Secure Search via Multi-Ring Fully Homomorphic Encryption

Adi Akavia[1], Dan Feldman[2], and Hayim Shaul[2]

[1] Cybersecurity Research Center, Academic College of Tel-Aviv Jaffa
akavia@mta.ac.il
[2] Robotics & Big Data Lab, University of Haifa
dannyf.post@gmail.com, hayim.shaul@gmail.com

Abstract. *Secure search* is the problem of securely retrieving from a database table (or any unsorted array) the records matching specified attributes, as in SQL "SELECT...WHERE..." queries, but where the database and the query are encrypted. Secure search has been the leading example for practical applications of Fully Homomorphic Encryption (FHE) since Gentry's seminal work in 2009, attaining the desired properties of a single-round low-communication protocol with semantic security for database and query (even during search). Nevertheless, the wide belief was that the high computational overhead of current FHE candidates is too prohibitive in practice for secure search solutions (except for the restricted case of searching for a uniquely identified record as in SQL UNIQUE constrain and Private Information Retrieval). This is due to the high degree $\Omega(m)$ for m the number of database records of existing solutions, which is too slow even for moderate sizes m such as a few thousands.

We present the first algorithm for secure search that is realized by a polynomial of *logarithmic degree* $\log^{O(1)} m$. 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 to retrieve database records in a rate of *searching in millions of database records in less than an hour* on a single machine.

We achieve our result by: (1) Designing 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. (2) Suggesting a multi-ring evaluation of FHE – instead of a single ring as in prior works – and leveraging this to achieve an exponential reduction in the degree.

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) [39,16,17] is an encryption scheme with the special property of enabling computing on the encrypted data, while simultaneously protecting its secrecy; see a survey in [23]. 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 [46,19]). 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 [35,20,31,48,12,34,15]). 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 the hallmark example for useful FHE applications since Gentry's breakthrough result construction the first FHE candidate [16]. Use case examples are abundant: secure search for a document matching a retrieval query in a corpus of sensitive documents, such as private emails, classified military documents, or sensitive corporate documents; secure SQL SELECT WHERE query to a database, e.g., searching for a patient's record in a medical database based on desired attributes; secure search engine; etc. In all these use cases security means that both the *searched data* (documents, DB, etc.) and the *search query* are encrypted with semantically secure FHE, and that the *data access pattern* likewise reveal no information on the searched data or query.

The secure search problem at the core of all aforementioned use case examples can be captured as searching for an encrypted lookup value in an encrypted array (the array representing, for example, an encrypted table/column in a relational database, or a word-by-word encryption of a document for full text search); see Section 2 for details on the relation to the real-life use cases and further discussion. We focus on *single round* protocols, 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. [7]). 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 from which the client can compute (i, x_i) (decode).*

We call the client efficient *if its running time is polynomial in the output length $|i| = O(\log m)$ and $|x_i|$ and in the time to encrypt/decrypt a single ciphertext. 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 $\Omega(m)$.)*

As an example consider searching for an exact match to the lookup value ℓ in the data *array* whose entries are given in binary representation $x_1, \ldots, x_m \in \{0,1\}^t$ of length t bits. For this case we set isMATCH to be the equality test $is\textsc{Match}(x_i, \ell) = 1$ if-and-only-if $x_i = \ell$, which can be realized for example by a polynomial of degree and size $O(t^2)$ when using the equality test polynomial $is\textsc{Equal}_t$ in Section 3.5. So in this case we call the server efficient if the secure search polynomial $y = f(array, \ell)$ is of degree polynomial in $\log m$ and t and of size polynomial in m and t.

Generic isMATCH predicate. The predicate isMATCH() in the Definition 1 is a generic predicate that can be instantiated to any desired functionality (with complexity affected accordingly, see Theorem 1). Moreover, a concise specification of isMATCH() can typically be used, e.g., by the client providing the function's name. For example, most generally, isMATCH() 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 isMATCH() 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 $is\text{MATCH}(x_i, \ell) = 1$ 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 isMATCH() condition (see Theorem 1).

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

1.1 Prior Works

Prior secure search protocols suffer from one of the following shortcomings: (i) the protocol provides only a *restricted search functionality*, or (ii) the protocol is *inefficient* in the sense of having at least linear dependence on the database size m for either the client's running time or the degree of the polynomial computed by the server, or (iii) the *security is weakened* to leak vital search information; see details below.

Private Information Retrieval (PIR) provide a restricted search functionality, where the client's lookup value must be a *unique identifier* for at most a single record x_i in *array* (as in SQL UNIQUE constraint). The standard PIR settings are when this unique identifier is the index $i \in [m]$ (where here and throughout this work we use the notation $[m] = \{1, \ldots, m\}$); the techniques however extend to any unique identifier, i.e., any lookup value ℓ so that $is\text{MATCH}(x_i, \ell) = 1$ for at most a single record x_i (0 otherwise). Low degree polynomials realizing secure data retrieval for these unique identifier settings have been shown in prior works [16,8,14]. Specifically, the degree is essentially the degree of isMATCH which is in turn $O(\log m)$ when the lookup value is the unique index $i \in [m]$.

We note that in cases where the server holds *non-encrypted* data *array* and only the lookup value is secret, indexing techniques can reduce the search problem to the unique identifier settings by transforming the data into a table with a unique row for each lookup value ℓ (i.e., the number of rows is the size of the space $\mathcal{L}$ of possible lookup values), and where the entries of each row ℓ consist of the list of all records matching ℓ [9,40]. However, this transformation may incur a considerable time and memory overhead, because the produced table is of size $m \cdot |\mathcal{L}|$ for $|\mathcal{L}|$ the number of possible lookup values (rows), compared to size m of the original data *array*. This may be a considerable overhead since often $|\mathcal{L}| \gg m$; for example, for data an encrypted document of $m = 1000$ words and lookup values the $|\mathcal{L}| \geq 170,000$ words in the English dictionary. More importantly, this transformation assumes the data is given as plaintext (i.e., it is not encrypted), which is not the case in secure outsourcing settings as is the focus of this work. For these non-encrypted data settings, software and hardware implementations demonstrate a secure retrieval rate of processing millions of records in an hour [40] (when scaling up their results to a strong 64-cores machine as in our experiments; see Section 5).

Private set intersection (PSI). A recent work [10] following the preprint publication of our work [2] gave an FHE based Private Set Intersection (PSI). In the PSI problem, 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. The PSI, while related to search, does not provide the desired output. Specifically, thinking of X as the lookup value (with $|X| = 1$) and Y as a database column, the output only solves the *decision problem* of whether the lookup value appears in the database, but without returning entire records (or handles to such records, as in returning the corresponding row index i). Moreover, the modeling of X, Y as sets (rather than multi-sets) implies a UNIQUE constraint on Y, and so the lookup value (X) is restricted to be a unique identifier to the database column Y (as in PIR).

The natural (folklore) secure search solution for unrestricted settings (i.e., with no UNIQUE constraint) suffers from an inefficient server that evaluates a degree $\Omega(m)$ polynomial (for m the number of records). This is too slow with current FHE candidates and implementations, even for moderate size m such as a few thousands.

In secure pattern matching works [47,13,32,11,26,21,27] the client is inefficient.[3] In these works, the server computes an encrypted array

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

specifying for each $i \in [m]$ the (encrypted) zero/one indicator of whether record x_i is a match to the lookup value ℓ. This encrypted vector $[\![indicator]\!]$ is sent from the server to the client, who decrypts and scans $indicator \in \{0,1\}^m$ to find a indices of the records to be retrieved. To retrieve the records themselves, the parties can now engage in a PIR protocol, requiring a second round of interaction. In these works, while the server is efficient (computes polynomials of degree $\deg(is\textsc{Match}())$), the client is not: the client's running time is linear in the number of records $\Omega(m)$.

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 the pioneering work of Song, Wagner and Perrig, IEEE S&P 2000 [42], with famous examples as CryptDB [38] and subsequent works [37,30,29] giving practical search solutions albeit with information leakage; see a survey in [6].

The wide belief was that secure search on FHE encrypted data cannot achieve reasonable running times. This is because in contrast to the said low degree solutions for the case of searching for a uniquely identified record as in PIR, no low degree polynomials are known for realizing the (unrestricted) secure search problem. Consequently, to the best of our knowledge including [41] there is no prior art implementation for secure search on FHE encrypted data. Indeed, the folklore polynomial realizing secure search has degree $\Omega(m)$ for m the number of records, which is too slow with current FHE candidates and implementations even for moderate size m such as a few thousands. The common belief was that the computational overhead of FHE is too prohibitive for secure search to be used in practice; see for example in [8,38,43,45,33,28].

1.2 Our Contribution

In this work we provide evidence that, counter to the common belief, secure search on FHE encrypted data may be of relevance to practice. Our contributions in this work are as follow.

The first efficient protocol for secure search (see Protocol 2 and Figure 2) that is applicable to large datasets with unrestricted search functionality. Specifically, our protocol provides:

- *Efficient client:* The client's running time is proportional to the time to compute $\log^{O(1)} m$ encryption/decryption operations plus the time to read the retrieved record (i, x_i).
- *Efficient server:* The server evaluates a polynomial of degree $\log^{O(1)}(m) \cdot \deg(is\textsc{Match})$ and size $m \cdot \text{size}(is\textsc{Match})$ (where we denote by $\deg(f)$ and $\text{size}(f)$ the degree and size of the polynomial f).
- *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.

[3] Some of these works offer alternative usage scenarios such as: obtaining a *YES/NO* answer on whether the lookup value appears in the data, or returning a vector of *scores* on how good a match to the lookup value each data entry is (as in Hamming or Edit distance). However, none of the suggested alternatives can return, as desired, an efficient and concise representation of a unique handle to the matching record (unless requiring lookup values to be unique identifiers, as discussed above for the PIR and PSI settings).

– *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 during searching.

Furthermore, the protocol is single round protocol and with low-communication complexity, as is the focus of this work. This is summarized in Theorem 1 below, with proof given in Section 4.2.

Theorem 1 (Secure Search). *Protocol 2 is an efficient secure search protocol (see Definition 1).*

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 to be a unique identifier to the data as in SQL UNIQUE constraint (see PIR and PSI solutions); or compromises security (see searchable encryption solutions). A summary of the comparison to prior works appears in Tables 1-2.

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 single 64-cores machine 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.

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. So we can answer, for example, search queries of a database of billions of entries in less than an hour, using a cluster of roughly 1000 machines; See Section 5.

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 4.3 and Figure 1.

Open Source Library for Secure-Search with FHE based on HELib [22] 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.

	Client's time	Server's degree	Protocol
Secure Pattern Matching	$\Omega(m)$	$(\log m)^{O(1)}$	inefficient
Folklore Secure Search	$O(\log m)$	$\Omega(m)$	inefficient
This Work: Secure Search	$(\log m)^{O(1)}$	$(\log m)^{O(1)}$	efficient

Table 1. Comparison of client, server, and protocol complexity (see Definition 1) in works supporting unrestricted search functionality, record retrieval, and full security.

1.3 Our Novel Techniques: First Positive Sketch (SPiRiT) and Multi-Ring FHE Evaluation

We propose a novel low degree secure search polynomial for the server to evaluate on the data $array = (array(1), \ldots, array(m))$ and lookup value ℓ to obtain, essentially, the binary representation $b^* \in \{0,1\}^{1+\lceil \log m \rceil}$ of the index $i^* \in [m]$ of the first match for ℓ in *array* (or $b^* = (0 \ldots 0)$ if no match exists):

$$i^* = \min \{\, i \in [m] \mid is\textsc{Match}(array(i), \ell) = 1 \} \qquad (i^* = 0 \text{ if no match exists}).$$

Protocols and Papers	Efficient Client	Efficient Server	Supports unrestricted search functionality	Retrieves record	Full security	Records per hour per machine
Searchable Encryption [6,42]	✓	N/A	✓	✓	×	Gb
PIR [16,8,14,9,40]	✓	✓	×	✓	✓	Mb
PSI [10]	✓	✓	×	×	✓	Mb
Secure Pattern Matching [47,13,32,11,21,27]	×	✓	✓	∼ ✓	✓	Kb
Folklore Secure Search	✓	×	✓	✓	✓	Kb
This work: Secure Search	✓	✓	✓	✓	✓	Mb

Table 2. *Comparison to single-round 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 specifies number of processed records per hours per machine in reported experiments: thousands (Kb), millions (Mb), billions (Gb).

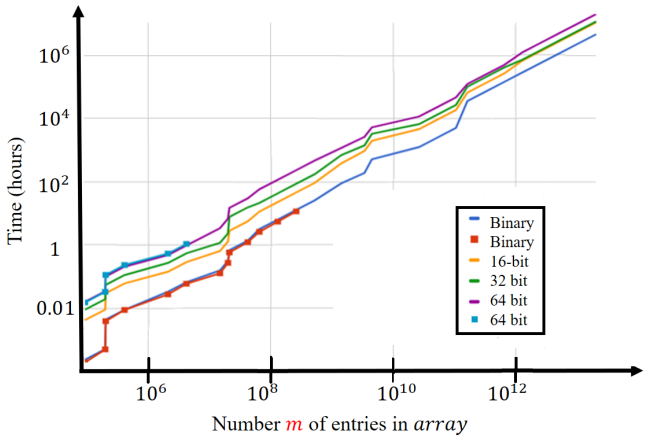

Fig. 1. Server's running time in Protocol 2 (y-axis) as a function of the number of records m in database $array$ (x-axis), where each entry $array(i)$ is represented by 1 bit, 16 bits, 32 bits, 64 bits (curves). The graph depicts both measured running times (squares), and estimated running times (curves). 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 4.3.

The server evaluates this polynomial over encrypted inputs $[\![array]\!], [\![\ell]\!]$ and obtains encrypted output $[\![b^*]\!]$, using homomorphic operations.

More precisely, the server computes and sends to the client *a short list of candidates* for the binary representation $b^* \in \{0,1\}^{1+\lceil \log m \rceil}$ of i^*. From this list of candidates we show that the client can efficiently decode the correct value i^*, essentially by choosing the smallest candidate; See Figure 2.

We note that to simplify the presentation we focus here on returning the index i^* and not the value $array(i^*)$. Nonetheless, using standard PIR techniques and with no further interaction, the server can easily return (index,value) pairs, in which case the client can efficiently decode to obtain the desired pair $(i^*, array(i^*))$.

At a high level the polynomial we propose is composed of two main parts. In the first part, on input $array, \ell$ the output is a binary vector $indicator \in \{0,1\}^m$ that indicates for each entry $i \in [m]$ whether the record

$array(i)$ matches the lookup value ℓ. Namely,

$$indicator = (isMatch(array(1), \ell), \ldots, isMatch(array(m), \ell))$$

(we point out that here we can use any generic pattern matching polynomial $isMatch()$ given as part of the problem specification). In the second part, we'd like to output (the binary representation $b^* \in \{0,1\}^{1+\lceil \log m \rceil}$ of) the *first positive index* $i^* \in [m] \cup \{0\}$ of $indicator \in \{0,1\}^m$ ($i^* = 0$ if $indicator$ is all zeros); namely, the index so that

$$indicator(1) = \ldots = indicator(i^* - 1) = 0 \text{ and } indicator(i^*) = 1.$$

Observe that i^* is precisely the index of the first match in $array$ for ℓ, namely, the first index so that

$$isMatch(array(i^*), \ell) = 1.$$

The challenge is that the natural polynomial for computing this first positive index i^* is of high degree $\Omega(m)$, resulting in an inefficient protocol. Our main technical contribution is proposing a novel way to compute this first positive index i^* via low degree polynomials of degree $\log^{O(1)} m$, resulting in an efficient protocol. For this purpose we introduce two novel techniques: SPiRiT *sketch for first positive* and *multi-ring FHE evaluation*, as discussed next.

The sketch for first positive $\text{SPiRiT}_{m,p}$, parameterized by the length m of the input vector and the modulus p for the arithmetic operations, is a degree $(p-1)^2$ polynomial whose desired output in its evaluation $\text{SPiRiT}_{m,p}(indicator)$ on $indicator \in \{0,1\}^m$ is the first positive index of $indicator$. That is, the binary representation $b \in \{0,1\}^{1+\lceil \log m \rceil}$ of the index $i \in [m] \cup \{0\}$ so that $i \in [m]$ is the index of the first positive entry of $indicator$, and $i = 0$ if no positive entry exists.

The problem is that to guarantee that the output is the first positive index, as desired, we require a large modulus $p = \Omega(m)$ which result in a polynomial $\text{SPiRiT}_{m,p}$ of high degree $\Omega(m^2)$.

To resolve this problem we suggest to replace the evaluation of $\text{SPiRiT}_{m,p}$ on a single large ring $p = \Omega(m)$ by few $k = o(\log^2 m)$ evaluations of such polynomials but on small rings moduli $p_1, \ldots, p_k = O(\log^2 m)$; namely, we suggest evaluating in parallel k polynomials each of low degree $O(\log^4 m)$. From an FHE point of view, we didn't solve the original problem of returning the desired value b^* to the client. Instead, we return the k values above. However, the additional time on the client side that is required to extract the desired value i^* from our k outputs is very fast with only a small overhead over receiving b^* by a factor of $o(\log^2 m)$.

In short, we redefine a hard problem (that requires a single output from a large-ring polynomial) to an easier problem that uses multiple outputs (from few small rings polynomial), in the cost of additional but minor amount of computation on the client side.

Multi ring FHE evaluation. Our techniques of evaluating FHE over multiple rings is novel in the context of FHE, whereas prior works model the FHE evaluation as computing an arithmetic circuit over a single ring modulus p (namely, with gates computing addition and multiplication modulo a single p). In those single ring arithmetic circuits all known secure search solutions have multiplicative depth $\Omega(\log m)$ (where the multiplicative depth is essentially equivalent to the logarithm of the degree of the polynomial evaluated by the circuit), and are therefore not considered practical.

In this work we propose a multi ring FHE evaluation, that is, computing an arithmetic circuit with gates for additions/multiplication over several ring moduli $p_1, \ldots, p_k$. Our result shows that there exists a multi ring arithmetic circuit for secure search of multiplicative depth $O(\log \log m)$. This presents an exponential improvement over the prior art; See Figure 3. We hope that this novel technique will be used in future works to attack other computations in the context of FHE that currently have impractical solutions.

Overview of SPiRiT. Our SPiRiT sketch for first positive is a novel low degree polynomial we introduce for computing the first positive index. This polynomial $\text{SPiRiT}_{m,p}()$ is parameterized by the length m of the input vector and the modulus p for the arithmetic operations, and is defined to be the composition

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

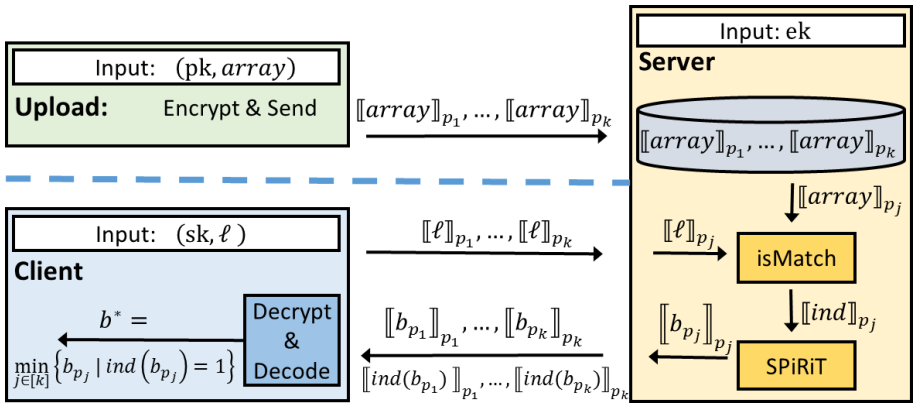

Fig. 2. Depicting our Secure Search Protocol 2 and the Data Upload Protocol 1. In the data upload protocol the client, whose input is the public key and the data *array*, encrypts the data and sends to the server. In the secure search protocol, the server's input is the evaluation key and the encrypted data that was previously uploaded; The client's input is the secret key and a lookup value; A common input (in both Protocols 1-2) is the set of prime numbers $p_1, \ldots, p_k$. The client sends to the server the lookup value ℓ encrypted in k ciphertexts with plaintext moduli $p_1, \ldots, p_k$; where we use the notation $[\![x]\!]_p$ to denote a ciphertext encrypting message x to enable homomorphic addition and multiplication modulo p. The server evaluates, for each modulus p_j, the pattern matching polynomial isMATCH on the encrypted lookup value $[\![\ell]\!]_{p_j}$ and data $[\![array]\!]_{p_j}$ to obtain an encrypted indicator vector $[\![ind]\!]_{p_j}$. The server then evaluates on this encrypted indicator vector our SPiRiT$_{m,p_j}$ sketch for first positive to obtain a candidate $[\![b_{p_j}]\!]_{p_j}$ for its first positive index. The server sends to the client these k candidates $[\![b_j]\!]_{p_j}$ together with the corresponding k entries in ind. The client decrypts and outputs the smallest candidate b_j s.t. $ind(b_j) = 1$.

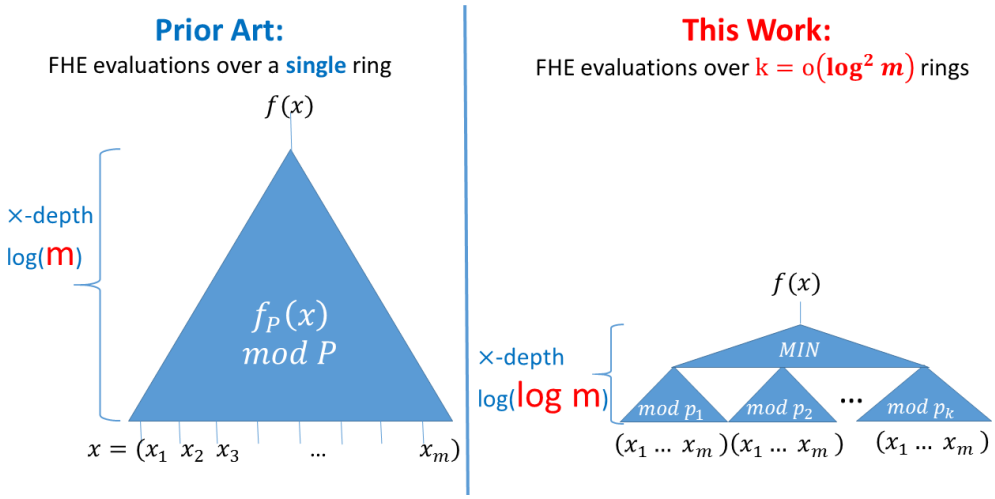

Fig. 3. Single/Multi ring arithmetic circuit for secure search: the multiplicative depth of known single ring circuits is exponentially higher than in our proposed multi ring circuit.

of a **S**ketch, **P**airwise, **R**oots, and **T**ree matrices, together with an operator $i(x) = \text{ISPOSITIVE}_p(x)$ for turning integer vectors x to binary vectors accepting value 1 on entries where x is nonzero, and value 0 on entries where x is zero.

We elaborate on the components of SPiRiT$_{m,p}$ and their roles. The role of $i \circ R \circ i \circ T(indicator)$ —which is not always satisfied, see discussion below— is to return the step function $u \in \{0,1\}^m$ accepting value 0 on entries $1, ..., i^* - 1$ and value 1 on entries $i^*, \ldots, m$. For this purpose the tree matrix T computes the labels of a binary tree with m leaves labeled by the entries of $indicator$, and where each node is labeled by the sum of

the labels of its children. These labels are then reduced to binary values using isPositive_p operator. Next, for each $i \in [m]$, the roots matrix R partitions the tree leaves $1, \ldots, i$ according to their deepest common ancestor whose leaves are contained in the leaves $1, \ldots, i$, and sums up the labels of these ancestors; see Figure 4. The resulting values are again reduced to binary values using isPositive_p operator. The pairwise difference matrix P then computes the derivative of u, which in the case where u is the said step function results in a binary vector with a single non-zero entry at index i^*. The sketch matrix $S \in \{0,1\}^{(1+\lceil \log m \rceil) \times m}$ is a standard sketch matrix for 1-sparse vectors, i.e., a matrix that given a binary vector with at most a single non-zero entry returns the binary representation of the index of this entry (or zero if none exists). Finally, to compute the isPositive_p operator we rely on Fermat's Little Theorem: $\text{isPositive}_p(x_1, \ldots, x_{m'}) = (x_1^{p-1} \bmod p, \ldots, x_{m'}^{p-1} \bmod p)$. The degree of $\text{SPiRiT}_{m,p}$ is $(p-1)^2$ (which can be lowered to $p \log m$ with further optimizations; see Lemma 2).

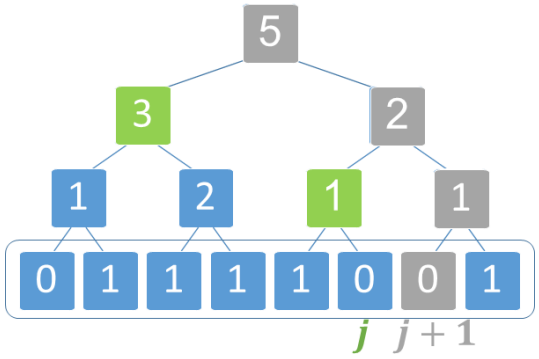

Fig. 4. The tree representation for a length $m = 8$ binary vector $indicator = (0,1,1,1,1,0,0,1)$ is the full binary tree with m leaves labeled by entries of $indicator$, and with internal nodes labeled by the sums of their children's labels. The array data structure for this tree is the length $2m-1$ vector $w = T \cdot indicator = (5,3,2,1,2,1,1,0,1,1,1,1,0,0,1)$. The prefix sum of leaves' labels up to the $j = 6th$ leaf from the left is $v(6) = 0+1+1+1+1+0 = 4$. More generally, the vector of prefix sums of $indicator$ is $v = (0,1,2,3,4,4,4,5)$. The root matrix R has the property that $RT \cdot indicator = Rw = v$. To construct sparse R we observe that every entry of v can be computed using only $O(\log m)$ labels; this is by summing the roots of subtrees forming a partition of the leaves in the considered prefix. For example, to compute the $j = 6th$ entry $v(6)$ we sum two labels as follows: First identify all the ancestors of the $j+1 = 7th$ leaf: labeled by $0,1,2,5$ in the figure (colored in grey). Among these ancestors, select those who are right children: labeled by 1 and 2 in the figure. Finally, sum the labels of the left siblings of these selected ancestors: labeled by 1 and 3 in the figure (colored in green) to get the desired sum. Indeed, $v(6) = 1+3 = 4$. .

The problem we face is that on the one side, correctness requires a large modulus $p = \Omega(m)$; otherwise, we get wrong outputs due to overflow. For example, when using a small $p \ll m$, applying the isPositive operator on the tree labels (i.e., computing $i \circ T(indicator)$) reduces to zero all tree labels that are a multiple of p, instead of only labels that are zero over the integers; consequently u might not be a step function, and the output will not be the desired index i^*. On the other side, efficiency requires a small modulus $p = \log^{O(1)} m$, because the degree of $\text{SPiRiT}_{m,p}$ is polynomial in p.

To resolve this problem we first prove in our analysis a key property of $\text{SPiRiT}_{m,p}$ as follows.

> **Key Property:** *If the labels of the ancestors of the first positive leaf i^* in the tree $T(indicator)$ are not multiples of p, then $\text{SPiRiT}_{m,p}(indicator)$ returns the binary representation of i^**; see Lemma 3, Section 4.1.

Next, we observe that there are not too many primes p that are divisors of these labels, specifically, at most $\log^2 m$ such primes. So by Pigeonhole principle, for any $k = 1 + \log^2 m$ primes p, at least one of them would

satisfy the above condition on the ancestors of i^*. By the above key property, for this p, $\mathrm{SPiRiT}_{m,p}(\mathit{indicator})$ returns the correct output. Therefore, by computing $\mathrm{SPiRiT}_{m,p}(\mathit{indicator})$ (in parallel) for k primes p, we obtain a list of candidates $b_1, \ldots, b_k$ for the binary representation b^* of the first positive index i^*, with the guarantee that for $i_1, \ldots, i_k \in [m] \cup \{0\}$ the corresponding indices:

$$i^* = \min_{j \in [k]} \{i_j \text{ s.t. } \mathit{indicator}(i_j) = 1\}.$$

The decoding algorithm on the client side is therefore to choose from the list of candidates received from the server the smallest candidate that is verified to be a match (i.e., $\mathit{indicator}(i_j) = 1$). We remark that for the purpose of this verification the server in our protocol sends to the client the values $\mathit{indicator}(i_j)$ for all $j \in [k]$ (computed using standard PIR techniques), on top of sending the candidates b_{p_j}.

More tightly, it in fact suffices to set $k = 1 + \log^2 m / \log\log m$ by choosing only primes p larger than $\log m$. Furthermore, by the Prime Numbers Theorem, we can find such $k = O(\log^2 m / \log\log m)$ primes in a sufficiently short interval to ensure that all the primes are of magnitude $O(\log^2 m)$. The degree of the $\mathrm{SPiRiT}_{m,p}$ polynomials for these primes p is therefore $O(p \log m) = O(\log^3 m)$.

In summary, we propose a new polynomial $\mathrm{SPiRiT} = (\mathrm{SPiRiT}_{m,p_1}, \ldots, \mathrm{SPiRiT}_{m,p_k})$ for computing a short list of $k = o(\log^2 m)$ candidates for the first positive entry in a given non-negative length m vector. Our analysis proves that the smallest of the verified candidates is the correct solution, that is, it is the binary representation $b^* \in \{0,1\}^{1+\lceil \log m \rceil}$ of the first strictly positive entry $i^* \in [m]$ of the input vector (and it is $b^* = 0$ if no such entry exists). We call this polynomial a "sketch for first positive". This sketch may be of independent interest beyond the context of secure computation, and it is extraordinary in that unlike other Group Testing sketches (e.g. [25]) it can be applied on non-sparse input vectors.

Technical difficulty: how to compute minimum when each ciphertext has a differing plaintext modulus p? We now elaborate on the encryption we use and explain why we relay computing the minimum verified candidate to the client, instead of requesting the server to do it.

Observe that computing the minimum of $k = o(\log^2 m)$ values requires only a low degree polynomial, so typically this could be accomplished efficiently by the server. However, the guarantee of FHE schemes is to enable given ciphertexts $[\![x]\!]$ to execute computations on the underlying plaintext x with respect to *a single ring*. For example, the plaintext space could be integers and the ring operations are addition and multiplication modulo a prime p (aka, the *plaintext modulus*). We denote by $[\![x]\!]_p$ a ciphertext encrypting the message x so that we can apply on it homomorphic addition and multiplication modulo p.

For some FHE candidates (e.g. [7]), given a ciphertext $[\![x]\!]_p$, it is possible to transform it to a ciphertext for a plaintext modulus p' which is a multiple of p (or other related primes p'). However we do not know how to achieve such a transformation for arbitrary p' (unless using bootstrapping, which is considered impractical).

To evaluate $\mathrm{SPiRiT}_{m,p}$ on encrypted data for k distinct primes p, we therefore use FHE scheme that allows encrypting ciphertext with respect to these moduli p (e.g. [7]), and require that each time the client encrypts a message x she produces a tuple of k ciphertexts $[\![x]\!] = ([\![x]\!]_{p_1}, \ldots, [\![x]\!]_{p_k})$ corresponding to these k distinct primes (specifically, we use the smallest k primes that are larger than $\log m$). The server then evaluates $\mathrm{SPiRiT}_{m,p_1}, \ldots, \mathrm{SPiRiT}_{m,p_k}$ (in parallel), where each SPiRiT_{m,p_j} is evaluated on the corresponding ciphertexts $[\![\cdot]\!]_{p_j}$. The output is the resulting list of k candidates $[\![b_1]\!]_{p_1}, \ldots, [\![b_k]\!]_{p_k}$.

Getting back to discussing the minimum function, computing the minimum of the verified of these candidates $[\![b_1]\!]_{p_1}, \ldots, [\![b_k]\!]_{p_k}$ seems to require switching the plaintext modulus to a common prime p_0, which we do not know how to accomplish efficiently. This is why we relay computing the minimum to the client.

We remark that the fact that we encrypt each message using k ciphertexts incurs a factor $k = o(\log^2 m)$ overhead on the time and space complexity in comparison to when using a hoped FHE scheme where we could efficiently switch between the plaintext moduli.

1.4 Extensions and Followups

Extensions to dynamic data with insert/updade/delete on top of search are straightforward. For example, insertion simply requires the server to append another ciphertext to the end of the encrypted data *array*

(note that we assume here that the size of the data is known to the server). Likewise, update are a simple extension of PIR techniques, when given the unique index i for the record to be modified. Delete can be implemented by updating a record to a reserved "Deleted" symbol, or by updating the value or record i to that of the last record and reducing the number of records by 1 (for cases when the dynamic size of the data is either maintain by the client, or is not a secret and can be maintained by the server).

Extensions to multiple clients are immediate when all clients posses the secret key. For example, a group of users and a company can share the secret key and search/upload records independently and simultaneously, using existing db transactions locks mechanisms on un-trusted cloud. Moreover, uploading data does not even require sharing the secret key for the FHE scheme. Instead, it suffices for the data owner (say, the company) to publish the public key, so that all participants can upload encrypted data.

Followup work on returning all matching database entries. In a recent work [3] following the preprint publication of this work, it is shown how to return *all* database entries that match the lookup value, where the search is realized by a polynomial of degree polynomial in $\log m$. Their complexity naturally grows with the number of matching records, but only in terms of the client's running time and the size of the polynomial evaluated by the server, whereas the degree of the polynomial computed by the server remains low even when the lookup value has many returned matches.

We point out that [3] is incomparable to this work: It may be undesirable to return all matching records if there are too many of them (as this incurs a communication and client complexity burden). Instead, our work allows the client to retrieve the matching records one-by-one (where each search query returns a single match as in SQL FETCH_FIRST, and repeated queries can retrieve the following matching records as in SQL FETCH_NEXT). Moreover, combining techniques of [3] with ours we can retrieve in each query the next s matches, for a parameter s (compared with $s = 1$ in this work, and s being the total number of matches in [3]); details to appear in the full version of this work.

Followup work on reducing the $k = o(\log^2 m)$ client's overhead due to the multi-ring encryptions/decyptions. Recall that the fact that we encrypt each message using k ciphertexts incurs a factor $k = o(\log^2 m)$ overhead on the time and space complexity. A followup work shows how to avoid such overhead [5], albeit with introducing a negligible probability of error.

2 Discussion of the Secure Search Problem Statement

In this section we discuss the secure search formulation of Definition 1 to demonstrate its wide applicability to real-life use case examples and remark on its properties.

2.1 Use case examples

The formulation in Definition 1, despite its simplicity, captures a wide variety of real-life problems; examples follow.

For securely searching a documents' corpus with a document-term matrix *representation* , the array entries correspond to the rows of the document-term matrix (where recall that this matrix has columns corresponding to terms, and each row indicates the terms appearing in the corresponding document), and the lookup value specifies the subset of matching documents, e.g., by specifying a list of terms so that the matching documents are those including the conjunction of these terms.

For securely searching a documents' corpus with a bag-of-words *representation* , each array entry corresponds to a bag-of-words representation of the corresponding document, and the lookup value specifies the subset of matching documents, e.g., by a regular expression for wildcard matching.

For securely searching a database table with rows corresponding to records and columns corresponding to their various attributes, each array entry corresponds to a database row, and the lookup value specifies the matching attributes, say an exact match query for a particular attribute (database column). We remark that our experimental results address this latter use case example.

2.2 Remarks on the Secure Search Definition

We continue with remarks on the various aspects of the secure search definition.

What pattern matching subroutines can be used? Importantly, to allow such a wide applicability for the above problem formulation, our solution works with any generic specification of what values x_i constitute a match to ℓ. The specification is given by a pattern matching algorithm $is\textsc{Match}(\cdot,\cdot)$ that given pairs x_i, ℓ returns 1 if they are a match and 0 otherwise (e.g. equality test, wildcard matching, conjunction/disjunction queries, Hamming/Edit/Euclidean distance measures etc.), and our algorithm is generic in the sense we can plug-in any such specification. The overall complexity depends of course on the complexity of this $is\textsc{Match}()$ subroutine; our experimental results are for the case of exact equality $is\textsc{Match}(x_i, \ell) = 1$ if-and-only-if x_i is identical ℓ. For concreteness we focus throughout this work on the exact match case; the extension to generic $is\textsc{Match}()$ matching algorithm is immediate.

Why return one match and not all? We focus here on applications where there may be an unlimited abundance of matched values, so that returning all matches can pose an undesirable burden on the client, both in term of the communication complexity for receiving all matches, and in terms of the computational complexity for reading all of them. We require therefore to returning one matching value.

Which match to return? For concreteness we focus on the case of returning the index i for the *first* match (as in SQL FETCH_FIRST). That is, we return (the binary representation of) the index

$$i^* = \min \{ i \in [m] \mid is\textsc{Match}(array(i), \ell) = 1 \}$$

(where here, and in the rest of the paper, we assume that the minimum of an empty set is 0). We stress nonetheless that our framework easily extends to allow the client to retrieve the matching records one-by-one (as in SQL FETCH_NEXT); details to appear in the full version of this work.

Returning index i or record (i, x_i)? For simplicity of the presentation we focus primarily on returning the index i, because this is the challenging heart of the retrieval problem. This can trivially be extended to returning the pair (i, x_i) for x_i the matched value (e.g. document, database row, etc.). This is because once the unique index i is found, retrieving the entire value is easily solved by applying known PIR on FHE encrypted data and index [16,8,14]. We stress that applying PIR here is done on the server side, using the vale $[\![i]\!]$ that the server has computed, and requires no interaction with the client.

How to employ secure search in secure outsourcing scenarios? The client uploads encrypted data *array* to the server at an offline phase; only the client knows the secret decryption key. To initiate a secure search, the client submits an encrypted lookup value ℓ, the server returns the encrypted result y, so that there is an efficient decoding algorithm to obtain from y the matching index i.

What is the complexity goal? Our primary complexity goal is to minimize the client's latency, that is, the wait time between sending a search query and obtaining the search result. This latency accounts for both the server's time for evaluating the search polynomial and the client's time for encryption, decryption, and decoding of the received evaluation outcome.

Our theoretical analysis of the server's computational complexity specifies the degree and size of the computed polynomial (or, more precisely, the maximum degree over all polynomials computed in parallel). This degree d corresponds to a $\log d$ upper bound on the aggregate homomorphic multiplication steps. That

is, the multiplicative depth of the corresponding arithmetic circuit is $\log d$. The size s is a (typically non-tight) upper bound on the width w of this arithmetic circuit. We ignore the homomorphic addition operations, because they are much much cheaper than homomorphic multiplications, both with respect to the error they introduce in the context of homomorphic evaluation, and also with respect to running time.

The client's latency naturally depends on the number of parallel processors available to the server. In case the server has w parallel processors, the client's latency is essentially the multiplicative depth $\log d$ (because the server can employ the parallel processors to compute each layer of the circuit in unit time). In case the server has a single processor, the client's latency is upper bounded by $w \log d$ (due to requiring $O(w)$ time for each of the $\log d$ layers). In general, for a server with t processors, the client's latency is essentially $\frac{w}{t} \log d$.

Our experimental results on the server's complexity measure the actual running time, which accounts for all executed operations: homomorphic additions on top of homomorphic multiplications. Moreover, most of our experimental results are done on a single computer, so the server's time in our experiments grows with the total number of operations, not only the $\log d$ time for aggregate multiplications.

Our analysis of the client's running shows that the decoding time is $o(\log^2 m)$, and the overall client's running time is proportional to computing $o(\log^2 m)$ encryption and decryption operations. Note that our theoretical analysis needs not specify the encryption/decryption times, because these are properties of the underlying encryption scheme (where any suitable FHE can be used), and not of the search algorithm we propose.

3 Our Secure Search Protocol

In this section we present our secure search protocol. We first specify the requirements in our black-box use of fully (or, leveled) homomorphic encryption in Section 3.1; then describe the data upload protocol in Section 3.2; then present our secure search protocol in Section 3.3 with details of the components of SPiRiT specified in Section 3.4. In Section 3.5 we specify the particular pattern matching polynomial that we implemented for our experimental results. Finally, in Section 3.6 we discuss a randomized variant of our secure search protocol introducing a probability of error for gaining a reduced overall number of multiplications.

3.1 Black Box Usage of Semantically Secure Fully Homomorphic Encryption (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), and could use almost any FHE schemes (both public key schemes and symmetric schemes). 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, [7]. 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 $\mathrm{E} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$ (defined here for the symmetric key settings for simplicity):

- Gen is a randomized algorithm that takes a security parameter λ as input and a prime p, and outputs a secret key $sk_p = (p, sk)$ and a evaluation key $ek_p = (p, ek)$ for plaintext modulus p, denoted:

$$(sk_p = (p, sk), ek_p = (p, ek)) \leftarrow \mathrm{Gen}(1^\lambda; p).$$

- Enc is a randomized algorithm that takes sk_p and a plaintext message msg, and outputs a ciphertext $[\![msg]\!]_p$ for plaintext modulus p, denoted:

$$[\![msg]\!]_p \leftarrow \mathrm{Enc}_{sk_p}(msg).$$

– Dec is an algorithm that takes sk_p and a ciphertext $[\![msg]\!]_p$ as input, and outputs a plaintext msg', denoted:

$$msg' \leftarrow \mathrm{Dec}_{sk_p}([\![msg]\!]_p).$$

Correctness is the standard requirement that $msg' = msg$.

– Eval is a (possibly randomized) algorithm takes ek_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:

$$c \leftarrow \mathrm{Eval}_{ek_p}(f, [\![m_1]\!]_p, \ldots, [\![m_t]\!]_p).$$

Correctness is the requirement that decryption would return the message resulting from evaluating (modulo p) the polynomial $f()$ on inputs $m_1, \ldots, m_t$

$$\mathrm{Dec}_{sk_p}(\mathrm{Eval}_{ek_p}(f, [\![m_1]\!]_p, \ldots, [\![m_t]\!]_p)) = f(m_1, \ldots, m_t) \mod p.$$

Semantic security implies that the resulting ciphertext c is computationally indistinguishable from a fresh ciphertext $[\![f(m_1, \ldots, m_t)]\!]_p$.

3.2 Uploading Encrypted Data

Secure outsourcing of computation is the settings we address. For this purpose the client uploads to the server its encrypted data. The upload protocol is simple: the clients encrypt the data and sends to the server. The encryption is with a semantically secure FHE scheme, where for each message msg we produce a tuple of $k = 1 + \log^2 m$ ciphertexts $([\![msg]\!]_{p_1}, \ldots, [\![msg]\!]_{p_k})$ for plaintext modulus $p_1, \ldots, p_k$, respectively, chosen to be the first k primes larger than $\log m$. For simplicity of the presentation we assume the entire data $array$ is uploaded in a single round; this can easily be modified to incremental upload of the data. See Protocol 1.

Algorithm 1: Data Upload Protocol

Shared Input: An FHE scheme $\mathrm{E} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$,
A number m of data records in $array$, where w.l.o.g. we assume m is a power of two,
A set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of the smallest $k = 1 + \log^2 m$ primes that are larger than $\log m$.

Inputs: The client's input is a security parameter λ and $array = (array(1), \ldots, array(m))$
The server has no input.

Outputs: The client's output is a secret key $sk = (sk_{p_1}, \ldots, sk_{p_k})$ (for the FHE and security λ).
The server's output is the corresponding evaluation key $ek = (ek_{p_1}, \ldots, ek_{p_k})$, and
the encrypted data $[\![array]\!] = ([\![array]\!]_{p_1}, \ldots, [\![array]\!]_{p_k})$
(where $array$ is encrypted entry-by-entry: $[\![array]\!]_p = ([\![array(1)]\!]_p, \ldots, [\![array(m)]\!]_p)$).

1. The client does the following:
   – Generate keys $(sk_{p_1}, ek_{p_1}) \leftarrow \mathrm{Gen}(1^\lambda; p_1), \ldots, (sk_{p_k}, ek_{p_k}) \leftarrow \mathrm{Gen}(1^\lambda; p_k)$. Denote
   $$ek = (ek_{p_1}, \ldots, ek_{p_k}).$$
   – Compute for all $i \in [m]$ and $j \in [k]$:
   $$[\![array(i)]\!]_{p_j} \leftarrow \mathrm{Enc}_{sk_{p_j}}(array(i)).$$
   – Send to server
   $$ek \text{ and } [\![array]\!] = ([\![array]\!]_{p_1}, \ldots, [\![array]\!]_{p_k}),$$
   where $array$ is encrypted entry by entry: $[\![array]\!]_p = ([\![array(1)]\!]_p, \ldots, [\![array(m)]\!]_p)$.

3.3 The Secure Search Protocol

To securely search for a lookup value ℓ in the encrypted data $[\![array]\!]$ outsourced to the server, the client encrypts ℓ and sends the corresponding cipehrtexts $[\![\ell]\!]$ to the server. The server evaluates our search polynomial which is the composition of the two parts (see the overview in 1.2): first computing the (encrypted) binary vector *indicator* with 1 in all entries of *array* that match ℓ using a generic $is\textsc{Match}()$ pattern matching protocol; and next returning a short list of candidates for the the index i^* of the first positive entry in *indicator* using our SPiRiT sketch for first positive (with the corresponding record $array(i^*)$, if so desired; see Section 2). The client then decodes and chooses the smallest of the verified candidates as the output. Encryption of each value *msg* is by a tuple of $k = 1 + \log^2 m$ ciphertexts, one ciphertext for plaintexts modulus $p_1, \ldots, p_k$ (where these moduli are chosen to be the first k primes larger than $\log m$). See Protocol 2 and Figure 2 for the protocol, with details on the components of SPiRiT sketch first positive in Section 3.4.

Algorithm 2: Secure Search Protocol

Shared Input: An FHE scheme $\mathrm{E} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec}, \mathrm{Eval})$,
A power of two m denoting the number of data records in *array*,
A set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of the smallest $k = 1 + \log^2 m$ primes that are larger than $\log m$.
A pattern matching polynomial $is\textsc{Match}(\cdot, \cdot)$.

Inputs: Client's input is the secret key $sk = (sk_{p_1}, \ldots, sk_{p_k})$ and a lookup value ℓ.
The server's input is the corresponding evaluation key $ek = (ek_{p_1}, \ldots, ek_{p_k})$, and the encrypted data $[\![array]\!] = ([\![array]\!]_{p_1}, \ldots, [\![array]\!]_{p_k})$.

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

1. The client compute for all $j \in [k]$:
$$[\![\ell]\!]_{p_j} \leftarrow \mathrm{Enc}_{sk_{p_j}}(\ell).$$
and sends to the server
$$([\![\ell]\!]_{p_1}, \ldots, [\![\ell]\!]_{p_k}),$$
2. The server does the following for each $j \in [k]$:
   (a) Compute
   $$[\![indicator]\!]_{p_j} \leftarrow (is\textsc{Match}([\![array(1)]\!]_{p_j}, [\![\ell]\!]_{p_j}), \ldots, is\textsc{Match}([\![array(m)]\!]_{p_j}, [\![\ell]\!]_{p_j})).$$
   (b) Compute
   $$[\![b_{p_j}]\!] \leftarrow \mathrm{SPiRiT}_{m,p_j}([\![indicator]\!]_{p_j})$$
   where $\mathrm{SPiRiT}_{m,p_j} = S \circ P \circ i \circ R \circ i \circ T$ for S, P, R, T and i the matrices and operator specified in Section 3.4 below.
   (c) Send to the client
   $$([\![b_{p_1}]\!], \ldots, [\![b_{p_k}]\!]) \text{ and } ([\![indicator(b_{p_1})]\!]_{p_1}, \ldots, [\![indicator(b_{p_k})]\!]_{p_k})$$
   (here we slightly abuse notation by addressing entries of *indicator* by the binary representation of the indices). To compute $[\![indicator(b_{p_j})]\!]_{p_j}$ the server applies standard PIR techniques, namely, evaluating on *indicator* and b_j the polynomial $\sum_{i=1}^{m} indicator(i) \cdot is\textsc{Equal}(i, b_{p_j})$; see Section 3.5.
3. The client decrypts and outputs the minimum
$$b^* \leftarrow \min_{j \in [k]} \left\{ b_{p_j} \text{ s.t. } indicator(b_j) = 1 \right\}.$$

3.4 The Components 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, the vector *indicator*) our sketch for first positive

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

In what follows we specify the components of S, P, R, T and i of $\text{SPiRiT}_{m,p}$. Without loss of generality, we assume that m is a power of 2 (otherwise we pad the input *array* by zero entries).

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

The *tree representation* $\mathcal{T}(x)$ *of* x is the full binary tree of depth $\log_2 m$ with labeled 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. The *array data structure* for $\mathcal{T}(x)$ 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 Fig. 4. 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 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 $x = \big(w(m), \cdots, w(2m-1)\big)$.

*The **R**oots matrix* $R \in \{0,1\}^{m\times(2m-1)}$ is a binary matrix with each row having $O(\log m)$ 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 $O(\log m)$ non-zero entries in each row, implying that even when applies on binary vectors the result may consist of values up to m; this in turn ruins the success of $\text{SPiRiT}_{m,p}$ 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 $j \in [2m-1]$,

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

We are now ready to define R: Each row i of the matrix R has values 1 in all entries $j \in \mathsf{Siblings}(i+1)$ (0 otherwise); see Fig. 4. That is,

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

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

Lemma 1 (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 **P**airwise matrix $P \in \{-1,0,1\}^{m \times m}$* is a matrix whose right multiplication by a given 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 $t = Pu = (0,0,0,1,0,0,0)$. 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 sketch in SPiRiT.

*The **S**ketch 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 $\text{isPositive}_p(\cdot)$ (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:

$$\text{isPositive}_p(v(1), \ldots, v(m')) = (v(1)^{p-1} \mod p, \ldots, v(m')^{p-1} \mod p).$$

A crucial issue —that we handle via our multi ring FHE evaluation 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, this is by summing the subset of vector entries, or their negation, 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.

3.5 The Secure Pattern Matching in our Implementations and Experimental Results

Recall that to securely search for an encrypted lookup value ℓ in an encrypted data $array = (array(1), \ldots, array(m))$, the server first computes the encrypted binary vector:

$$indicator = (is\text{Match}(array(1), \ell), \ldots, is\text{Match}(array(m), \ell)) .$$

Our protocol is generic in the sense that we could plug here any pattern matching polynomial $is\text{Match}()$.

Naturally, in order to implement our protocol and produce experimental results we must run our protocol on some concrete $is\text{Match}()$ polynomial. The concrete implementation for $is\text{Match}$ 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 is 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 in to 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 $p > 2$.

3.6 Monte Carlo Secure Search Protocol

In this section we briefly discuss a variant of Protocol 2 introducing a probability of error for saving a factor $\widetilde{O}(\log^2 m)$ in the overall number of multiplications.

The randomized protocol is similar to Protocol 2 except for working over a single random $j \in [k]$, instead of the repetition over all $j \in [k]$ in Protocol 2. We set the parameter k to control the success probability. Specifically, the success probability is set to $1 - \delta$ by taking $k = \frac{1}{\delta} \cdot \frac{\log^2 m}{\log m}$. The success analysis simply follows by observing that the randomized protocol succeed whenever the sampled prime p_j is "good" (see Definition 3, Section 4.1), which happens with probability $1 - \delta$ by our upper bound $\frac{\log^2 m}{\log m}$ on the number bad primes (see proof of Claim 4) and the choice of k. This success probability can be amplified via parallel repetition, as standard. The complexity saving is due to avoiding k repetitions. The degree growth is by a factor of $\frac{1}{\delta}\log(1/\delta)$, which is $O(1)$ for any $\delta = O(1)$.

4 Analysis of our Secure Search Protocol

In this section we analyze our secure search Protocol 2. First we prove Theorem 1 showing this protocol is an efficient secure search protocol; see Sections 4.1-4.2. Next we present Formula 1 specifying a concrete running time estimation for the protocol; see Section 4.3.

Elaborating on the former, we first analyze the complexity and correctness of $\text{SPiRiT}_{m,p}$; see Lemma 2 and 3 in Section 4.1. Then we employ this analysis to prove the correctness and complexity of our Protocol 2; see Theorems 2 and 3 in Section 4.2. The security of Protocol 2 follows immediately by observing that the server processes only semantically secure ciphertexts; this is standard, details omitted.

4.1 Analysis of SPiRiT Sketch for First Positive

In this section we analyze the complexity and correctness of $\text{SPiRiT}_{m,p}$; see Lemmas 2-3. We begin with the complexity analysis.

Lemma 2 ($\text{SPiRiT}_{m,p}$ complexity). $\text{SPiRiT}_{m,p}\colon \{0,1\}^m \to \{0,1\}^{1+\log m}$ *is a polynomial of degree* $O(p^2)$ *that can be evaluated using* $O(m \log p)$ *multiplications. The degree of* SPiRiT *can be reduced to* $O(p \log m)$ *with implementation optimizations.*

Proof. Computing the matrix-vector products when multiplying by S, P, R, T can be done using only additions/subtraction and no multiplications whatsoever, because their entries values in $\{-1, 0, 1\}$ correspond to summing subsets of the vector's entries (or their negation). 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).

Computing isPositive_p for a single entry requires only $\log p$ multiplications, when using repeated squaring. Since we apply isPositive_p on a total of $(2m-1)+m = O(m)$ entries, we get that the total number of multiplications is $O(m \log p)$.

The optimizations for reducing the degree to $p \log m$ are by introducing the following optimization in evaluating $u = i \circ R(w')$ on $w = i \circ T(\mathit{indicator})$: Instead of evaluating the degree p polynomial that sums

up label in the roots as specified by R and reduces them to binary using isPositive_p, we directly compute the OR of these roots labels. The latter gives an identical result to the former, but with a polynomial of degree proportional to the number of roots $\log m$, rather than to the modulus p. □

We next analyze the correctness of $\text{SPiRiT}_{m,p}$, showing a sufficient condition for its success.

Definition 2 (First positive index). *Let $x = (x_1, \ldots, x_m) \in [0, \infty)^m$ be a vector of m non-negative entries. The* first positive index *of x is the smallest index $i^* \in [m]$ satisfying that $x_{i^*} > 0$, or $i^* = 0$ if $x = (0, \ldots, 0)$.*

Definition 3 (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 $j \in \mathsf{Ancestors}(i^*)$ with non-zero labels $Tx(j) \neq 0$ in the tree representation of x, their label $Tx(j)$ is not a multiple of p.*

Lemma 3 ($\text{SPiRiT}_{m,p}$ correctness). *Let p be a prime, m a power of 2, $x \in \{0,1\}^m$, $i^* \in [m] \cup \{0\}$ the first positive index of x. If p is good for x, then $b_p \leftarrow \text{SPiRiT}_{m,p}(x)$ 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 $x = (0, \ldots, 0)$, $b_p \leftarrow \text{SPiRiT}_{m,p}(x)$ is the zero vector $b_p = (0, \ldots, 0)$, as the matrix-vector products and application of isPositive_p operator all evaluates to 0.

For the case $i^* \in [m]$, we prove the following. If p is good, then $b_p \leftarrow \text{SPiRiT}_{m,p}(x)$ is the binary representation of i^*. 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^* its first non-zero, then SPu is the binary representation of i^*; see Claim 1. 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 2. The challenging part is to show that, if p is good, then $u(i^*) = \ldots = u(m) = 1$, as argued below. Put together we conclude that, if p is good, then b_p is the binary representation i^*.

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

Proof. Multiplying u by the pairwise difference matrix $P \in \{-1, 0, 1\}^{m \times m}$ returns the binary vector $t = Pu \mod p$ in $\{0,1\}^m$ defined by $t(k) = u(k) - u(k-1)$ (for $u(0) = 0$). This vector accepts value $t(i^*) = 1$ and values $t(i) = 0$ elsewhere. Multiplying t by the sketch $S \in \{0,1\}^{(1+\log m) \times m}$ returns the binary vector $y = St \mod p$ in $\{0,1\}^{1+\log m}$ specifying the binary representation of i^*. We conclude that $(SP \cdot u \mod p)$ is the binary representation of i^*. □

Claim 2. $u(1) = \ldots = u(i^* - 1) = 0$.

Proof. The definition of the tree matrix T and the roots matrix R implies that $v = RTx$ is the vector of prefix sums of x; See Lemma 1. Namely,

$$v(j) = \sum_{k \in [j]} x(k).$$

Clearly, these prefix sums are zero on all entries $j < i^*$, as $x(1) = \ldots = x(i^* - 1) = 0$. Computing the vector v differs from computing u in that we did not apply the isPositive_p operator. Observe that isPositive_p has one-sided error in the sense that for $z = 0$ it always holds that $\text{isPositive}_p(z) = 0$. Therefore, the first $i^* - 1$ entries remain zero even when applying isPositive_p on the tree representation Tx and on the roots output $RiT(x)$. Namely, $u(1) = \ldots = u(i^* - 1) = 0$. □

Claim 3. If p is good, then $u(i^*) = \ldots = u(m) = 1$.

Proof. Fix $j \geq i^*$. The key observation is that the intersection $\mathsf{Ancestors}(i^*)$ and $\mathsf{Siblings}(j+1)$ is non-empty:

$$\exists k^* \in \mathsf{Ancestors}(i^*) \bigcap \mathsf{Siblings}(j+1).$$

The above holds because the left and right children v_L, v_R of the deepest common ancestor of i^* and $j+1$ must be the parents of i^* and $j+1$ respectively (because $i^* < j+1$), implying that v_L is both an ancestor of i^* and a left-sibling of the ancestor v_R of $j+1$. Namely, v_L is in the intersection of $\mathsf{Ancestors}(i^*)$ and $\mathsf{Siblings}(j+1)$. Now, since $k^* \in \mathsf{Ancestors}(i^*) = A$ then when summing over the integer (i.e., without reducing modulo p),

$$Tx(k^*) = \sum_{j \text{ s.t. } k^* \in \mathsf{Ancestors}(j)} x(j) \geq x(i^*) \geq 1,$$

implying for good p that the above holds also when reducing modulo p:

$$iTx(k^*) = 1 \mod p.$$

Now, when applying the roots matrix to compute $u(j)$, since $k^* \in \mathsf{Siblings}(j+1)$ then the summation includes the summand $iTx(k^*) = 1 \mod p$, so it is strictly positive when computer over the integer. Moreover, the sum remains strictly positive even when reducing it sum modulo p, because the sum is smaller than p (because the sum is over at most $\log m$ values by the property that R has at most $\log m$ non-zero entries in each row, and all these values are bits by as they are entries of iTx). We conclude therefore that

$$iRiTx(j) = \textsc{isPositive}_p \left(\sum_{k \in \mathsf{Siblings}(j+1)} iTx(k) \right) = 1.$$

Namely, we've shown that $u(j) = 1$ for all $j \geq i^*$. □

□

4.2 Analysis of our Secure Search Protocol 2

Consider an execution of Protocol 2 between two parties called client and server. Let the shared/client's/server's inputs be as specified there; denoted, $(\mathrm{E}, m, \mathcal{P} = \{p_1, \ldots, p_k\})$, (sk, ℓ), and $(ek, [\![array]\!] = ([\![array]\!]_{p_1}, \ldots, [\![array]\!]_{p_k}))$ respectively. Then the following holds:

Theorem 2 (Correctness). *Protocol 2 when executed by parties that follow the protocol specifications (i.e., semi-honest) terminates with no output for the server, and with client's output being the binary representation $b^* \in \{0,1\}^{1+\log m}$ of the index of the first match for ℓ in array:*

$$i^* = \min \{ i \in [m] \mid is\textsc{Match}(array(i), \ell) \}.$$

Proof. Recall that the client's output b is the smallest candidate b_{p_j} received from the server that is verified to have $indicator(b_{p_j}) = 1$:

$$b \leftarrow \min \left\{ b_{p_j} \text{ s.t. } indicator(b_{p_j}) = 1 \right\}.$$

By definition of $is\textsc{Match}$ and $indicator$ any b with $indicator(b) = 1$ is the index a matching entry, i.e., so that $is\textsc{Match}(array(b), \ell) = 1$. It remains to prove that b is the *first* match b^*. For this purpose it suffices to prove that $b^* \in \{b_{p_1}, \ldots, b_{p_k}\}$, because if b^* is in this set then it must be the smallest of the verified matches in this set. By Lemma 3, to prove that b^* belongs to this set it suffices to prove that there exists a $p \in \{p_1, \ldots, p_k\}$ so that p is good for $indicator$. The latter follows from Pigeonhole Principle together with bounding the number of prime divisors for the ancestors of i^* in the tree representation of $indicator$; see Claim 4.

Claim 4. There exists a prime $p \in \mathcal{P} = \{p_1, \ldots, p_k\}$ that is good for $indicator$.

Proof. Recall that p is good for $indicator \in \{0,1\}^m$ if it divides non of the non-zero labels of the ancestors of the i^*-th leaf in the tree representation $T \cdot indicator$ of $indicator$ for i^* the first positive index of $indicator$.

Observe that there are at most $\log m$ ancestors for i^*, each labeled by a number in the range $\{0,\ldots,m\}$. Recall that all primes p_j are chosen to be larger than $\log m$, so each label has at most $\log_{\log m} m = \log m / \log\log m$ divisors in $\mathcal{P}$. Taking the union over all labels we conclude that they have at most $\log^2 m / \log\log m$ divisors in $\mathcal{P}$. Now since $|\mathcal{P}|$ is greater than the former, then by Pigeonhole Principle $\mathcal{P}$ must contain a good p. □

□

Theorem 3 (Complexity). *Protocol 2 is a single round protocol with communication complexity of $o(\log^2 m)$ ciphertexts. The server evaluates $o(\log^2 m)$ polynomials (in parallel) each of degree $O(\log^3 m) \cdot d$ and total number of $\widetilde{O}(ms)$ homomorphic multiplications, for d and s the degree and total number of multiplications of isMATCH(). The client's decoding time is $o(\log^2 m)$, and the client's overall running time is proportional to computing $o(\log^2 m)$ encryption and decryption operations.*

Proof. It is straightforward to verify by inspection that Protocol 2 is a single round protocol with communication complexity of $o(\log^2 m)$ ciphertexts.

The server evaluates the composition of the polynomial isMATCH with each of the polynomials SPiRiT_{m,p_j} for $j \in [k]$, where computation is done on encrypted data using homomorphic operations. Recall that $\mathcal{P} = \{p_1,\ldots,p_k\}$ consists of the first $k = 1 + \log^2 m / \log\log m = o(\log^2 m)$ primes larger than $\log m$.

The degree of this composition is the product of the degree d of isMATCH and the degree $O(p_j \log m)$ of SPiRiT_{p_j}. Assigning the upper bound

$$p_1,\ldots,p_k = O(\log^2 m)$$

on the magnitude of $p_1,\ldots,p_k$ (see Claim 5), we get that the degree is at most $O(\log^3 m) \cdot d$.

The total number of homomorphic multiplications computed by the server is the sum of that following: (1) $m \cdot s$ multiplications in computing m applications of isMATCH, and (2) $k \cdot O(m \log p) = o(m \log^2 \log\log m)$ multiplications in computing $\text{SPiRiT}_{m,p_1},\ldots,\text{SPiRiT}_{m,p_k}$ (where the latter holds by the choice of $k = o(\log^2 m$ and the bound $p = O(\log m)$ in Claim 5). Put together we get that the server computes a total of $\widetilde{O}(ms)$ homomorphic multiplications.

The client's decoding algorithm simply selects the minimum of $k = o(\log^2 m)$ values satisfying a test $(indicator(i) = 1)$ that can be verified in $O(1)$ time. So the decoding time is $O(k) = o(\log^2 m)$.

Claim 5. $p_1,\ldots,p_k = O(\log^2 m)$.

Proof. We bound the magnitude of the primes p in $\mathcal{P}$. Recall that by the Prime Number Theorem (see, e.g., in [24]) asymptotically we expect to find $x/\ln x$ primes in the interval $[1,x]$. Thus, we expect to find $\frac{x}{\ln x} - \frac{b}{\ln b} = \Omega(\frac{x}{\ln x})$ primes in the interval $[b,x]$, where the last equality holds for every $b = o(x)$. Assign $x = k \ln k$ for $k = 1 + \log^2 m / \log\log m$ and $b = \log m$. For sufficiently large m there are k primes larger than b in the interval $[b, t \ln t]$; so all the primes in $\mathcal{P}$ are of magnitude at most $p = O(k \ln k) = O(\log^2 m)$. □

□

4.3 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 [36] 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:

$$\boxed{T(m,t) = n \cdot 2t \cdot MUL + 2n(1 + \lceil \log_2 n \rceil) \cdot ADD + 2n \cdot IsPOSITIVE}, \tag{1}$$

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 [23].
- 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 example, in our system (see Section 5) on input parameters $t = 1$ and $m = 255,844,736$ array entries, we have $SIMD = 122$ and $CORES = 64$ resulting in $n = 32,767$ packed ciphertexts; ring size $p = 17$; and measured timings of $ADD = 0.123$ms, $MUL = 62.398$ms, $IsPOSITIVE = 695.690$ms. See Fig. 1 for graph of T on various m, t parameters.

See Figure 1 depicting Formula 1 for various parameters settings ($t = 1, 16, 32, 64$ and m ranging from roughly 10^3 to $2.55 \cdot 10^8$). Comparison to experimentally measured running times there shows that the formula quite accurately predicts the actual running time.

The formula was derived as follows. The summand $n \cdot 2t \cdot MUL$ accounts for computing *indicator* in the protocol, where using n applications of isEqual$_t$, each requiring $2t$ multiplications. Note that this assumes the use of isEqual$_t$ equality test as the isMatch pattern matching polynomial. More generally, when using other isMatch() polynomials, this summand should be replaced to the corresponding running time.

The summand $2n(1 + \lceil \log_2 n \rceil) \cdot ADD$ accounts for the matrix-vector product when computing SPiRiT, i.e., the multiplication by the matrices S, P, R, T. Importantly, we compute these products using only homomorphic additions, and no homomorphic multiplications. This is by observing that these matrices have values only in $\{-1, 0, 1\}$, so that matrix-vector product can be implemented by computing sum of subsets of vector's entries (or their negation), and with no multiplications. Furthermore, the matrices P, R are sparse, requiring only $1 + \log n$ additions per row; the product by the matrix T, despite not being sparse, can be computed with using $n - 1$ additions: one per each internal node in the tree summing up the labels of its children; likewise, the product by the matrix S can be computed with $\log n$ additions with further optimizations.

The summand $2n \cdot IsPOSITIVE$ accounts for the applications of isPositive operator on n internal nodes in the tree representation $T \cdot indicator$ and on the n entries of $RiT \cdot indicator$.

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 HELib library [22] implementation for the Brakerski-Gentry-Vaikuntanthan's FHE scheme [7] together with the Smart-Vercauteren [36] and Gentry-Halevi-Smart [18] Single Instruction Multiple Data (SIMD) optimization. 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 5.1-5.3.

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 2 in protocol 2), and returns to the client a short list of encrypted candidates for the first match for ℓ is *array*. The web browser then decrypts this candidates list on 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 share the same security key.

Hardware. Our system is generic but in this section discuss how we evaluate 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 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 [22] 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 and can be found in [1].

Security. Our system and all the experiments below use a security key of 80 bits of security. 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 zeroes except for a random index. In case $t = 64$, *array* is a vector of m random 64-bits entries.

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. This SIMD parameter is determined by the 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; this is expected to yield a speedup by factor of roughly 10 over our reported results.

Experiments. We ran our secure search algorithm (Step 2 in Protocol 2), running on the server the probabilistic version in which a single p is chosen; see Section 3.6. 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 $\frac{1}{2} \cdot 10^9$). 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 SPiRiT sketch for first positive.

5.3 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, and under a second for the deterministic variant; experiments are on our old 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 computed, 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, in 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 of *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).

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" [44]). 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.

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 a single machine on Amazon's cloud for growing database *array* size (left column) over encrypted database. Middle column shows the running times for SPiRiT sketch for first positive of length m binary array ($t = 1$). Right column shows the running times for secure search in a length m array of 64-bits records ($t = 64$).

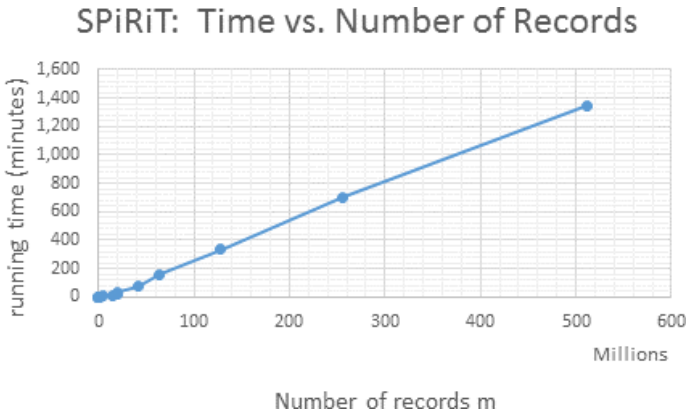

Fig. 5. SPiRiT running time as a function of the number of records. Number of records ranged from $m = 90,047$ to $m = 511,697,280$ in roughly doubling values; measured running times started at under a second (0.7sec) and reached up to under a day (22.5 hours).

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.

Comparison to theoretical analysis. Theorem 3 show that the degree of our secure search protocol is the degree of SPiRiT times the degree of the used pattern matching subroutine, and that the degree of SPiRiT is

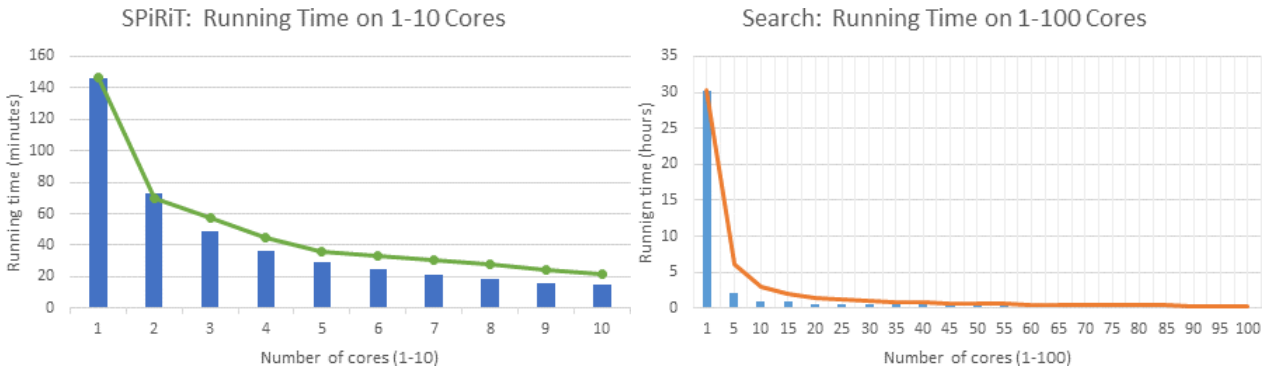

Fig. 6. Left: Server's running time (minutes) for computing SPiRiT on a million records using 1-10 cores (bars); compared with running time reduction by a factor of 1/#cores (curve). Right: Server's running time (hours) for computing Secure Search on a billion records using 1-100 cores (bars); compared 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.

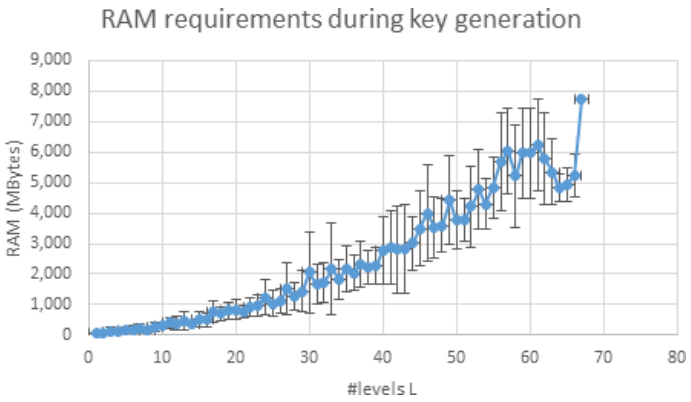

Fig. 7. RAM requirements for the evaluation key as a function of the multiplicative depth L (x-axis), averaged over the plaintext modulus values $p = 5, 7, 11, 13, 17, 19, 23, 29, 257, 1249, 4241, 16519, 64091$ (curve and std error bars); demonstrating that RAM requirements are typically around 4Gb,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

poly-logarithmic in the number of records. The multiplicative depth of SPiRiT (i.e., log of its degree) is therefore doubly logarithmic in the number of records: $O(\log\log m)$. This is demonstrated by the multiplicative depth as measured in our experiments; see Figure 8.

Why the curves are not smooth? Each of the curves in Fig. 1 has 4–5 non-continuous increasing steps. These are not artifacts or noise. They occur every time whenever there is an increase in the number of primes numbers $|\mathcal{P}|$ used in Protocol 2. Recall that the cardinality $|\mathcal{P}|$ grows as a function of the number of records m (see there), affecting the ring size p that are used by the SPiRiT sketch, which in turn increases the depth of the polynomial realizing isPositive_p, and consequently the overall server's running time.

6 Conclusions

In this work we present the first secure search protocol on FHE encrypted lookup value and searched data, achieving all the following: (1) efficient communication consisting of a single round with communication volume proportional to the lookup value and search outcome; (2) efficient client with running time polynomial in the size of the retrieved records; (3) efficient server evaluating a polynomial of degree poly-logarithmic in

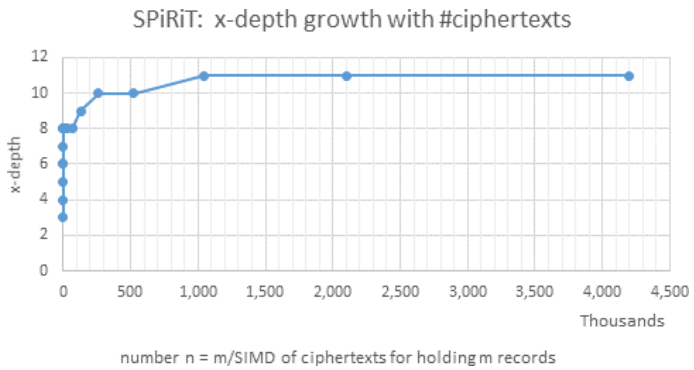

Fig. 8. The graph shows SPiRiT's multiplicative depth (equivalently: log of the degree, or the number of levels parameter L in HElib) as a function of the number of ciphertexts $n = m/SIMD$ for holding the data array (for m the number of records). By our analysis we expect the depth to be $O(\log \log n)$.

the number of records in the searched database; (4) guaranteeing the semantic security for the lookup value and database both at rest and during search. We implemented our protocol in an open source library based on HELib implementation for the Brakerski-Gentry-Vaikuntanthan's FHE scheme, 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. This is counter to the wide prior belief on the FHE's computational overhead being too prohibitive for secure search implementations.

7 Acknowledgment

We thank Shai Halevi, Craig Gentry, Shafi Goldwasser and Vinod Vaikuntanathan for helpful discussions and comments.

References

1. A. Akavia, D. Feldman, and H. Shaul. SearchLib: Open library for fhe search, with an example system, will be published upon acceptance., 2018.
2. A. Akavia, D. Feldman, and H. Shaul. Secure search on the cloud via coresets and sketches. *arXiv preprint arXiv:1708.05811*, 2017.
3. A. Akavia, D. Feldman, and H. Shaul. Secure database queries in the cloud: Homomorphic encryption meets coresets, 2018. sumbitted.
4. A. Akavia, D. Feldman, and H. Shaul. Secure optimization and learning in the cloud, 2018. in preparation.
5. A. Akavia and M. Leibovitch. Secure search via binary raffle (in preparation), 2018.
6. C. Bösch, P. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)*, 47(2):18, 2015.
7. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.
8. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 97–106, 2011.
9. G. S. Çetin, W. Dai, Y. Doröz, W. J. Martin, and B. Sunar. Blind web search: How far are we from a privacy preserving search engine? *IACR Cryptology ePrint Archive*, 2016:801, 2016.
10. H. Chen, K. Laine, and P. Rindal. 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*, pages 1243–1255, 2017.

11. J. H. Cheon, M. Kim, and M. Kim. Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *IEEE Trans. Information Forensics and Security*, 11(1):188–199, 2016.
12. J. H. Cheon, M. Kim, and K. E. Lauter. Homomorphic computation of edit distance. In *Financial Cryptography Workshops*, pages 194–212, 2015.
13. J. H. Cheon, M. Kim, and K. E. Lauter. Homomorphic computation of edit distance. In *Financial Cryptography Workshops*, pages 194–212, 2015.
14. Y. Doröz, B. Sunar, and G. Hammouri. 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*, pages 195–207, 2014.
15. N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. 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, pages 201–210. JMLR.org, 2016.
16. C. Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009. AAI3382729.
17. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.
18. C. Gentry, S. Halevi, and N. Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482, 2012. Springer.
19. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.
20. T. Graepel, K. Lauter, and M. Naehrig. Ml confidential: Machine learning on encrypted data. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, pages 1–21, Berlin, Heidelberg, 2013. Springer-Verlag.
21. T. H, X. Jiang, X. Wang, S. Wang, H. Sofia, D. Fox, K. Lauter, B. Malin, A. Telenti, L. Xiong, and L. Ohno-Machado. Protecting genomic data analytics in the cloud: state of the art and opportunities. *BMC Med Genomics*, 9(1):63, Oct 2016.
22. S. Halevi and V. Shoup. HElib - An implementation of homomorphic encryption. `https://github.com/shaih/HElib/`, 2013.
23. S. Halevi and V. Shoup. Algorithms in helib. In *34rd Annual International Cryptology Conference, CRYPTO 2014*. Springer Verlag, 2014.
24. G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford, fourth edition, 1975.
25. P. Indyk, H. Q. Ngo, and A. Rudra. Efficiently decodable non-adaptive group testing. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1126–1142. SIAM, 2010.
26. M. Kim, H. T. Lee, S. Ling, S. Q. Ren, B. H. M. Tan, and H. Wang. Better security for queries on encrypted databases. *IACR Cryptology ePrint Archive*, 2016:470, 2016.
27. M. Kim, H. T. Lee, S. Ling, B. H. M. Tan, and H. Wang. Private compound wildcard queries using fully homomorphic encryption. *IEEE Transactions on Dependable and Secure Computing*, 2017.
28. M. Kim, Y. Song, and J. H. Cheon. Secure searching of biomarkers through hybrid homomorphic encryption scheme. *BMC medical genomics*, 10(2):42, Jul 2017.
29. F. Krell, G. Ciocarlie, A. Gehani, and M. Raykova. Low-leakage secure search for boolean expressions. In *CryptographersâĂŹ Track at the RSA Conference*, pages 397–413. Springer, 2017.
30. C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely outsourcing middleboxes to the cloud. In *NSDI*, pages 255–273, 2016.
31. K. E. Lauter, A. López-Alt, and M. Naehrig. Private computation on encrypted genomic data. *LATINCRYPT*, 8895:3–27, 2014.
32. K. E. Lauter, A. López-Alt, and M. Naehrig. Private computation on encrypted genomic data. *IACR Cryptology ePrint Archive*, 2015:133, 2015.
33. F. K. Loy. *Secure Computation Towards Practical Applications*. Columbia University, 2016.
34. W. Lu, S. Kawasaki, and J. Sakuma. Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. *IACR Cryptology ePrint Archive*, 2016:1163, 2016.
35. M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.
36. N. Smart and F. Vercauteren. Fully homomorphic SIMD operations. In *Designs, codes and cryptography*, pages 1–25 Springer, 2014.

37. V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.
38. R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.
39. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.
40. S. S. Roy, F. Vercauteren, J. Vliegen, and I. Verbauwhede. Hardware assisted fully homomorphic function evaluation and encrypted search. *IEEE Transactions on Computers*, 2017.
41. S. H. Shafi Goldwasser, Vinod Vaikuntanathan. personal communication.
42. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.
43. B. Wang. *Search over Encrypted Data in Cloud Computing*. PhD thesis, Virginia, 2016.
44. B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, 1999.
45. Z. Xia, X. Wang, X. Sun, and Q. Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):340–352, 2016.
46. A. C.-C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.
47. M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, pages 65–76, New York, NY, USA, 2013. ACM.
48. M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba. New packing method in somewhat homomorphic encryption and its applications. *Sec. and Commun. Netw.*, 8(13):2194–2213, Sept. 2015.