector $v = (v_1, \ldots, v_{m'})$, and returns a binary vector $u \in \{0, 1\}^{m'}$ where, for every $j \in [m']$, we have $u(j) = 0$ if and only if $v(j)$ is a multiple of p. This is achieved using Fermat's Little Theorem:

$$\begin{aligned} & \text{isPositive}_p(v(1), \ldots, v(m')) \\ = \; & (v(1)^{p-1} \mod p, \ldots, v(m')^{p-1} \mod p) \end{aligned}$$

A crucial issue —that we handle via our multi-ring sketch for FHE technique— is that, unlike the matrices S, P, R, T that require from us no multiplication operations (specifically, we compute the matrix-vector multiplication using only addition operations, by summing the subset of vector entries or their negation, as specified by the non-zero entries of the matrix), the degree of the polynomial x^{p-1} that is used in isPositive$_p$ is $p - 1$ imposing the requirement that we use only small moduli p.

3.3 Optimizations

We discuss next several modifications and implementation choices we make to optimize complexity.

First, we show how to reduce the degree of SPiRiT$_{m,p}$ from $(p - 1)^2$ to $(p - 1) \log m$. Instead of applying $i \circ R$ on the vector $iT(x)$, we compute the logical OR of the corresponding (flattened) tree labels. That is, after computing the tree-representation and flattening its labels to binary values using the isPositive$_p$ operator, instead of summing up the labels indicated by the roots matrix R and applying isPositive$_p$ on the sum, we compute the logical OR of these (flattened) labels. Note that the sum of the (flattened) labels is zero if-and-only-if their OR is zero, so this change does not affect correctness. Yet, the degree reduces from $(p - 1)^2$ (two composed evaluations of isPositive$_p$ to $(p - 1) \log m$ (evaluation of isPositive$_p$ composed with the evaluation of OR of up to $\log m$ binary variables).

Second, we show how to reduce the the overall number of multiplications. This is useful because homomorphic additions are much less costly than homomorphic multiplications with the current FHE candidates. We achieve the reduction by replacing the matrix-vector multiplications by computing sums of the corresponding vectors entries. This is possible because all matrices are plaintext matrices and with entries accepting values in $\{-1, 0, 1\}$.

Third, we show how to reduce the degree blow-up due to applying PIR from, reducing it from the multiplicative factor $O(\log m)$ to an additive increament by 1. Instead of using a black-box PIR polynomial to retrieved the values $array(b_p)$ and $indicator(b_p)$, we use the selector vector $t_p \leftarrow P \circ i \circ R \circ i \circ T(indicator)$ that was already computed as part of our SPiRiT$_{m,p}$ computation. Specifically, the selector vector is a length m binary vector with a single non-zero entry at index b_p. We use it to replace the equality testing in PIR: instead of testing for each array index whether it is equal to the given index b_p, we simply plug in the pre-computed value in t_p. This saves a $\log m$ factor in the degree – the degree of the equality test of the binary representations of array indices. Instead, this optimized PIR computation increases the degree of computing t_p only additively with a degree increment by 1.

3.4 Monte Carlo Secure Search Protocol

In this section we briefly discuss a variant of our (deterministic) secure search protocol (see above and in Figure 9) that introduces a noticeable probability of error for saving a factor $O(\log^2(m)/\log\log(m))$ in the overall number of multiplications.

The randomized protocol is similar to our deterministic protocol except for working over a single $j \in [k]$ chosen uniformly at random, instead of repeating over all $j \in [k]$ in our deterministic protocol. We set the parameter k to control the success probability. Specifically, the success probability is set to $1 - \delta$ by taking $k = \frac{1}{\delta} \cdot \frac{\log^2 m}{\log\log m}$. See details of the success analysis in the full version of this work [3].

4 ANALYSIS

In this section we analyze our secure search protocol, addressing correctness in section 4.3, complexity in section 4.4, and security in section 4.5. We begin however with analyzing the central component of our protocol, the SPiRiT sketch for first positive, in sections 4.1-4.2.

4.1 SPiRiT: Correctness Analysis

In this section we analyze the correctness of SPiRiT$_{m,p}$. Specifically, we present a condition on p and x that is sufficient to guarantee that SPiRiT$_{m,p}(x)$ returns the correct output, that is, SPiRiT$_{m,p}(x)$ returns the first positive index of x.

We begin with formally defining the first positive index and the sufficient condition.

Definition 5 (First positive index). Let $x = (x_1, \ldots, x_m) \in [0, \infty)^m$ be a vector of m non-negative integers. The *first positive index of* x is the smallest index $i^* \in [m]$ satisfying that $x_{i^*} > 0$, or $i^* = 0$ if $x = (0, \ldots, 0)$.

The sufficient condition for SPiRiT$_{m,p}(x)$ to succeed in returning the first positive index i^* of x is that, the non-zero labels of the ancestors of leaf i^* in the tree-representation of x (see definition 3) are labels non-multiples of p. In other words, for these ancestors of i^*, their label is zero in the tree-representation if-and-only-if it is zero in the p-flattened tree-representation.

Definition 6 (Good p). We call a prime number p *good* for $x \in \{0, 1\}^m$ if for $i^* \in [m] \cup \{0\}$ the first positive index of x the following condition holds: for all nodes $j \in \text{Ancestors}(i^*)$ with non-zero labels in the tree-representation of x (i.e., $Tx(j) \neq 0$), their label $Tx(j)$ is not a multiple of p.

Lemma 7 (Sufficient condition for SPiRiT$_{m,p}$ correctness). *Let m be a power of 2, $p > \log m$ a prime, $x \in \{0, 1\}^m$ and $i^* \in [m] \cup \{0\}$ the first positive index of x. If p is good for x, then $b_p \leftarrow$ SPiRiT$_{m,p}(x)$ is the binary representation $b^* \in \{0, 1\}^{1+\log m}$ of i^*.*

Proof. The case $i^* = 0$ is trivial: it is immediate to verify that when evaluated on $x = (0, \ldots, 0)$, $b_p \leftarrow$ SPiRiT$_{m,p}(x)$ is the zero vector $b_p = (0, \ldots, 0)$, as the matrix-vector products and application of isPositive$_p$ operator all evaluate to 0.

For the case $i^* \in [m]$, denote

$$u = i \circ R \circ i \circ T(x).$$

It is easy to verify that if $u = (0, \ldots, 0, 1, \ldots, 1)$ is a step function with i^* the index of its first non-zero entry, then SPu is the binary representation of i^*; see Claim 8. We next show that indeed this is the form of u, when p is good. Showing that $u(1) = \ldots = u(i^* - 1) = 0$ is simple; see Claim 9. The challenging part is to show that, if p is good, then $u(i^*) = \ldots = u(m) = 1$; see Claim 10. Put together we conclude that, if p is good, then b_p is the binary representation i^*.

See the proofs for the following claims in the full version of this work [3].

Claim 8. *If $u = (0, \ldots, 0, 1 \ldots, 1)$ is a binary vector accepting values 1 starting from its i^*th entry, then $(SP \cdot u \mod p)$ is the binary representation of i^*.*

Claim 9. $u(1) = \ldots = u(i^* - 1) = 0$.

Claim 10. *If p is good, then $u(i^*) = \ldots = u(m) = 1$.*

□

4.2 SPiRiT: Complexity Analysis

In this section we analyze the complexity of SPiRiT$_{m,p}$.

Lemma 11 (SPiRiT$_{m,p}$ complexity). *The polynomial realizing SPiRiT$_{m,p}$: $\{0, 1\}^m \to \{0, 1\}^{1+\log m}$ has degree $(p - 1)^2$ and total number of multiplications $O(m \log\log m)$.*

The polynomial realizing the optimized version of SPiRiT$_{m,p}$ *(see Section 3.3) has degree $(p - 1)\log m$ and total number of multiplications $O(m \log m)$.*

Proof. Computing the matrix-vector products when multiplying by S, P, R, T can be done using only addition/subtraction operations and no multiplications whatsoever (see Section 3.3). So this part of the computation adds nothing to the degree or to the total number of multiplication.

The degree in each applications of the isPositive$_p$ operator is $p-1$, and since we compute it twice the total degree is $(p-1)^2$ (where we use here the standard fact that degrees multiply when composing polynomials).

To analyze the overall number of multiplications, observe that computing isPositive$_p$ for a single entry, using repeated squaring, requires at most $2\log p$ multiplications. Since we apply isPositive$_p$ on a total of $(2m-1)+m$ entries, we get that the total number of multiplications is $O(m\log p)$. Assigning $p=O(\log^2 m)$ we get that the number of multiplications is $O(m\log\log m)$

To reduce the degree to $(p-1)\log m$ we have introduced the following optimization in evaluating $u=i\circ R(w')$ on $w'=i\circ T(indicator)$: Instead of evaluating the degree $p-1$ polynomial that first sums up labels in the p-flattened tree-representation as specified by R, and then reduces the sum to a binary value by applying on it isPositive$_p$, we directly compute the *OR* of these labels. The latter gives an identical result to the former, but with a polynomial of degree proportional to the number of roots which is upper bounded by $\log m$ (instead of degree $p-1$). The overall degree therefore is the product of the degree $p-1$ for applying the isPositive$_p$ operator to flatten the label in the tree-representation, and the degree at most $\log m$ of computing the OR over the labels specified by R.

We analyze the overall number of multiplications for the above degree optimized version. There are $2\log p$ multiplications for applying isPositive$_p$ on each the labels of each of the $2m-1$ nodes in the tree-representation, plus up to $\log m$ multiplications for computing the value of each of the m entries of u using the *OR* as specified above. The total number of multiplications is therefore upper bounded by $O(m(\log m+\log p))$. Assigning $p=O(\log^2 m)$ we get that the number of multiplications is $O(m\log m)$. □

4.3 Secure Search: Correctness Analysis

We analyze the correctness of our secure search protocol (see Section 3 and Figure 9).

Theorem 12 (Correctness). *The secure search protocol (see Figure 9), when executed with parameters and input as specified there by semi-honest client and server who follow the protocol, satisfies the following. The client's output is*

$$(b, array(b))$$

for $b\in\{0,1\}^{1+\log m}$ the binary representation of the index i of the first match for ℓ in $array$:

$$i=\min\{\,i\in[m]\mid isMatch(array(i),\ell)=1\}$$

($i=0$ if no match exist). The server has no output.

Proof. The proof follows from Lemma 7 and the claim below; see details in the full version of this work [3]. □

Claim 13. *There exists a prime $p\in\mathcal{P}=\{p_1,\ldots,p_k\}$ that is good for $indicator$.*

Proof. The proof follows from the Pigeonhole Principle; see details in the full version of this work [3]. □

4.4 Secure Search: Complexity Analysis

We analyze the complexity of our secure search protocol (see Section 3 and Figure 9).

Theorem 14 (Complexity). *The secure search protocol (see Figure 9), when executed with parameter m for the number of data records, satisfies the following for $k=1+\frac{\log^2 m}{\log\log m}$.*

- *The protocol has a single round of communication, with communication complexity proportional to k encryptions of the lookup value ℓ and k encryptions of the search outcome (i,x_i).*
- *The client's running time is proportional to the time to compute k encryptions for the lookup value ℓ, and k decryptions of the search outcome (i,x_i).*
- *The server evaluates k polynomials (possibly, in parallel), each of degree $O(\log^3 m)\cdot d$ and $O(m(s+\log m))$ overall number of multiplications, for d and s the degree and number of multiplications for computing isMatch on a single values pair in $\mathcal{M}\times\mathcal{Q}$.*

Proof. The proof follows from Lemma 11 together with the claim below; see details in the full version of this work [3]. □

Claim 15. *For $m,\mathcal{P}$ as in the secure search protocol all primes $p\in\mathcal{P}$ have magnitude $p=O(\log^2 m)$.*

Proof. The proof follows from the Prime Number Theorem; see details in the full version of this work [3]. □

4.5 Secure Search: Security Analysis

The upload-and-search functionality (see Figure 4) involves two parties, called client and server, where the client's input is a data array and lookup value, the client's output is first data record that matches the lookup value (index and record), the server has no input or output except for the shared parameters (the security parameter λ, FHE scheme E, message and query spaces $\mathcal{M},\mathcal{Q}$, number of data records m, set of primes $\mathcal{P}=\{p_1,\ldots,p_k\}$, and the matching criterion $isMatch\colon\mathcal{M}\times\mathcal{Q}\to\{0,1\}$).

We show that our upload and search protocol securely realizes the above functionality against semi-honest adversaries controlling the server.

Theorem 16 (Security). *The upload and search protocol (see Figure 9) securely realizes the upload-and-search functionality (see Figure 4) against a semi-honest adversary controlling the server, assuming the underlying encryption E is semantically secure.*

Proof. The security proof is straightforwards; see details in the full version of this work [3]. □

5 SYSTEM AND EXPERIMENTAL RESULTS

In this section we describe the secure search system we implemented using the secure search protocol presented in this paper. To our knowledge, this is the first implementation of such an FHE based secure search system.

We implemented our protocol in an open source library based on the HElib library [24] implementation for the Brakerski-Gentry-Vaikuntanthan's FHE scheme [6] together with the Single Instruction Multiple Data (SIMD) optimization of Smart-Vercauteren [40]

Participants: The functionality involves two parties called client and server.

Parameters (Shared Input): Both parties receive the security parameter λ, an FHE scheme $\mathsf{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ with messages space $\mathcal{M}$ (of equal length messages) and query space $\mathcal{Q} \subseteq \mathcal{M}$; a number m of data records (w.l.o.g m is a power of two); a set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of the smallest $k = 1 + \log^2(m)/\log\log(m)$ primes larger than $\log m$; and a polynomial realizing the matching criterion $is\textsc{Match}\colon \mathcal{M} \times \mathcal{Q} \to \{0,1\}$.

Inputs: The client's input is a data array $array = (x_1, \ldots, x_m) \in \mathcal{M}^m$ and a lookup value $\ell \in \mathcal{Q}$. The server has no input.

Outputs: The client's output is the pair (i, x_i) s.t. $i = \min\{\, i \in [m] \mid is\textsc{Match}(x_i, \ell) = 1\}$ (and $i = 0$ if no match exists). The server has no output.

Figure 4: Upload-and-Search Functionality.

and Gentry-Halevi-Smart [18]. We ran experiments on Amazon's AWS EC2 cloud occupying up to 100 processors. Our experiments show that we can securely retrieve records from a database, where *both database and query are encrypted with FHE*, achieving a rate of *searching in millions of database records in less than an hour* on a single 64-cores machine. Moreover, our experiments show that the running time reduces near-linearly with the number of cores. So, for example, we can achieve a rate of *searching in a billion of database records in roughly two hour* using 100 such machines. The system is fully open source, and all our experiments are reproducible. For details of our system and experimental results see Sections 5.1-5.4.

5.1 System

System Overview. We implemented our secure search protocol into a system that maintains an encrypted database that is stored on Amazon Elastic Compute Cloud (EC2) provided by Amazon Web Services (AWS). The system gets from the client an encrypted lookup value ℓ to search for, and a column name *array* in a database table of length m. Encrypting the column name is optional. The encryption is computed on the client's side and can be decrypted using a secret key that is unknown to the server. The client can send the search request through a web-browser, that can run e.g. from a smart-phone or a laptop. The system then runs on the cloud our secure search algorithm (Step 3 in Figure 9), and returns to the client a short list of encrypted candidates for the first match for ℓ is *array* (the multi-ring sketch). The web browser then decrypts this candidates list in the client's machine and uses it to compute the smallest index i^* in *array* that contains ℓ ($i^* = 0$ if ℓ is not in *array*). As expected by the analysis, the decoding and decryption running time on the client side is very fast (less than a second) and practically all the time is spent on the server's side (cloud). Database updates can be maintained between search calls, and support multiple users that hold the public key. Search queries can be issued by multiple clients that hold the public key. The search outcome can be decrypted and decoded by any client (or multiple clients) holding the secret key.

Hardware. Our system is generic but in this section we discuss how we evaluated it with server running on Amazon's AWS cloud, and client running on a home computer. For the server we use one of the standard suggested grids of EC2 **x1.32xlarge** servers. Such a server has 128 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) cores (that are also common in standard laptop), 1,952 GigaByte of RAM, and $2 \times 1.9TB$ SSD disk. For the client we use a personal computer with Intel(R) Core(TM) i7-4790 CPU at 3.60GHz, 4 cores, and 16GB RAM.

Open Software. The algorithms were implemented in $C{+}{+}$. HElib library [24] was used for the underlying FHE scheme, including its usage of SIMD (Single Instruction Multiple Data) technique. The source of our system is open under the GNU v3 license that can be found in [1].

Security. Our system and all the experiments below use a security key of 80 bits of security, as standard in works computing on FHE encrypted data. This setting can be easily changed by the client.

5.2 Experiments

Data. We ran the system on a lookup value ℓ and a length m *array*, where values ℓ and $array(1), \ldots, array(m)$ are in binary representation of length t bits. We ran experiments on binary representation lengths tested for both the case $t = 1$ and $t = 64$ bits, and on a roughly doubling number of records m starting with $m = 90,048$ and reaching to $m = 41,408,640$ records for the case $t = 64$ and $m = 511,697,280$ for the case $t = 1$. In case $t = 1$, *array* is a vector of all zeros except for a random index. In case $t = 64$, *array* is a vector of m random 64-bits entries.

We note that experiments for the case $t = 1$ are aimed to facilitate estimating run-time with other *is*Match implementations (e.g., similarity search), by giving the run-time for processing the binary indicator vector (while excluding the time to compute *is*Match). The reader can then add to the $t = 1$ run-time the time to compute $is\textsc{Match}(x_i, \ell)$ with her preferred *is*Match (for appropriate setting of HElib parameter L).

Let us elaborate on the choice of the number of records m for our experiments. The values m were determined by taking doubling numbers of ciphertexts n and letting m be $n \cdot SIMD \cdot CORE$ for $SIMD$ the number of messages packed in each ciphertext and $CORES = 64$ the number of cores in the machine on which we ran our experiments. Although the machine we used had 128 cores we used only 64 of them, specifically those with even indices. Using more cores did not improve the running time. Specifically, when running on the $(2i)$-th core and the $(2i + 1)$ core, each core seemed to run in half of the capacity. The SIMD parameter is determined by the HElib context parameters of number of levels L and prime p. The $SIMD$ factor we used was not very high, ranging from 122 to 444; in particular $SIMD = 122$ (respectively, 158) on our high-end result on number of records $m \approx 500,000,000$ and $t = 1$ (respectively, $m \approx 40,000,000$ and $t = 64$).

We remark that we did not attempt to optimized the SIMD factor: by slight modification of L, p it is often possible to reach much higher SIMD factors, say, 1000 or 2000.

Concrete choice of *is*MATCH in our experiments. The concrete implementation for *is*MATCH used in our experiments is the equality test, returning $is\text{MATCH}(array(i), \ell) = 1$ if-and-only-if $array(i) = \ell$. We assume the input is given, as standard, in binary representation, and where encryption is bit-by-bit. Denoting by p the plaintext modulus we use and by $a, b \in \{0,1\}^t$ the patterns whose equality we wish to determine, the equality test we implemented is defined by:

$$is\text{EQUAL}_t\,(a, b) = \prod_{j \in [t]} \left(1 - (a_j - b_j)^2\right) \mod p.$$

The degree of this test is $2t$. This degree is independent from the number of entries in the data *array*, and depends only on the binary representation length t for each entry. This should be interpreted as one possible example for a pattern matching polynomial that can be plugged into our secure search protocol.

We remark that the above equality test differs from the standard equality test used when the plaintext modulus is $p = 2$ (specifically, $\prod_{j \in [t]}(1 + a_j + b_j) \mod 2$), as is necessitated by working with higher modulus $p > 2$.

Experiments. We ran our secure search algorithm (Step 3 in Figure9), running on the server the probabilistic version in which a single p is chosen; see Section 3.4. The experiments address the data as specified above (binary representation length for data elements $t \in \{1, 64\}$, and number of records m ranging approximate from 10^5 to $0.5 \cdot 10^9$). In case $t = 64$ the server first compares ℓ to each entry of *array* by calling $is\text{EQUAL}_t$ for producing the vector *indicator* $\in \{0,1\}^m$ on which the server applies the SPiRiT sketch for first positive to return the index of the first match for ℓ in *array*. In case $t = 1$ the first above step is degenerated, and the experiment measures performance of $\text{SPiRiT}_{m,p}$ sketch for first positive.

5.3 Formula for Concrete Running Time

When we move from theory to implementation it is useful to have running time estimation with concrete numbers rather than the $O()$ notation; in this section we provide such a formula (see Formula 1).

The formula we provide takes into account the following additional factors, beyond the algorithm: First, the acceleration gained by employing the Smart-Vercauteren [40] SIMD (Single Instruction Multiple Data) optimization that enables packing multiple plaintext messages in a single ciphertext. Second, the acceleration gained by running the algorithm on a multi-core hardware, where we distribute the work in a "MapReduce-like" fashion with processors searching in disjoint subsets of the data *array*.

Our formula for the concrete running time on data *array* of m records, each record of length t bits, is given in Eq 1,

$$\begin{aligned} T(m,t) \quad &= \quad n \cdot 2t \cdot MUL + 2n(1 + \lceil \log_2 n \rceil) \cdot ADD && (1)\\ &\quad\quad +2n \cdot IsPOSITIVE && (2) \end{aligned}$$

where

- $n = \dfrac{m}{CORES \cdot SIMD}$
- $CORES$ is the number of core processors that work in parallel.
- $SIMD$ is the number of plaintext messages that are packed in each single ciphertext. This $SIMD$ factor is a function of the ring size $p = O(\log^2 n)$ and $L = \log(d + \log s) = O(\log_2 \log_2 n)$ for d, s upper bounds on the degree and size of evaluated polynomials; in HELib, this parameter can be read by calling EncryptedArray::size(), see [25].
- ADD, MUL, and $IsPOSITIVE$ are the times for computing a single addition, multiplication, and the ISPOSITIVE_p operator, respectively, in the context of parameters p and L.

For details on how we derive this formula see the full version of this work [3].

5.4 Results

Our experimental results on a single machine on the cloud are summarized in Table 3 and Figure 1; and results on up to 100 machines in Figure 6.

The client's running time (i.e., the time for encryption, decryption and decoding) was very fast: under 30ms for the randomized variant of our protocol (see Section 3.4), and under a second for the deterministic variant (see Section 3); experiments are on our personal computer (see Hardware specification in Section 5.1).

Elaborating on the former, the time for processing a single ciphertext on a single core was under 30ms, so this is the client's time in our randomized variant with no amplification. For our deterministic protocol, the number of parallel ciphertexts k (the length of the list of candidates) was under 110 in all our experiments. For example, for $m = 511,697,280$ records we had $SIMD = 122$ implying the records are packed in $n' = m/SIMD = 4,194,240$ ciphertexts and so $k = 1 + \log_2^2(n')/\log_2\log_2(n') = 1 + 22^2/\log_2 22 < 110$. Partitioning the work between the 4 cores on the client's computer, leads at most 28 ciphertexts to be processed per core, and an overall time of essentially $28 \times 30ms = 840ms$. Namely, the client's running time was under a second for both the randomized and the deterministic variants of our secure search protocol. So the server's time is essentially the overall running time of the protocol.

The server's running time on a single machine (with hardware as specified in Section 5.1) depends on the size parameters m and t, but not on the actual entries' content in *array* or the desired lookup value ℓ. This is because the server computes on encrypted data, and is therefore oblivious to the data content. Our experiments demonstrate the following server's running times on a single machine; see Table 3 and Figures 1,5 for details.

- For 64-bits records ($t = 64$), our system can search in a data *array* of approximately $m = 100,000$ records in a minute. Similarly, our system can search in approximately $m = 4,000,000$ ($m = 40,000,000$) records in an hour (a day).
- For 1-bits records ($t = 1$), i.e., when isolating the time for our SPiRiT sketch for first positive, our system can search in a data *array* of approximately $m = 100,000$ records in less than a second. Similarly, our system can search in approximately $m = 40,000,000$ ($m = 500,000,000$) records in an hour (a day).

number of records m	SPiRiT time	Search time
90,048	0.7 sec	1 min
192,960	2 sec	2 min
196,416	14 sec	7 min
399,168	33 sec	14 min
2,048,256	2 min	33 min
4,112,640	4 min	66 min
14,520,576	8 min	2.3 hours
19,641,600	17 min	4.6 hours
20,699,264	35 min	14.4 hours
41,408,640	1.25 hours	26.7 hours
63,955,328	2.6 hours	
127,918,464	5.5 hours	
255,844,736	11.7 hours	
511,697,280	22.5 hours	

Table 3: *Server's running time on encrypted database and encrypted lookup value, as measured on a single 64-cores machine on Amazon's cloud, with growing database array size (left column). Middle column shows the running times for* $\text{SPiRiT}_{m,p}$ *sketch for first positive on length m binary array ($t = 1$). Right column shows the running times for secure search in a length m array of 64-bits records ($t = 64$).*

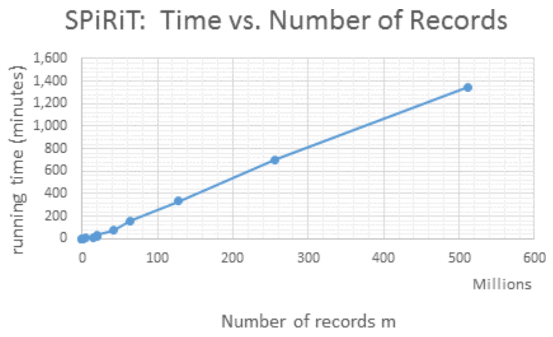

Figure 5: $\text{SPiRiT}_{m,p}$ *running time as a function of the number of records. Number of records ranged from* $m = 90,047$ *to* $m = 511,697,280$ *in roughly doubling values; measured running times started at under a second (0.7sec) and reached up to under a day (22.5 hours).*

Scalability: Server's running time on parallel machines. In a parallel computation on multiple machines we can have machines that are almost independent ("embarrassingly parallel" [43]). To use s servers we split the data evenly among them, where each server stores and searches n/s of the entries. The split is in consecutive chunks (elements $1, \ldots, n/s$ for first server, elements $(n/s)+1, \ldots, 2n/s$ for second server, and so forth). The output is then taken to be the output of the first server who returned a non empty output $i^* \neq 0$. The running time on each machine was almost identical (including the non-smooth steps; see below) and the running time decreases linearly when we add more machines (cores) to the cloud, as expected. So, for example, using 100 machines we could search 1,000,000,000 (a billion) 64-bits records in roughly two hours; see Figure 6.

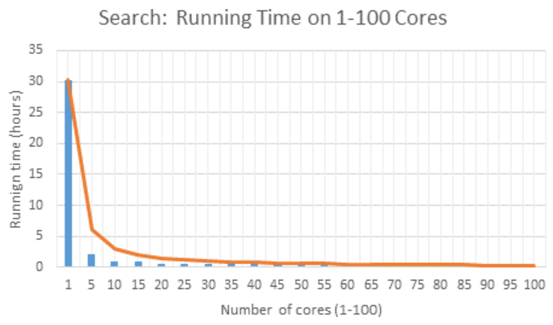

Figure 6: *Server's running time (hours) for computing Secure Search on a billion records using 1-100 cores (bars); compared with running time reduction by a factor of 1/#cores (curve). Note that we gain more than a factor 1/#cores speedup, because splitting the data decreases the overall degree.*

Storage, I/O, and RAM. Our experiments with HElib show that a single ciphertext takes about 10KB to store, for a total of 3TByte for 300 million database entries when $t = 1$, or 6.4TByte for 10 million entries entries when $t = 64$. With SSD disk prices getting lower, these amount of data are feasible to be stored on SSD, which are significantly faster than regular disks. Also, since reading data from a disk requires very little CPU, data can be read from multiple threads from multiple disks in parallel and be made ready for a primary CPU intensive thread. We also measured the RAM requirements. During secure search evaluation RAM requirements were a few GigaBytes, typically not exceeding 3Gb; this is because we uploaded ciphertexts from drive as needed, never requiring to simultaneously hold many ciphertexts in RAM. For generating the evaluation key in various contexts of the multiplicative depth L and the plaintext modulus p we saw that RAM requirements were typically around 4Gb, with some peaks reaching towards 8Gb; see Figure 7.

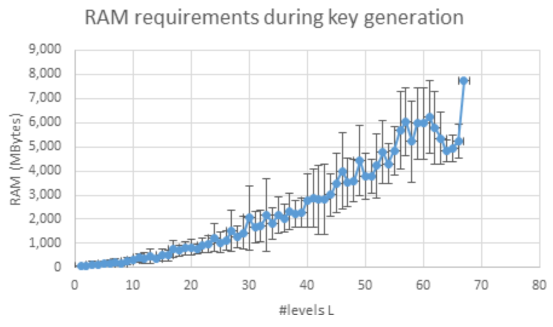

Figure 7: *RAM requirements for generating the public evaluation key as a function of the multiplicative depth L (x-axis), averaged over the plaintext modulus values* $p = 5, 7, 11, 13, 17, 19, 23, 29, 257, 1249, 4241, 16519, 64091$ *(curve and std error bars); demonstrating that RAM requirements are typically around 4Gb, with peaks reaching towards 8Gb.*

Comparison to our theoretical analysis. Theorem 14 show that the degree of our secure search protocol is the degree $O(\log^3 m)$ of $\mathrm{SPiRiT}_{m,p}$ times the degree d of the used pattern matching subroutine. The multiplicative depth of SPiRiT (i.e., the logarithm of its degree) is therefore doubly logarithmic in the number of records: $3 \log \log m + d + O(1)$. This is demonstrated by the multiplicative depth as measured in our experiments; see Figure 8.

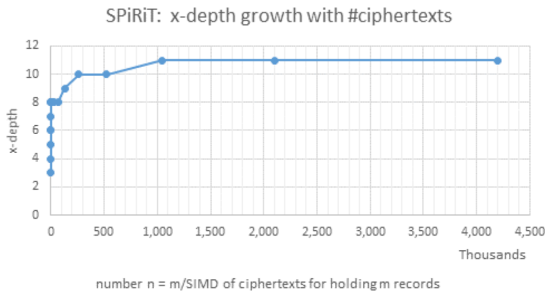

Figure 8: *The graph shows the multiplicative depth of* $\mathrm{SPiRiT}_{m,p}$ *(equivalently: the based* 2 *logarithm of the degree, or the number of levels parameter L in HElib), as a function of the number of ciphertexts* $n = m/(SIMD \cdot CORES)$ *for holding the data array (for m the number of records). By our analysis we expect the depth to be* $3 \log \log n + O(1)$.

Why the curves are not smooth? Each of the curves in Figures 1 has 4–5 non-continuous increasing steps. These are not artifacts or noise. They occur whenever there is an increase in the number of primes $|\mathcal{P}| = 1 + \log^2 m / \log \log m$ used in our secure search protocol. As the cardinality $|\mathcal{P}|$ grows, so grow the primes $p \in \mathcal{P}$, corresponding to the ring modulus p where we compute the $\mathrm{SPiRiT}_{m,p}$ sketch. The increase in p in turn increases the depth of the polynomial realizing $\textsc{isPositive}_p$, and consequently the overall server's running time.

6 CONCLUSIONS AND DISCUSSION

In this work we present the first secure search protocol on FHE encrypted lookup value and searched data, achieving all the following. (1) Efficient client: running time polynomial in computing $|input| + |output|$ encryptions and decryptions; (2) Efficient server: evaluating polynomials of degree poly-logarithmic in the number of searched data records times the degree of the used *is*Match polynomial; (3) Efficient communication: single round, communication volume grows only with input and output sizes, regardless on the complexity of the outsources search computation; (4) Unrestricted search functionality: no unique identifier requirement, and modular use of matching criteria; (5) Semantic security for both lookup value and data, both at rest and during search, against semi-honest adversary controlling the server.

We implemented our protocol in an open source library based on HElib, and ran experiments on Amazon's AWS EC2 cloud. Our experiments show that we can search in a rate of millions of records per hour per machine.

We note that while we focused here on the search functionality of returning the first match, this can be easily extended in various ways. For example, extensions to dynamic management of encrypted data with insert/update/delete functionalities on top of search are easy to incorporate (with additional interaction). Likewise, our solution for returning the first match enables the client to retrieve the matching records one-by-one (as in SQL FETCH_FIRST and FETCH_NEXT) using additional interaction, as well as for returning all matching in a single round protocol (while leaking an upper bound on the number of matches), as studied in a follow-up work [2].

Furthermore, while achieving a 1-round protocol was a main goal that we achieved via our FHE-based construction (motivated for settings where communication is one-time, unstable or of high latency, as well as for exploring and expanding the infamous running time limits of FHE-based search by order of magnitudes), our secure search protocol can be used as a component also in multi-round protocols. This is since our low degree search polynomial is a general purpose tool that may be useful beyond the context of FHE.

7 ACKNOWLEDGMENT

We thank Shai Halevi, Craig Gentry, Shafi Goldwasser and Vinod Vaikuntanathan as well as the anonymous referees for helpful discussions and comments. The second author is grateful for the support of the Simons Foundation for part of this work that was done while he was visiting the Simons Institute for the Theory of Computing. Part of this work was sponsored by the Center for Cyber Law & Policy in the University of Haifa.

REFERENCES

[1] Adi Akavia, Dan Feldman, and Hayim Shaul. 2017. SearchLib: Open Library for FHE search. (2017). https://github.com/HayimShaul/liphe

[2] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Database Queries in the Cloud: Homomorphic Encryption meets Coresets. (2018). sumbitted.

[3] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search via Multi-Ring Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2018/245. (2018). https://eprint.iacr.org/2018/245.

[4] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. 2013. Private Database Queries Using Somewhat Homomorphic Encryption. In *Applied Cryptography and Network Security*, Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–118.

[5] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2015. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)* 47, 2 (2015), 18.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science (FOCS '11)*. IEEE Computer Society, Washington, DC, USA, 97–106. https://doi.org/10.1109/FOCS.2011.12

[8] Gizem S. Çetin, Wei Dai, Yarkin Doröz, William J. Martin, and Berk Sunar. 2016. Blind Web Search: How far are we from a privacy preserving search engine? *IACR Cryptology ePrint Archive* 2016 (2016), 801. http://eprint.iacr.org/2016/801

[9] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 1243–1255. https://doi.org/10.1145/3133956.3134061

[10] 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. Information Forensics and Security* 11, 1 (2016), 188–199. https://doi.org/10.1109/TIFS.2015.2483486

[11] Jung Hee Cheon, Miran Kim, and Kristin E Lauter. 2015. Homomorphic Computation of Edit Distance.. In *Financial Cryptography Workshops*. 194–212.

[12] Jung Hee Cheon, Miran Kim, and Kristin E Lauter. 2015. Homomorphic Computation of Edit Distance.. In *Financial Cryptography Workshops*. 194–212.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 41–50.

[14] Yarkin Doröz, Berk Sunar, and Ghaith Hammouri. 2014. Bandwidth Efficient PIR from NTRU. In *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers*. 195–207. https://doi.org/10.1007/978-3-662-44774-1_16

[15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 201–210. http://dl.acm.org/citation.cfm?id=3045390.3045413

[16] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. AAI3382729.

[17] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[18] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology–CRYPTO 2012*. Springer, 850–867.

[19] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[20] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[21] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *Proceedings of the 15th International Conference on Information Security and Cryptology (ICISC'12)*. Springer-Verlag, Berlin, Heidelberg, 1–21. https://doi.org/10.1007/978-3-642-37682-5_1

[22] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. 2010. Multi-query Computationally-Private Information Retrieval with Constant Communication Rate. In *Public Key Cryptography – PKC 2010*, Phong Q. Nguyen and David Pointcheval (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–123.

[23] Tang H, X Jiang, X Wang, S Wang, H Sofia, D Fox, K Lauter, B Malin, A Telenti, L Xiong, and L Ohno-Machado. 2016. Protecting genomic data analytics in the cloud: state of the art and opportunities. *BMC Med Genomics* 9(1) (Oct 2016), 63.

[24] Shai Halevi and Victor Shoup. 2013. HElib - An implementation of homomorphic encryption. https://github.com/shaih/HElib/. (2013).

[25] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *34rd Annual International Cryptology Conference, CRYPTO 2014*. Springer Verlag.

[26] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. 2016. Better Security for Queries on Encrypted Databases. *IACR Cryptology ePrint Archive* 2016 (2016), 470.

[27] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. 2017. Private Compound Wildcard Queries using Fully Homomorphic Encryption. *IEEE Transactions on Dependable and Secure Computing* (2017).

[28] Fernando Krell, Gabriela Ciocarlie, Ashish Gehani, and Mariana Raykova. 2017. Low-Leakage Secure Search for Boolean Expressions. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 397–413.

[29] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. 2016. Embark: Securely Outsourcing Middleboxes to the Cloud.. In *NSDI*. 255–273.

[30] Kristin E Lauter, Adriana López-Alt, and Michael Naehrig. 2014. Private Computation on Encrypted Genomic Data. *LATINCRYPT* 8895 (2014), 3–27.

[31] Kristin E. Lauter, Adriana López-Alt, and Michael Naehrig. 2015. Private Computation on Encrypted Genomic Data. *IACR Cryptology ePrint Archive* 2015 (2015), 133. http://eprint.iacr.org/2015/133

[32] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma. 2016. Using Fully Homomorphic Encryption for Statistical Analysis of Categorical, Ordinal and Numerical Data. *IACR Cryptology ePrint Archive* 2016 (2016), 1163. http://eprint.iacr.org/2016/1163

[33] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can Homomorphic Encryption Be Practical?. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2046660.2046682

[34] Femi Olumofin and Ian Goldberg. 2012. Revisiting the Computational Practicality of Private Information Retrieval. In *Financial Cryptography and Data Security*, George Danezis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–172.

[35] Rafail Ostrovsky and William E. Skeith. 2007. A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In *Public Key Cryptography – PKC 2007*, Tatsuaki Okamoto and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 393–411.

[36] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 359–374.

[37] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 85–100.

[38] R L Rivest, L Adleman, and M L Dertouzos. 1978. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press* (1978), 169–179.

[39] Sujoy Sinha Roy, Frederik Vercauteren, Jo Vliegen, and Ingrid Verbauwhede. 2017. Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search. *IEEE Trans. Comput.* (2017).

[40] Nigel P Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, codes and cryptography* (2014), 1–25.

[41] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 44–55.

[42] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 299–313. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wang-frank

[43] Barry Wilkinson and Michael Allen. 1999. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall.

[44] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, Washington, DC, USA, 162–167. https://doi.org/10.1109/SFCS.1986.25

[45] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, Washington, DC, USA, 162–167. https://doi.org/10.1109/SFCS.1986.25

[46] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. 2013. Secure Pattern Matching Using Somewhat Homomorphic Encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop (CCSW '13)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2517488.2517497

[47] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. 2015. New Packing Method in Somewhat Homomorphic Encryption and Its Applications. *Sec. and Commun. Netw.* 8, 13 (Sept. 2015), 2194–2213. https://doi.org/10.1002/sec.1164

A PROTOCOL SUMMARY

Our upload and search protocol is summarized in the following figure. We point out that we did not include in the figure our optimizations for reducing degree of SPiRiT from $O(p^2)$ to $O(p \log m)$ and reducing the effect on degree due to applying PIR from multiplicative factor $O(\log m)$ to additive 1; see section 3.3.

Parameters (Shared Input): The security parameter λ, the FHE scheme $\mathsf{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, the records space $\mathcal{M}$ and lookup values space $\mathcal{Q}$, the number of data records m (w.l.o.g m is a power of two), the set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of the smallest $k = 1+\log_2^2(m)/\log_2\log_2(m)$ primes larger than $\log_2 m$, and the polynomial realizing the matching criterion $is\textsc{Match}\colon \mathcal{M}\times\mathcal{Q} \to \{0,1\}$.

Inputs: The client's input is a data $array = (array(1), \ldots, array(m)) \in \mathcal{M}^m$, and a lookup value ℓ. The server has no input.

Outputs: The client's output is the binary representation $b^* \in \{0,1\}^{1+\log m}$ of the index $i^* = \min\{\, i \in [m] \mid is\textsc{Match}(array(i), \ell) = 1\}$. The server has no output.

Protocol:

(1) **Data Upload.** The client does the following:
- Generate keys $(pk_{p_1}, sk_{p_1}) \leftarrow \mathsf{Gen}(1^\lambda; p_1), \ldots, (pk_{p_k}, sk_{p_k}) \leftarrow \mathsf{Gen}(1^\lambda; p_k)$.
- Compute for all $i \in [m]$ and $j \in [k]$: $[\![array(i)]\!]_{p_j} \leftarrow \mathsf{Enc}_{pk_{p_j}}(array(i))$.
- Send to server $pk = (pk_{p_1}, \ldots, pk_{p_k})$ and $[\![array]\!] = ([\![array]\!]_{p_1}, \ldots, [\![array]\!]_{p_k})$, where for all $j \in [k]$, $[\![array]\!]_{p_j} = ([\![array(1)]\!]_{p_j}, \ldots, [\![array(m)]\!]_{p_j})$.

(2) **Client's search query.** The client computes for all $j \in [k]$: $[\![\ell]\!]_{p_j} \leftarrow \mathsf{Enc}_{sk_{p_j}}(\ell)$ and sends to the server

$$([\![\ell]\!]_{p_1}, \ldots, [\![\ell]\!]_{p_k})\,.$$

(3) **Server's computation.** The server executes the following steps for each $j \in [k]$:

(a) Compute for every $i \in [m]$

$$[\![indicator(i)]\!]_{p_j} \leftarrow \mathsf{Eval}_{pk_{p_j}}\left(is\textsc{Match}, [\![x_i]\!]_{p_j}, [\![\ell]\!]_{p_j}\right),$$

and denote

$$[\![indicator]\!]_{p_j} = ([\![indicator(1)]\!]_{p_j}, \ldots, [\![indicator(m)]\!]_{p_j})$$

(b) Compute

$$[\![b_{p_j}]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(\mathrm{SPiRiT}_{m,p_j}, [\![indicator]\!]_{p_j}).$$

where $\mathrm{SPiRiT}_{m,p_j} = S \circ P \circ i \circ R \circ i \circ T$ is the composition of the following components S, P, i, R, i, T (see section 3.2):
- **T**ree matrix $T \in \{0,1\}^{(2m-1)\times m}$ transforms a length m vector $x = (x_1, \ldots, x_m)$ to the length $2m-1$ array data structure $w = (w_1, \ldots, w_{2m-1})$ of the tree representation of x.
- **R**oots matrix $R \in \{0,1\}^{m\times(2m-1)}$ transforms the said tree representation $w = (w_1, \ldots, w_{2m-1})$ of x to the vector $v = Rw$ of prefix-sums $v_j = x_1 + x_2 + \ldots + x_j$ of x.
- **P**airwise difference matrix $P \in \{-1,0,1\}^{m\times m}$ transforms a a length m vector $u = (u_1, \ldots, u_m)$ to the vector $t = Pu$ of pairwise differences between consecutive entries; i.e., $t_j = u_j - u_{j-1}$ for $j \in \{2, \ldots, m\}$ and $t_1 = u_1$.
- **S**ketch matrix $S \in \{0,1\}^{(1+\log m)\times m}$ is a 1-sparse sketch matrix that transforms a binary vector $t = (0, \cdots, 0, 1, 0, \cdots, 0) \in \{0,1\}^m$ with a single non-zero entry in its jth coordinate to the binary representation $y = St \in \{0,1\}^{1+\log m}$ of $j \in [m]$ ($y = 0^{1+\log m}$ if $t = 0^m$). See construction of S in section 3.2.
- **i**sPositive operator $i()$ for a prime p, denoted $\textsc{isPositive}_p(\cdot)$ (or $i(\cdot)$ in short, when p is clear from the context), gets as input an integer vector $a = (a_1, \ldots, a_{m'})$, and returns a binary vector $\textsc{isPositive}_p(a) = \left(a_1^{p-1} \mod p, \ldots, a_{m'}^{p-1} \mod p\right)$.

(c) Compute

$$[\![indicator(b_{p_j})]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(PIR, [\![indicator]\!]_{p_j}, [\![b_{p_j}]\!]_{p_j}).$$

and

$$[\![array(b_{p_j})]\!]_{p_j} \leftarrow \mathsf{Eval}_{p_j}(PIR, [\![array]\!]_{p_j}, [\![b_{p_j}]\!]_{p_j})$$

where we apply here on *indicator* (similarly, on *array*) the known FHE based PIR polynomials (see section 1.1): $PIR(indicator, b_{p_j}) = \sum_{i=1}^m indicator(i) \cdot is\textsc{Equal}(i, b_{p_j})$, where $is\textsc{Equal}$ is the binary equality test specified in section 5.2; See faster alternatives in Section 3.3.

(d) Send to the client

$$\left([\![b_{p_j}]\!]_{p_j}, [\![indicator(b_{p_j})]\!]_{p_j}, [\![array(b_{p_j})]\!]_{p_j}\right)_{j\in[k]}$$

(4) **Client's decoding.** The client decrypts and outputs $(b, array(b))$ for:

$$b \leftarrow \min_{j\in[k]} \left\{b_{p_j} \text{ s.t. } indicator(b_{p_j}) = 1\right\}$$

Figure 9: Secure Search Protocol.