Adi Akavia*, Craig Gentry, Shai Halevi, and Max Leibovich

Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free

Abstract: We present a novel *secure search* protocol on data and queries encrypted with Fully Homomorphic Encryption (FHE). Our protocol enables organizations (client) to (1) securely upload an unsorted data array $x = (x[1], \ldots, x[n])$ to an untrusted honest-but-curious sever, where data may be uploaded over time and from multiple data-sources; and (2) securely issue repeated search queries q for retrieving the first element $(i^*, x[i^*])$ satisfying an agreed matching criterion $i^* = \min\{\, i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$, as well as fetching the next matching elements with further interaction. For security, the client encrypts the data and queries with FHE prior to uploading, and the server processes the ciphertexts to produce the result ciphertext for the client to decrypt. Our secure search protocol improves over the prior state-of-the-art for secure search on FHE encrypted data (Akavia, Feldman, Shaul (AFS), CCS'2018) in achieving:

- *Post-processing free* protocol where the server produces a ciphertext for the correct search outcome with overwhelming success probability. This is in contrast to returning a list of candidates for the client to post-process, or suffering from a noticeable error probability, in AFS. Our post-processing freeness enables the server to use secure search as a sub-component in a larger computation without interaction with the client.
- *Faster protocol:* (a) Client time and communication bandwidth are improved by a $\log^2 n / \log\log n$ factor. (b) Server evaluates a polynomial of degree linear in $\log n$ (compare to cubic in AFS), and overall number of multiplications improved by up to $\log n$ factor. (c) Employing only GF(2) computations (compare to GF(p) for $p \gg 2$ in AFS) to gain both further speedup and compatibility to all current FHE candidates.
- *Order of magnitude speedup exhibited by extensive benchmarks* we executed on identical hardware for implementations of ours versus AFS's protocols.

Additionally, like other FHE based solutions, our solution is setup-free: to outsource elements from the client to the server, no additional actions are performed on x except for encrypting it element by element (each element bit by bit) and uploading the resulted ciphertexts to the server.

Keywords: Secure search, Fully homomorphic encryption, Randomized algorithms, Universal hash functions

DOI 10.2478/popets-2019-0038
Received 2018-11-30; revised 2019-03-15; accepted 2019-03-16.

1 Introduction

Following the rapid advancement and widespread availability of cloud computing it is a common practice to outsource data storage and computations to cloud providers. Placing cleartext (i.e unencrypted) data on the cloud compromises data security. To regain data privacy one could encrypt the data prior to uploading to the cloud. However, if using standard encryption (e.g. AES), this solution nullifies the benefits of cloud computing: when given only ciphertexts the cloud provider cannot process the underlying cleartext data in any meaningful way.

Fully homomorphic encryption (FHE) [22, 23, 49] is an encryption scheme that allows processing the underlying cleartext data while it still remains in encrypted form, and without giving away the secret key (see Definition 2.1). With FHE it is possible for the client to securely outsource computations to the server as follows: The client first encrypts its data x with an FHE scheme to obtain the ciphertext $[\![x]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x)$, and sends $[\![x]\!]$ to the server. The server can now compute any function f on the underlying clear-text data x by evaluating a homomorphic version of f on the ciphertext $[\![x]\!]$. The outcome of this computation is a ciphertext $[\![y]\!] \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, [\![x]\!])$ that decrypts to the desired output $y = f(x)$. The server can now send the ciphertext $[\![y]\!]$ to the client who would decrypt $y \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![y]\!])$ to obtain the result.

The homomorphic computations achievable by the known FHE candidates (e.g. [7, 19, 24, 43]) are specified

***Corresponding Author: Adi Akavia:** University of Haifa, E-mail: adi.akavia@gmail.com
Craig Gentry: IBM Research, E-mail: craigbgentry@gmail.com
Shai Halevi: IBM Research, E-mail: shaih@alum.mit.edu
Max Leibovich: University of Haifa, E-mail: max.fhe.phd@gmail.com

by a polynomial over a finite ring (i.e. by repeated application of homomorphic-addition and homomorphic-multiplication for that ring). For example, for data in binary representation, bitwise operations on plaintext bits (addition and multiplication modulo 2) can be replaced by their homomorphic counterparts on encrypted bits (homomorphic-addition and homomorphic-multiplication).

Key factors influencing the running-time of such homomorphic computations are the degree and overall multiplications of the polynomial. Leading to the main two constraints in designing algorithms that compute on FHE encrypted data: they must be realized by a polynomial of low degree and low amount of overall multiplications.

Note that this FHE approach for securely outsourcing to the server the computation of $y = f(x)$ has the benefits of requiring only a single round of communication, and with low communication bandwidth (communicating only the encrypted input $[\![x]\!]$ and output $[\![y]\!]$). Furthermore, the server in this protocol learns no new information about x or y (assuming the FHE is semantically secure).

Secure search is a fundamental computational problem, useful in numerous data analysis and retrieval tasks. An abundance of proposed solutions were presented to solve it using different cryptographic tools (see Section 1.1 and Tables 1-2). In particular, Gentry [22, 23] proposed using FHE to securely search on encrypted data.

In this work we address the natural and simple formulation for *secure search on FHE encrypted data* (*secure search*) as considered by [2]: Secure search is a two party protocol between a server and a client. The server holds an unsorted array $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$ of encrypted elements (not necessarily distinct) that were previously encrypted and uploaded by the client, as well as a specification of a predicate $\mathsf{IsMatch}(a, b) \in \{0, 1\}$ specifying the matching condition. The client submits encrypted queries $[\![q]\!]$ to the server in order to retrieve the first matching element. The server returns to the client the encrypted index and element pair $[\![y]\!] = \big([\![i^*]\!], [\![x[i^*]]\!]\big)$ for i^* the index of the first element satisfying the matching condition, $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$. See detailed definition and extensions in Section 3 and 5.

Restrictions on protocols. Note that the above secure search formulation, as addressed in this work, focuses on protocols that involve (a) a **single server**, and where the client-server interaction is of (b) a **single round**, and (c) **low communication** complexity. Furthermore, (d) **no initial setup** is performed on x except for encrypting it element-by-element (each element encrypted bit-by-bit) and uploading the resulting ciphertexts to the server.

We point out that the latter condition, among other things, prevents speeding up the search by using standard data structures such as search-trees, hash-tables, or sorted arrays (on top of, or instead of, the encrypted unsorted array $[\![x]\!]$). A *linear scan lower bound* is thus implied by the addressed formulation, even if we were to search on clear-text data. This restriction is nonetheless motivated by many use-cases, as discussed next.

Use-cases motivating the aforementioned no-setup restriction arise in settings where, for example:

- Matching criteria are unknown in advance, thus precluding appropriate indexing or sorting at setup;
- High dimensional range queries, where index size is exponential in the number attributes and infeasible to compute or store;
- Streaming data with client discarding each element immediately after encrypting and uploading to the server, thus precluding client's setup or maintenance of the desired data-structures (and where for the server, seeing only ciphertexts, secure maintenance of advanced data structures seems even harder than secure search);
- Low capacity clients that are too weak to run setup over the entire cleartext array prior to encrypting and uploading it to the server;
- Fragmented data uploaded to the server from multiple distinct client endpoints (data-sources) with no single endpoint that can perform setup over the entire cleartext data.

The single-round and low-communication restrictions are motivated by use-cases in settings where communication is a major bottleneck, e.g. in being intermittent or unreliable, or where communicating is with data-sources that are mostly offline, or have restricted battery capacity as in sensors-networks or some Internet-of-Things (IoT) devices.

The single server restriction is motivated, not only by the simplicity of such architecture, but also by its stronger security guarantee: requiring no non-collusion assumption on servers.

Threat model. We address computationally-bounded semi-honest adversaries that follow the protocol but may try to learn additional information. Our security requirement is that adversaries controlling the server cannot distinguish between two adversarially-chosen equal size queries or data arrays. See Section 3.3.

The leakage of our protocols include only size information (specifically, upper-bounds on array size, elements' sizes, number of queries, and queries' sizes); see detailed leakage discussion in section 3.3.

1.1 Prior Works

We survey related works, focusing primarily on works addressing similar secure-search formulation as addressed in this work: *single-server, single-round, low-communication* and *no-setup*. See Tables 1-2.

1.1.1 Secure Search on FHE Encrypted Data

The most relevant works are those addressing the same secure-search formulation as considered in our work (aka, secure search on FHE encrypted data); See above and Definition 3.1.

Folklore solutions for secure search on FHE encrypted data suffered from inefficient server runtime due to evaluating degree $\Omega(n)$ polynomials, for n the number of elements; see discussion in [2].

SPiRiT. The prior state-of-the-art for secure-search on FHE encrypted data appeared in a recent work of Akavia, Feldman and Shaul CCS'2018 [2], where the server evaluates a polynomial of logarithmic degree $\log^3 n$ (instead of degree at least linear in the folklore solution). Their work proposes both a deterministic and a randomized variant.

Their deterministic variant (SPiRiT Det.) uses modern data summarization techniques known as sketches [60] alongside multi-ring simultaneous evaluations of their search polynomial to retrieve a poly-logarithmic short list of candidates for the first matching element. This novel technique essentially reduces the degree of the polynomial evaluated by the server from linear to poly-logarithmic in the number of elements n.

Their randomized variant (SPiRiT Rand.) offers an efficiency improvement by working over a single random ring instead of several different rings. The disadvantage of this randomized variant is that it achieves an error probability that is only polynomially small in n (i.e., noticeable error) rather than a negligible error probability.

1.1.2 Other Related Works

We next discuss other related methods and works.

Setup vs. no-setup – server's efficiency gap. There is a major efficiency gap between works allowing initial setup to works disallowing setup, as in our work. This is because setup allows sub-linear search time (e.g. using indexing, search trees or hash table), whereas disallowing setup necessitates –even on cleartext data– a linear scan of the data. Our work focuses on the no-setup case.

Secure two-party computation (2PC w/o FHE). Seminal works dating back to the 1980s [26, 62] showed that two parties can compute any polynomial-time computable function of their private inputs via an interactive protocol that reveals no information beyond what can be inferred from the function's output. In particular, parties can securely compute the search functionality.

However, secure two-party protocols preceding the constructions of FHE schemes (2PC w/o FHE) suffer from a communication complexity, and hence also the client's time, grows with the complexity of the computed function. This is in contrast to growing only with the input and output sizes $|q| + |(i, x[i])|$ in FHE based solutions.

We note that, while we focus on secure search in the two parties settings (client and a single server), promising results have been shown for settings where the server can be partitioned into several non-colluding entities that secret-share the data; a comprehensive survey of such works is beyond our scope. A few examples include [3, 5, 59], all addressing search on cleartext data held by the multiple servers, namely, protecting the query but not the data against the server.

Searchable Encryption (SE) focuses on inherent efficiency versus security trade-off when searching on encrypted data. Specifically SE focuses on achieving sublinear search time. Main primitives for SE include searchable symmetric encryption (SSE) and public key encryption with keyword search (PEKS) with schemes first introduced by Song et al. [56] and Boneh et al. [4] respectively. See also [6] for a thorough survey.

To achieve sublinear time SE incorporates setup such as sorting, indexing or usage of auxiliary data structures. Furthermore SE deliberately leaks information to enable highly efficient search over encrypted data. This leakage typically includes the access pattern (which elements match a given keyword) and/or search pattern (whether different queries were generated for the same searched keyword).

Starting with the work of [17], research on SE formalized security by defining a *leakage profile* that characterizes the information an adversary may learn. In some cases, an adversary can exploit a scheme's leakage to completely reveal the content of elements and queries it comes across. See discussions and example attacks in [1, 10, 25, 29, 30, 33, 42, 47, 64].

Setup Allowed	Security				Sub-Linear Complexity in $\|x\|$?	Single Round
	Hide Query Content	Hide Elements Content	Hide Search Pattern	Hide Access Pattern		
Cleartext	×				✓ $_{(iii)}$	✓
SE	✓ $_{(i)}$	✓ $_{(i)}$	× $_{(ii)}$	× $_{(ii)}$		✓
SE + ORAM	✓	✓	✓	✓		×
PIR by Keywords	✓	×	✓	✓		✓

Table 1. Comparing search solutions that **allow setup** in the single-server settings. Rows correspond to related works (see Section 1.1.2); Columns correspond to properties; Cells contain ✓ if the column's property is attained by the row's work (× if not attained). **Acronyms and Abbreviations:** Comm. – Communication; SE – Searchable Encryption; PIR – Private Information Retrieval; FHE – Fully Homomorphic Encryption; ORAM – Oblivious RAM. **Comments:** (i) Known methods exploit leakage of SE protocols to obtain query and/or elements content [1, 10, 25, 29, 30, 33, 47, 64]. (ii) SE deliberately leaks information to enable highly efficient search over encrypted data, see additional info in [6]. (iii) Sub-linear client, server and communication complexity in $|x|$.

In contrast, in FHE based solutions such as our work there is no leakage other than size information; see detailed leakage discussion in Section 3.3.

SE using Oblivious RAM (SE+ORAM). ORAM [27] is a cryptographic primitive that allows a client to conceal its access pattern to a remote storage held by the server via continuous re-shuffling of a dedicated structure and re-encryption of the data as it is being accessed. Typical ORAM constructions (e.g. [51, 57]) achieve poly-logarithmic server run-time by requiring the client to perform a "download-decrypt-compute-encrypt-upload" each round, for at least $\mathcal{O}(\log n)$ rounds.

ORAM can be combined with other techniques to suppress leakage in SE. This was demonstrated in several recent works, including: TWORAM [21] in which ORAM is utilized together with garbled circuits [61] to hide search and access patterns while achieving constant number of rounds; and [34] that provides generic compilers that suppress leakage in SE schemes by using Square-Root ORAM [27].

Private Information Retrieval (PIR) [16] in a single server scenario [39, 44], allows a client holding the *physical address* $i \in [n]$ to retrieve the ith element $x[i]$ from a data array $x = (x[1], \ldots, x[n])$ held by the server. The above is achieved while ensuring the server learns no new information on i or $x[i]$ and while keeping the communication complexity strictly smaller than $|x|$. The state-of-the-art communication complexity, as achieved by FHE based PIR [8, 18, 22, 23], is proportional only to the size of i, $x[i]$ and the security parameter. We note however that the server's run-time in a single server PIR (whether or not FHE based) is inherently linear in $|x|$.

PIR-by-Keywords via setup on unencrypted data. PIR-by-keywords protocols [11, 15, 50] remove the requirement to know the physical address i of the sought element $x[i]$, and allow instead to retrieve elements using keywords search. These works however address different settings than our work: their server holds *unencrypted data* and performs *setup* on that data to produce auxiliary data-structures used to speedup search (in contrast to the server's holding encrypted data and performing no-setup in our work).

Several disadvantage of this PIR-by-keyword approach make it unsuitable for the use-cases and settings considered in our work: (1) Their server holds unencrypted data, offering no data protection against the server. This is inherent for their setup, as efficient setup on encrypted data is a challenging problem not addressed by these works.[1] (2) They require setup for producing their data-structures, which is unsuitable for use-cases where setup is disallowed or infeasible; See use-cases examples above. (3) Their search-space is restricted by the initial setup choices, e.g., the keywords used in producing their index [11, 50].

In contrast, in our work (1) Data is encrypted by the client to guarantee data protection against server. (2) No initial setup or maintenance of additional data structures is required, neither by the server nor by the client. (3) We enable arbitrary matching criteria to be chosen by the client, on the fly, for each query.

Private Set Intersection (PSI) [20] enables two parties each holding a private set to securely compute the intersection of their sets. PSI protocols were constructed using various cryptographic primitives (e.g. [37, 45, 46],

1 In contrast, in standard PIR when given the physical address i, it is irrelevant whether x is encrypted, because the server simply retrieves whatever content is in $x[i]$.

including FHE [12]). PSI protocols can be employed to solve the decision problem of whether the lookup value (size 1 set) appears in the data array (size n set); however, with server holding *unencrypted data*.

Secure Pattern Matching (SPM) on FHE encrypted data [13, 14, 35, 36, 40, 58, 63], given an encrypted lookup value, returns a vector of n ciphertexts $(c_1, \ldots, c_n)$, for n the number of elements, where c_i indicates whether the ith data element is a match to the lookup value (or sometimes returning only a YES/NO answer of whether a match exists). The main drawback of these protocols is that the communication complexity and client's running time are proportional to the number of stored elements $\Omega(n)$.

1.2 Our Contributions

In this work we present a new and improved solution for secure search on FHE encrypted data, analyze its efficiency compared to the prior state-of-the-art and demonstrate its concrete run-time performance by providing an implementation (built on top of HElib C++ library [31]) together with extensive experiments.

Our secure-search protocol is a single-server, single-round, low-communication protocol that requires no initial setup or maintenance of additional data-structures.

Our protocol is compatible with generic matching criteria, such as exact and wild-card matching; similarity search in metric spaces, e.g., with Hamming/Euclidean/Edit distance; Boolean and range queries; and so forth. See Section 5 for specific instantiations examples; efficiency improvements via universal hashing; and employment for fetch-next queries.

The client's complexity is optimal in the sense of only encrypting the input and decrypting the output. The communication consists only of the encrypted input and output. The server sees only ciphertexts for both data and queries, encrypted with FHE in a black-box fashion and compatible with all known FHE candidates. The server evaluates a search polynomial over the encrypted data and encrypted query. This polynomial for computing both index i^* and element $x[i^*]$ is of degree $\log(n/\varepsilon) \cdot d$ and overall multiplication $n\,(\log(n/\varepsilon) + \mu + w)$. Here n is the number of data elements, ε the failure probability, w the binary representation length of $x[i^*]$, and d, μ the degree and overall multiplications respectively for the polynomial realizing the matching criterion. See Table 3.

The security guarantee against semi-honest adversaries controlling the server is that our protocol leaks no information on data and queries, except for size information (aka, full-security). Namely, the leakage profile consists solely of upper bounds on the counts and sizes of queries and data elements. In particular, the adversary cannot tell whether two queries are for the same keyword, or whether the client issued a "fresh" query or a "fetch-next" query, etcetera. See Section 3.3.

Comparison to prior works. We next compare our protocol to prior works on secure search, focusing on single-server, single-round protocols, and discussing both works that allow and disallow setup; See Table 1 and Table 2, respectively.

Our work is incomparable to works allowing setup; See SE, SE+ORAM and PIR-by-Keywords in Section 1.1.2. On the one hand, setup enables attaining search with sub-linear server complexity, which is impossible without setup even on cleartext data and query. On the other hand, we attain a stronger security guarantee of hiding all the following: data content, query content, access pattern, and search pattern; in contrast to leaking at least some of the former in the aforementioned works; See Table 1.

Our work strictly improves over prior secure-search works that disallow setup (see Folklore, SPiRiT, 2PC w/o FHE, PIR, PSI and PSM in Section 1.1), in the following sense. Our secure-search protocol is the first to simultaneously attain all desired properties that follows (Properties 1-10, Section 3.2): full security (i.e., completely hiding all the following: query content, data elements content, search pattern, access pattern); efficient client and communication (i.e., polynomial in input and output size, and not in the time to compute the search functionality), single and efficient server (in the sense of evaluating over encrypted data a polynomial of degree poly-logarithmic in the number of data elements); unrestricted search functionality; retrieval of both index and element; post-processing free; negligible error probability; and compatibility with all current FHE schemes. In contrast, all prior secure search solutions achieve only a strict subset of these properties. See Table 2.

In particular, when comparing to the prior state-of-the-art secure-search on FHE encrypted data (SPiRiT) [2] our protocol offers the following contributions:

Contribution 1. Our protocol simultaneously achieves both the properties of post-processing free client and negligible error probability. In contrast, the protocols of [2] achieve either post-processing free client or negligible error probability, but not both. Simultaneously achieving both properties, as in our work, is highly motivated as it allows the server to employ secure search

Setup Disallowed	Security	Sub-Linear Complexity in $\|x\|$?			Allows Multiple Matches	Retrieval of Index and Element	Post Processing Free	Negligible Error Probability	Compatibility with all FHE Schemes
		Server	Client	Comm.					
Cleartext	×	×	✓	✓	✓	✓	✓	✓	**N/A**
2PC w/o FHE	✓		×	×	✓	✓	✓	✓	**N/A**
PIR	✓		✓	✓	×	✓	✓	✓	✓
PSI	✓		✓	✓	×	×	✓	✓	✓
SPM	✓		×	×	✓	×	×	✓	✓
Folklore	✓		✓	✓	✓	✓	✓	✓	✓
SPiRiT Det.	✓		✓	✓	✓	✓	×	✓	×
SPiRiT Rand.	✓		✓	✓	✓	✓	✓	×	×
Binary Raffle	✓		✓	✓	✓	✓	✓	✓	✓

Table 2. Comparing search solutions that **disallow setup** in the single-server, single-round settings. Rows correspond to related works (see Section 1.1); Columns correspond to properties (see Section 3.2); Cells contain ✓ if the column's property is attained by the row's work (× if not attained, N/A if not applicable). **Acronyms and Abbreviations:** Comm. – Communication; 2PC – Two Party Computation; PIR – Private Information Retrieval; FHE – Fully Homomorphic Encryption; PSI – Private Set Intersection; SPM – Secure Pattern Matching; Folklore – Natural secure search on FHE encrypted data; SPiRiT Det. and Rand. – deterministic and randomized protocols of AFS [2]; **Binary Raffle – This Work**. **Comments:** (i) All the works in this table attain **Single Round** protocols.

as a sub-component in a larger computation without interaction with the client.

Contribution 2. Our secure search solution is asymptotically faster than [2], in attaining: (1) Optimal client run-time in the sense of requiring only encrypting the input and decrypting the output. (2) Considerable improvement of the server's run-time: we reduce the degree of the evaluated polynomial from cubic to linear in $\log n$, and from linear to logarithmic in $1/\varepsilon$, and reduce the overall multiplications by up to $\log n$ factor. See Table 3.

Contribution 3. Our secure search solution requires computations solely over GF(2) (instead of GF(p) for primes $p > 2$ in [2]). This leads to compatibility with all currently known candidate FHE schemes including GSW [24], unlike [2]. This also allows further run-time speedup when using current FHE schemes implementations, including HElib [31] that implements BGV [7] scheme. The reason for the speedup is that in all current FHE schemes, working over GF(p) for larger primes $p > 2$ causes a general slowdown of all the homomorphic operations and size inflation of the keys and ciphertexts.

Contribution 4. Our secure search solution is concretely faster than [2] by an order of magnitude. This is demonstrated by our implementation, based on the FHE HElib C++ library [31], and our extensive run-time benchmarks experiments, performed on a mid-range Linux server of 16 CPU cores and 16GB RAM. A few examples of comparing our results on same server and with similar parameters for bits per element, execution time and error probability follow; See more details in Section B.3.

(i) We securely search on $\approx 3 \times 10^6$ (in contrast to $\approx 0.2 \times 10^6$ for SPiRiT) 16-bit elements in 4.5 hours, with error probability 2^{-80}.
(ii) We securely search on $\approx 3 \times 10^6$ (in contrast to $\approx 0.3 \times 10^6$ for SPiRiT) 16-bit elements in 1 hour, with error probability 1/2.
(iii) We securely search on $\approx 1 \times 10^6$ (in contrast to running out of RAM and being unable to complete the experiment for SPiRiT) 64-bit elements in 1 hour, with error probability 1/2.
(iv) We securely search on $> 10 \times 10^6$ (in contrast $\approx 1.5 \times 10^6$ for SPiRiT) 1-bit elements in 1 hour, with error probability 1/2.

1.3 Our Techniques Highlights

When considering secure search over unsorted FHE encrypted data, approaches like binary-search are rejected immediately.

The approach proposed by Akavia et. al. [2] for solving secure search on FHE encrypted array includes the following steps: (1) Obtaining a binary array of indicators after executing the desired IsMatch predicate between the given query and each array element; (2) Calculating an array of prefix-sums of the array of binary indicators; (3) Transforming the prefix-sums array to a binary step-function array with value 1 at every non-zero prefix-sum, namely, the first 1 bit is in the index of the first match; (4) Transforming the step-function array to a selector array where only the index of the first match contains 1 (and all other indices contain 0); (5) Utilizing this selector array to calculate and return this first match (index and element).

Table 3. Complexity comparison of first match index (i^*) computation phase between SPiRiT (rows (i)-(ii)) vs. our work (rows (iii)-(v)). Notations: n – array size; ε – failure probability; d, μ – degree and overall multiplications of IsMatch; $k = \log^2 n / \log\log n$; $\alpha = \mathcal{O}(\log(1/\varepsilon))$; c – a constant depending on the density of prime numbers.

	• **Server's Degree,** • **Server's Overall Multiplications** • **Client's Decryptions**
(i) *SPiRiT* Det.	• $\mathbf{\log^3(n) \cdot d}$ • $\mathbf{k \cdot n \cdot \left(\log^2(n) + \mu\right)}$ • $\mathbf{k \cdot \log(n)}$
(ii) *SPiRiT* Rand.	• $\mathbf{\frac{c}{2\varepsilon} \log^3(n) \cdot d}$ • $\mathbf{n \cdot \left(\log\left(\frac{n}{\varepsilon} \cdot \frac{c}{2} \cdot \log n\right) + \mu\right)}$ • $\mathbf{\log(n)}$
(iii) *Binary Raffle*	• $\mathbf{\log(n/\varepsilon) \cdot d}$ • $\mathbf{n \cdot (\log(n/\varepsilon) + \mu)}$ • $\mathbf{\log(n)}$
(iv) *Binary Raffle* + Universal Hash	• $\mathbf{2 \cdot \log^2(2n/\varepsilon)}$ • $\mathbf{n \cdot (3 \cdot \log(2n/\varepsilon))}$ • $\mathbf{\log(n)}$
(v) *Binary Raffle* + Client Probability Amplification	• $\mathbf{\log(3n) \cdot d}$ • $\mathbf{\alpha \cdot n \cdot (\log(3n) + \mu)}$ • $\mathbf{\alpha \cdot \log(n)}$

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

To address this challenge Akavia et. al. [2] propose combining steps (2)-(3) above to a single probabilistic step that returns the required step-function (albeit, with noticeable error probability). They later show how to eliminate the error using few repetitions over multiple rings GF(p) for $p > 2$ together with client post-processing for selecting the correct result.

To avoid the aforementioned post-processing we propose an alternative for the probabilistic test combining steps (2)+(3). First, we make the straightforward observation that instead of testing if the sum of binary indicators is zero (as done in [2]), we can compute the logical-OR of these indicator values. However, this would result in high degree, as the logical-OR over n variables has degree n. Next, to reduce the degree, we employ the method of Razborov and Smolenski [48, 55] for low-degree approximation of the logical-OR function. This method yields a polynomial of degree logarithmic in both n and $1/\varepsilon$, for ε the failure probability.

Elaborating on the above, the Razborov-Smolenski method is applicable in GF(q) for any $q \geq 2$; we apply it with $q = 2$ on all k-th prefix $\big(v[1], \ldots, v[k]\big) \in \{0,1\}^k$ of the aforementioned vector of n binary indicator values. Their low-degree approximation for $\mathrm{OR}\big(v[1], \ldots, v[k]\big) \in \{0,1\}$ is computed as follows. First, for $N(\varepsilon) = \lceil \log_2(n/\varepsilon) \rceil$ uniformly random i.i.d. $r_1, \ldots, r_{N(\varepsilon)} \in \{0,1\}^n$, we compute the parity of the corresponding random subset of entries,

$$\mathsf{p}(r_j) = \sum_{i=1}^{k} r_j[i] \cdot v[i] \mod 2.$$

The parity bit $\mathsf{p}(r_j)$ is always zero when $v = 0^k$ and it is one with probability half when $v \neq 0^k$. Next, we compute the OR of these parity values using the standard degree $N(\varepsilon)$ polynomial for the logical-OR of $N(\varepsilon)$ binary values:

$$\mathrm{OR}\Big(\mathsf{p}\big(r_1\big), \ldots, \mathsf{p}\big(r_{N(\varepsilon)}\big)\Big) = 1 - \prod_{j=1}^{N(\varepsilon)} \big(1 - \mathsf{p}\big(r_j\big)\big) \mod 2.$$

This is equal to $\mathrm{OR}(v[1], \ldots, v[k])$ with probability $1 - \frac{\varepsilon}{n}$.

We note that the Razborov's and Smolenski's [48, 55] approximation method has numerous uses in computer science. In particular, in the context of secure search Barkol and Ishai [3], building on [32, 48, 55], gave a generic transformation from constant-depth unbounded fan-in boolean circuits to low-degree polynomials. They employ their technique for secure multi-party computation of common search functionalities; albeit, in settings of multiple-servers holding *unencrypted data* (cf. single-server holding encrypted data in our work).

1.4 Article Road-map

The rest of this paper is organized as follows. Preliminary definitions and notations in Section 2; Problem statement and threat model in Section 3; Our protocol and the main theorem in Section 4; Instantiations of IsMatch demonstrating performance and functionality enhancements and extensions in Sections 5-6; Conclusions in Section 7. We have moved to the appendix protocols summary (Appendix A), and detailed experimental results (Appendix B).

2 Preliminaries

We state some preliminary notations and definitions.

For natural numbers $k < n$, denote $[n] = \{1, \ldots, n\}$, $[k, n] = \{k, \ldots, n\}$, and $(k, n) = \{k+1, \ldots, n-1\}$. For

array v denote $v[i]$ the i-th element in v. Similarly, for $x \in \{0,1\}^*$, $x[i]$ denotes its i-th bit. We follow the convention of enumerating array entries starting from entry number 1 (not 0), unless stated otherwise. For matrix M the element in row i and column j will be denoted as $M[i,j]$. For a field $\mathbb{F}$, vectors $v, u \in \mathbb{F}^n$ and $k \in [n]$, denote: $\langle v,u\rangle = \sum_{i=1}^{n} v[i]\cdot u[i] \pmod 2$, $\mathsf{prefix}_k(v) = (v_1,\ldots,v_k) \in \mathbb{F}^k$, $\mathsf{suffix}_k(v) = (v_{k+1},\ldots,v_n) \in \mathbb{F}^{n-k}$, and $|v|$ the size (length, dimension) of v $(= n)$.

For $k, n \in \mathbb{N}$, denote by $r_1,\ldots,r_k \leftarrow\!\!\$\, \{0,1\}^n$ the sampling of k arrays independently at random from the uniform distribution over $\{0,1\}^n$. As standard, PPT denotes *probabilistic polynomial time*; and a function $\nu\colon \mathbb{N} \to \mathbb{R}$ is called *negligible* in κ, denoted $\mathsf{negl}(\kappa)$, if for every constant $c > 0$ there exists n_0 such that for all $n > n_0$, $\nu(n) < \kappa^{-c}$.

Definition 2.1 (FHE)**.** *A* leveled homomorphic encryption (FHE) *scheme is defined by a quadruple of* PPT *algorithms* $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *as follows.*

- ***Key generation.*** $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$ *takes a security parameter* κ *and a circuit depth upper-bound* L*, and outputs public key* pk *and secret key* sk*.*
- ***Encryption.*** $[\![b]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(b)$ *takes the public key* pk *and a message* $b \in \{0,1\}$*, and outputs a ciphertext* $[\![b]\!]$*. For* $x \in \{0,1\}^n$*, we denote its bit-by-bit encryption* $[\![x[i]]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x[i])$ *by* $[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$*.*
- ***Decryption.*** $x' \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![x]\!])$ *takes the secret key* sk *and a ciphertext* $[\![x]\!]$*, and outputs a message* $x' \in \{0,1\}^*$*. When* $[\![x]\!]$ *is an array of ciphertexts, decryption is ciphertext-by-ciphertext.*
  Correctness *says that* $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(x)) = x$*.*
- ***Homomorphic evaluation.*** $[\![y]\!] \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, [\![x[1]]\!],\ldots,[\![x[t]]\!])$ *takes* pk*, a function* $f : \{0,1\}^t \to \{0,1\}$ *represented as an arithmetic circuit over* $GF(2)$ *and a set of* t *ciphertexts* $\big([\![x[i]]\!]\big)_{i=1}^{t}$ *outputs a ciphertext* $[\![y]\!]$ *such that* $\mathsf{Dec}_{\mathsf{sk}}([\![y]\!]) = f(x[1],\ldots,x[t])$*. As a shorthand notation we write* $f([\![x]\!])$ *in place of* $\mathsf{Eval}_{\mathsf{pk}}(f, [\![x]\!])$*.*

We will use the standard equality operator, for $a, b \in \{0,1\}^w$, of degree w and $w-1$ overall multiplications:

$$\mathsf{IsEqual}_w(a,b) = \prod_{i\in[w]} \Big(1 + a[i] + b[i]\Big) \qquad \bmod 2 \tag{1}$$

We will also use the standard "greater than" operator $(a > b)$, for $a, b \in \{0,1\}^w$, of degree $w+1$ and $2w$ overall multiplications:

$$\begin{aligned}\mathsf{IsGrt}_w(a,b) = \textstyle\sum_{i\in[w-1]} \Big[\big(a[i]\cdot(b[i]+1)\big)\cdot \\ \mathsf{IsEqual}_{w-i}\big(\mathsf{suffix}_i(a), \mathsf{suffix}_i(b)\big)\Big] + \\ \big(a[w]\cdot(b[w]+1)\big) \qquad \bmod 2\end{aligned} \tag{2}$$

3 Problem Statement

Suppose a client (Alice) wants to use a server (cloud service provider, Bob) for data storage, management and retrieval (Search, Insert, Update, Delete). To protect her privacy Alice uploads only encrypted data to the cloud. She encrypts it using FHE so that Bob has processing capabilities on the data, with single round and low communication protocols that hide Alice's data, queries, returned results and access pattern from Bob.

3.1 Secure-Search on Encrypted Data

In this paper we focus on the problem of setup-free secure-search on FHE encrypted data, following [2]; see Definition 3.1 below. In this problem given an unsorted and encrypted data array $[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$ and encrypted query $[\![q]\!]$ the goal is to find the encrypted index $[\![i^*]\!]$ and element $[\![x[i^*]]\!]$ so that $x[i^*]$ is the first match for query q in array x (formally, $\mathsf{IsMatch}(x[i^*], q) = 1$ and $\forall j < i^* : \mathsf{IsMatch}(x[j], q) = 0$). The predicate IsMatch can be generic (see below).

Definition 3.1 (Secure search)**.** *The server holds an array of encrypted elements (previously encrypted and uploaded by the client to the server, and where the server has no access to the secret decryption key):*

$$[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$$

The original array $x = (x[1],\ldots,x[n])$ *is* ***unsorted*** *and its elements are* ***not necessarily distinct****. The client sends to the server an encrypted query* $[\![q]\!]$*. The server returns the client an encrypted index* $[\![i^*]\!]$ *and element* $[\![x[i^*]]\!]$ *where the index* i^* *is satisfying the condition that it is the index of the first match for query* q *in array* x*:* $i^* = \min\big\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\big\}$

We note that returning a fixed number of matches is inherent in the FHE model. This is because the processing server has to be oblivious of the query, and hence the length of the result has to be the same for every query.

We also would like to point out that above definition does not prevent the client from retrieving multiple matches to a given query. The client can retrieve additional matches by issuing "fetch-next" queries (cf. Section 5.6). This is done without introducing any leakage and without revealing whether "fresh" queries or

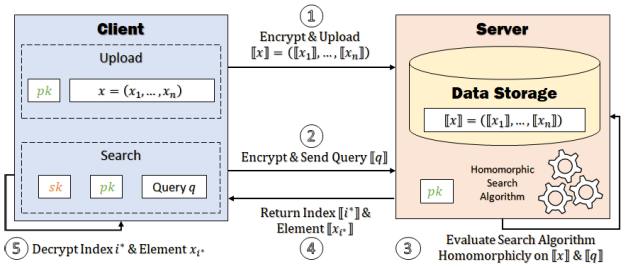


Fig. 1. Depiction of *secure search* on FHE encrypted data

"fetch-next" queries were issued (see also leakage profile discussion in Section 3.3).

Setup-free secure-search protocol employing a solution to the above secure search problem follows (cf. Figure 1):

1. *Keys Generation*: Alice initializes the scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ and generates the keys $(\mathsf{pk}, \mathsf{sk})$. Alice keeps pk, sk and sends pk to Bob.
2. *Array Upload*: Alice gradually, over time, encrypts and uploads elements to Bob. At any given moment, Bob holds an encrypted and unsorted array $[\![x]\!] = \left([\![x[1]]\!], \ldots, [\![x[n]]\!]\right)$.
3. *Secure search:* At any time, Alice may issue a search query q by encrypting it and sending $[\![q]\!]$ to Bob. Bob employs the secure search solution to obtain and send to Alice the encrypted search outcome $([\![i^*]\!], [\![x[i^*]]\!])$ for i^* the index of the first match and $x[i^*]$ the corresponding element. Alice then decrypts to obtain i^* and $x[i^*]$.

Usage could be versatile. For example, the client (Alice) may upload additional data over time with search queries interleaved between uploads; See Section 6.2. Furthermore, Alice could be instantiated by multiple parties with distinct roles: a key generation authority in Step 1, and multiple data-sources and search-clients in Steps 2 and 3 respectively, where the key generation authority sends pk to the data-sources and server, and sends pk, sk to the search-clients.

Generic IsMatch. In order for secure search to be applicable to versatile settings we emphasize that it can be instantiated with various $\mathsf{IsMatch}$ predicates: any predicate that when evaluated on two elements a, b, for a from the space of data elements and b from the space of queries, returns a binary indicator accepting value $\mathsf{IsMatch}(a, b) = 1$ if a, b are considered a match in the context of the given settings (0 otherwise). The server's complexity depends on the complexity of the $\mathsf{IsMatch}$ predicate.

3.2 Desired Properties

We define properties desired from a secure search protocol (following [2]); see Tables 1-2:

1. ***Full security:*** *two (equal size) adversarially-chosen queries and data arrays are computationally-indistinguishable from the search and upload protocols; see formal statement in Definition 3.2.*
2. ***Efficient client:*** *client's running-time is polynomial in the time to encrypt the query q and decrypt the search outcome ciphertexts $([\![i^*]\!], [\![x[i^*]]\!])$.*
3. ***Efficient server:*** *the server evaluates polynomials $f([\![x]\!], [\![q]\!])$ of degree polynomial in $\log n$ and the degree of $\mathsf{IsMatch}$, and of size (i.e. the overall number of multiplication and addition operations) polynomial in n and the size of $\mathsf{IsMatch}$.*
4. ***Efficient communication:*** *the protocol has single-round protocol and communication bandwidth polynomial in $|q|$, $|i^*| = \log n$ and $|x[i^*]|$ (for $|z|$ denoting the binary representation length of z).*
5. ***Setup-free:*** *data elements are maintained in an unsorted array where each new element is inserted at the end of the array.*[2]
6. ***Unrestricted search functionality:*** *no restrictions are placed on the number of array elements that match the query.*
7. ***Retrieval of both index and element:*** *client's output consists of both index and element $(i^*, x[i^*])$.*
8. ***Post-processing free:*** *the server sends the encrypted search outcome $([\![i^*]\!], [\![x[i^*]]\!])$ for the client to decrypt, with no client's post-processing needed.*
9. ***Negligible error probability:*** *with overwhelming probability the client's output $(i^*, x[i^*])$ is the correct search outcome, i.e., $i^* = \min\{\, i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$.*
10. ***Compatibility with all FHE schemes:*** *the protocol can employ (as a black-box) any FHE scheme.*

3.3 Threat Model

The untrusted party in our scenario is the *honest-but-curious* (also called, semi-honest) and *computationally-bounded* adversary controlling the server (as in the case of hacked cloud servers). Semi-honest means, as standard, that the adversary follow the protocol, but may try to learn sensitive information. Namely, for the up-

2 Examples to techniques that require setup include: preprocessing of the entire plaintext data, as in sorting or indexing; maintenance of additional search-oriented data structures such as search-trees or hash tables; in general, the use of any additional tools for the purpose of achieving sub-linear search time.

load functionality the server provides the storage facility and is prohibited from modifying or destroying the encrypted array ($[\![x]\!]$). Likewise, for the search functionality the server receives encrypted queries ($[\![q]\!]$) and is obligated to follow the protocol and return encrypted search outcomes accordingly ($[\![i^*]\!]$, $[\![x[i^*]]\!]$). On the other hand, the adversary can try to derive sensitive information from the stored elements, received queries, data access patterns and search outcomes. Computationally-bounded means, as standard, that the adversary's actions are captured by a probabilistic polynomial time (PPT) algorithm.

We mention that there is no need to consider adversaries controlling the client as it can trivially simulate the entire protocols (Upload, Search) by herself. This is because the server's role is not to provide input or receive output, but rather to take the bulk of computational burden off the client.

Our security requirement is that if the client issues the protocol with one of two adversarially-chosen equal size queries $q^{(0)}$, $q^{(1)}$ (similarly, arrays $x^{(0)}$, $x^{(1)}$), the adversary controlling the server cannot distinguish between them; see the formal attack games below.

Definition 3.2 (Full security)**.** *We say that an upload and search protocol provides* full security *if every* PPT *semi-honest adversary $\mathcal{A}$ controlling the server has no more than a negligible advantage* $\mathbf{Adv}(\mathcal{A}, \mathcal{FHE}, \kappa) = \mathsf{negl}(\kappa)$ *in winning the attack games on query or data (as specified below).*

Attack on query (respectively, data). The attack games involve the adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, both given the FHE scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ and the security parameter κ, proceeding as follows.

1. $\mathcal{C}$ executes the key generation step (Step I, Figure 3) to obtain $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\kappa)$, and sends pk to $\mathcal{A}$.
2. $\mathcal{A}$ chooses parameters n, w, w' for array size, elements size, and query size, respectively; generates and sends to $\mathcal{C}$ the tuple $(x, q^{(0)}, q^{(1)})$ for x an array of n elements of size w, and $q^{(0)}, q^{(1)}$ queries of size w' (respectively, the tuple $(x^{(0)}, x^{(1)}, q)$ for arrays $x^{(0)}, x^{(1)}$ and query q of sizes as specified above).
3. $\mathcal{C}$ samples $b \leftarrow\!\!\$\ \{0,1\}$ uniformly at random.
4. $\mathcal{C}$ and $\mathcal{A}$ execute the upload and search protocols (Step II-III, Figure 3) playing the roles of client and server respectively. The client's input is x and $q^{(b)}$ (respectively, $x^{(b)}$ and q); the server has no input.
5. $\mathcal{A}$ sends $b' \in \{0,1\}$ to $\mathcal{C}$, and wins if $b' = b$.

The advantage of $\mathcal{A}$ in the search attack on query (respectively, data) is defined to be

$$\mathbf{Adv}(\mathcal{A}, \mathcal{FHE}, \kappa) = \left|\Pr\left[b' = b\right] - 1/2\right|$$

We remark that, since $\mathcal{A}$ holds pk, he can simulate on its own the upload step for additional data entries x' of its choice. Likewise, since $\mathcal{A}$ holds $x, q^{(0)}, q^{(1)}$ (respectively, $x^{(0)}, x^{(1)}, q$), he can simulate on its own the search step – excluding the client's final decryption step– for whatever and as many queries q' as $\mathcal{A}$ wishes, including queries $q^{(0)}, q^{(1)}$ (respectively, q). These upload and search steps can occur both before and after the challenge.

Full security leakage profile discussion. Full security implies that the adversary participating in the protocol does not learn new information on *data, queries, and search outcomes*, other than the following size information: (1) plaintext space; (2) array size upper-bound; (3) element size upper-bound; (4) overall count of executed queries. This holds both for data-at-rest (upload) and data-in-use (search).

In particular, the protocol hides *access-patterns* to prevent, for example, identifying frequently searched data elements; and hides *search-patterns* to prevent inferring from search outcomes whether two searches use related query values. The overall count of executed queries does not reveal any information regarding the content or distribution of stored elements; and the server is unable to distinguish between "fresh" queries and "fetch-next" queries (cf. Section 5.6).

4 Secure Search

We specify our secure search protocol (see Figures 3-4, Section A) that we name: *Binary Raffle Protocol*.

This section is organized as follows. The simple keys generation and data upload steps are in Sections 4.1–4.2; the secure search step, which is the heart of this work, in Section 4.3; and our main theorem in Section 4.4.

4.1 Keys Generation Step

In the keys generation step the client executes the key generation algorithm of the leveled scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ (see Definition 2). The input to the KGen algorithm are the security parameter κ and the level $L = \log_2(d)$ for d the degree of the secure search polynomial (see Section 4.3 below).

In details, the level L depends on the following upper-bounds: (1) error probability ε; (2) array size n; (3) degree of the desired matching polynomial $d_{\texttt{IsMatch}}$. Specifically it needs to be set to $L = \lceil \log\log(n/\varepsilon) + \log(d_{\texttt{IsMatch}}) \rceil$.

The output is $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$ where $(\mathsf{pk},\mathsf{sk})$ are kept by the client, and pk is sent to the server.

4.2 Data Upload Step

In the array upload step the client encrypts its data array $x = \big(x[1],\ldots,x[n]\big)$ and sends the ciphertexts $[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$ to the server. The encryption is performed element by element. Each element $x[i]$ is given by its binary representation and contains up to w bits. Prior to encryption, to avoid revealing the number of bits in each element the client pads all elements with leading zeros until all of them are of length w. To pad an element without "losing" its leading zeros one can slightly modify the aforementioned padding by adding a single 1 bit next to the MSB position of the element and afterwards add 0 bits up to the desired size. The encryption of each element is then performed bit by bit.

We would like to emphasize that we do not conciser the padding procedure described above as setup. This is due to padding being performed locally element-by-element with only constant additional memory, requiring no processing of the entire plaintext data, and with no reduction in the search time.

4.3 Secure Search Step

We specify the secure search step, which is the heart of our protocol (Figure 3, Step III).

The starting point of the secure search protocol is after the client obtained sk,pk, the server obtained pk and the encrypted array $[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$ has been uploaded to the server (see Sections 4.1-4.2).

The secure search step (Figure 3, Step III) proceeds as follows. First the client encrypts her search query $[\![q]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(q)$ and sends it to the server (Step III.(a)). Next, the server evaluates the steps specified below on the stored array $[\![x]\!]$ and the query $[\![q]\!]$ to obtain and send to the client $([\![b^*]\!],[\![x[i^*]]\!])$ for $b^* \in \{0,1\}^{\lceil \log_2 n\rceil+1}$ the binary representation of the index of the first match $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i],q) = 1\}$ (Step III.(b)). Finally, the client decrypts to obtain the desired output $(b^*, x[i^*])$ (Step III.(c)).

We elaborate below on the server's computations (Step III.(b)). We start by specifying how to return a selector array $s' \in \{0,1\}^n$ accepting value 1 on entry i^* and 0 otherwise (Section 4.3.1), then elaborate on our key algorithm for achieving the former (Section 4.3.2), and finally specify the additional actions for returning the ciphertext for index and element $(b^*, x[i^*])$ (Section 4.3.3).

4.3.1 Binary Raffle for Computing Selector Array s'

We specify how the server computes (the encryption of) a selector array $s' \in \{0,1\}^n$ accepting value 1 on entry i^* and 0 otherwise (Figure 3, Step III.(b).(1)–(3)).

First (Step III.(b).(1)), the server evaluates the specified pattern matching polynomial $\mathsf{IsMatch}$ on each entry of the stored array $[\![x]\!]$ and the lookup value $[\![q]\!]$. This results in an encrypted array $[\![ind]\!]$ that contains in every index $i \in [n]$ the encrypted boolean result $\mathsf{IsMatch}([\![x[i]]\!],[\![q]\!]) \in \{[\![0]\!],[\![1]\!]\}$:

$$[\![ind]\!] \leftarrow \big(\mathsf{IsMatch}([\![x[1]]\!],[\![q]\!]),\ldots,\mathsf{IsMatch}([\![x[n]]\!],[\![q]\!])\big)$$

Next (Step III.(b).(2)), the heart of the protocol is converting $[\![ind]\!]$ to a step function array of size n

$$[\![s]\!] = \big([\![0]\!],\ldots,[\![0]\!],[\![1]\!],\ldots,[\![1]\!]\big)$$

that contains $[\![0]\!]$ in every index before i^* and $[\![1]\!]$ from index i^* and further on. This $[\![s]\!]$ is computed using our randomized algorithm detailed in Section 4.3.2:

$$[\![s]\!] \leftarrow \mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}([\![ind]\!])$$

With probability $1-\varepsilon$, the result $[\![s]\!]$ of our randomized algorithm will be the encryption of the step function described above.

Third (Step III.(b).(3)), we compute the pairwise difference of adjacent indices in $[\![s]\!]$ (i.e. its derivative):

$$\forall i \in [2,n]: \quad [\![s'[i]]\!] \leftarrow [\![s[i]]\!] - [\![s[i-1]]\!] \pmod 2 \quad \text{and}$$

$$[\![s'[1]]\!] \leftarrow [\![s[1]]\!], \quad [\![s'[n+1]]\!] \leftarrow [\![1]\!] - [\![s[n]]\!]$$

The resulting array $[\![s']\!]$ will contain $[\![0]\!]$ in every index except in the index of the first match i^* (or at index $n+1$ if no match exists) where it will be $[\![1]\!]$.

4.3.2 $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ Algorithm

We next describe the $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm, which is the heart of our secure search protocol (Figure 3, Step III.(b).(2)).

The $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm transforms any array $v \in \{0,1\}^n$ of binary values into an array $t = (0,\ldots,0,1,\ldots,1) \in \{0,1\}^n$ that contains the step function with value 1 starting from the first index i where $v[i] = 1$. This algorithm is a randomized Monte Carlo algorithm with failure probability ε (see Figure 4). In addition we provide an illustration for the main steps of the algorithm in Figure 2.

For clarity of presentation we present the algorithm as preforming computations on plaintext values. Modifying the algorithm to apply it on FHE encrypted data is straightforward: simply replace each addition/multiplication operation with its homomorphic counterpart.

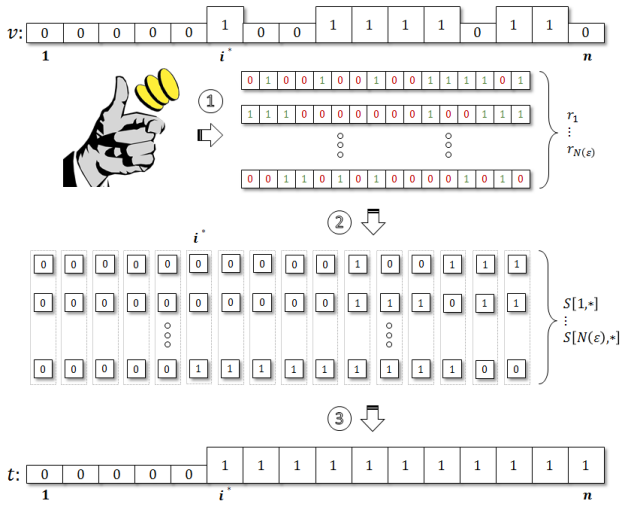


Fig. 2. An illustration of the main steps in $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm: On input array v the following steps are performed: (1) Sample $N(\varepsilon)$ random binary arrays $r_1,\ldots,r_{N(\varepsilon)}$ of length n; (2) Matrix S will hold $N(\varepsilon)\cdot n$ random partial prefix sums as follows $\forall j \in [N(\varepsilon)], \forall k \in [n]$: $S[j,k] = \langle \mathsf{prefix}_k(v), \mathsf{prefix}_k(r_j)\rangle$; (3) Each index $k \in [n]$ in t will hold the logical OR between the elements in column k of S. That is, calculate $t[k] = 1 - \left(\prod_{j=1}^{N(\varepsilon)} \left(1 - S[j,k]\right)\right) \pmod 2$. With probability $1-\varepsilon$, the output is array t that contains a step function with 0s before index i^* and 1s from index i^* and onwards.

Random Partial Prefix Sums To determine whether a k-prefix vector $\mathsf{prefix}_k(v) = \big(v[1],\ldots,v[k]\big) \in \{0,1\}^k$ of the binary indicator vector $v = \big(v[1],\ldots,v[n]\big) \in \{0,1\}^n$ is non-zero (i.e. not $0^k = (0,\ldots,0)$ of size k), we do the following. We compute the parity of a random subset for the entries of $\mathsf{prefix}_k(v)$, that is $\mathsf{parity}(r) = \sum_{i=1}^{k} r[i]\cdot v[i]$ for uniformly random $r \in \{0,1\}^n$. The parity bit $\mathsf{parity}(r)$ is always zero when $v = 0^k$ and it is one with probability half when $v \neq 0^k$. By repeating for $N(\varepsilon)$ i.i.d. random variables $r_1,\ldots,r_{N(\varepsilon)} \in \{0,1\}^n$ (for sufficiently large $N(\varepsilon)$) and computing the OR of the resulting bits $\mathsf{parity}\big(r_1\big),\ldots,\mathsf{parity}\big(r_{N(\varepsilon)}\big)$, i.e. computing: $t[k] = \mathrm{OR}\Big(\mathsf{parity}\big(r_1\big),\ldots,\mathsf{parity}\big(r_{N(\varepsilon)}\big)\Big)$ we obtain the desired step-function $t = \big(t[1],\ldots,t[n]\big) \in \{0,1\}^n$ with overwhelming probability.

4.3.3 Returning Index and Element

Finally we specify the additional server's steps for returning the ciphertext for index and element $(b^*, x[i^*])$ that are sent then to the client (Figure 3, Step III.(b).(4)-(5)).

Computing $[\![b^*]\!]$. The server computes $[\![b^*]\!] = B \cdot [\![s']\!]$ for $B \in \{0,1\}^{(\lceil \log_2 n\rceil+1)\times n}$ the matrix that contains in each column $k \in [n]$ the binary representation of k. The resulting array $[\![b^*]\!]$ will hold the binary representation of the index i^* of the single $[\![1]\!]$ in $[\![s']\!]$ (or 0 if the array contains only $[\![0]\!]$'s). This is because multiplying the matrix B by any array of size $n+1$ that contains a single 1 bit in some index $j \in [n]$ results in a array of size $\lceil \log_2 n\rceil + 1$ that holds the binary representation of j (and a array of zeros if $j = n+1$).

Computing $[\![x[i^*]]\!]$. Since the problem of privately retrieving a uniquely identifiable element from an encrypted array has efficient FHE based solutions, we first focused above (Sections 3-4.3.2) on the task of computing and returning the encrypted index alone. We next explain how to retrieve also the corresponding element.

The most straightforward way to retrieve the element $x[i^*]$ in addition to i^* is to utilize a *Private Information Retrieval* (PIR) protocol (see Section 1.1) on the encrypted array $[\![x]\!]$ and index $[\![i^*]\!]$. This would require no further interaction (as the server already had $[\![i^*]\!]$); However it would increase the degree of our secure search protocol by a factor of $d_{\mathrm{PIR}} = \log n$.

Instead we suggest a more efficient alternative for retrieving the matched element $[\![x[i^*]]\!]$. This is by reusing the array $[\![s']\!]$ that already contains $[\![0]\!]$'s in all indices except in the index of the first match i^*, where it contains $[\![1]\!]$. The additional step would be to calculate for each index $j \in [n]$ and each bit $k \in [w]$: $[\![x[i^*][k]]\!] = \sum_{j=1}^{n}([\![x[j][k]]\!]\cdot[\![s'[j]]\!])$. This method would increase the degree of our secure search protocol only by 1 since $[\![x]\!] = \big([\![x[1]]\!],\ldots,[\![x[n]]\!]\big)$ are freshly encrypted ciphertexts.

4.4 Main Theorem

The advantages and properties of our secure search protocol are given in the following theorem.

Theorem 4.1. *There exists a secure search protocol attaining desired properties 1-10 as specified in Section 3.2.*

In particular, the protocol of Figure 3, when executed on shared parameters $(\mathsf{IsMatch}, \mathcal{FHE}, \kappa, \varepsilon, n, w)$ *and client's input data array* $x = (x[1],\ldots,x[n])$ *for* $x[i] \in \{0,1\}^w$ *and query* q*, satisfies the following:*

1. Correctness: *With probability* $1-\varepsilon$*, the client's output is* $(b^*, x[i^*])$ *for*

$$i^* = \min\{\, i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$$

and $b^* \in \{0,1\}^{\lceil \log_2 n\rceil+1}$ *the binary representation of* i^**. The server has no output.*

2. Complexity of the search step (Step III, Figure 3): *The client's running-time is the time compute* $|q|$ *encryptions and* $|b^*|+|x[i^*]|$ *decryptions. The server*

evaluates a polynomial of degree $\log(n/\varepsilon) \cdot d$ *and overall multiplications* $n \cdot (\log(n/\varepsilon) + \mu + |x[i^*]|)$ *for* d *and* μ *the degree and overall multiplications of* IsMatch. *The communication is 1-round, consisting of* $|q|$ *ciphertexts sent from client and* $|b^*| + |x[i^*]|$ *ciphertexts from server. See Table 3.*

3. Security: *The protocol attains full security (see Definition 3.2), assuming semantic security of* $\mathcal{FHE}$.

Proof. Correctness follows from the correctness of $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm, which holds with probability $1-\varepsilon$. Complexity analysis follows by inspection. See the security proof below. Showing that properties 1-10 holds easily follows.

Suppose there exists a PPT algorithm $\mathcal{A}$ for the query attack game (Section 3.3). (The case of the data attack game is analogous; details omitted.) We construct a PPT algorithm $\mathcal{A}'$ for the (single message) IND-CPA game for $\mathcal{FHE}$ (see Definition 11.2 in [41]), and show $\mathcal{A}'$ has the same advantage as $\mathcal{A}$. By semantic security of $\mathcal{FHE}$ we conclude that no PPT adversary $\mathcal{A}$ has non-negligible advantage in the former.

The adversary $\mathcal{A}'$ plays the role of the adversary in the IND-CPA game for $\mathcal{FHE}$, and of the challenger in the query attack game: (1) Upon receiving $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{KGen}(1^\kappa)$ from the challenger in the IND-CPA game for $\mathcal{FHE}$ ($\mathcal{C}_{\mathcal{FHE}}$), $\mathcal{A}'$ invokes the query attack game with $\mathcal{A}'$ playing the challenger's role and $\mathcal{A}$ the adversary, by sending $(\mathsf{pk}, \mathsf{sk})$ to $\mathcal{A}$. (2) Upon receiving from $\mathcal{A}$ the array x and two queries $q^{(0)}, q^{(1)}$, $\mathcal{A}'$ sends the two queries to $\mathcal{C}_{\mathcal{FHE}}$. (3) Upon receiving from $\mathcal{C}_{\mathcal{FHE}}$ the challenge ciphertext $c = [\![q^{(b)}]\!]$, $\mathcal{A}'$ encrypts (element-by-element and bit-by-bit) to produce the ciphertexts array $[\![x]\!]$, and sends $[\![x]\!], c$ to $\mathcal{A}$. (4) Upon receiving a guess b' from $\mathcal{A}$, $\mathcal{A}'$ sends b' to $\mathcal{C}_{\mathcal{FHE}}$. Clearly, $\mathcal{A}'$ perfectly simulates the challenger in the query attack game and wins the (single message) IND-CPA game for $\mathcal{FHE}$ if-and-only-if $\mathcal{A}$ wins the query attack game. Thus, they have the same advantage. □

5 Make IsMatch Great Again!

To demonstrate the strength of our compatibility with generic matching criteria we present versatile matching criteria that can be integrated into our protocol, exhibiting advantageous properties for both performance (Section 5.1) and functionality (Sections 5.2-5.6). We stress that security holds for all examples to follow, due to the server obliviously performing homomorphically operations on ciphertexts; See Theorem 4.1, Section 4.4.

5.1 Faster Exact Match via Hashing

To speedup performance in exact match search we propose applying Universal Hashing [9, 38] to reduce the degree and overall multiplications for computing IsMatch from $\mathcal{O}(w)$ to $\mathcal{O}(\log(n/\varepsilon))$, where w is the elements' size, n the number of elements, and ε the probability of error. This is particularly appealing in use-cases with large elements, specifically, when $w > 2\log(2n/\varepsilon)$. See Table 3, row (iv).

In details, we propose that the server first chooses a uniformly random Toeplitz Matrix $A \in \{0,1\}^{v \times w}$ and a random vector $b \in \{0,1\}^v$ for $v = 2\log(2n/\varepsilon)$ to specify a hash function $h_A(x) = Ax + b \mod 2$ mapping $\{0,1\}^w$ to $\{0,1\}^v$. The server then homomorphically applies on encrypted values the following equality operator on hashed values:

$$\mathsf{IsMatch}(x[i], q) := \mathsf{IsEqual}_v\left(h_A(x[i]), h_A(q)\right)$$

for $\mathsf{IsEqual}_v$ as specified in Section 2.

To analyze the complexity of the above IsMatch polynomial note that this hashing requires solely homomorphic additions operations. So the degree and overall multiplications is $d = v$ and $\mu = v - 1$ respectively. The failure probability due to collision is $\varepsilon/2$.

Note that the above optimization does not contradict the setup-freeness of the whole solution, because the hash function is applied by the server on x while x is already encrypted.

5.2 Boolean Logic Queries

The matching criterion in our protocol can express any Boolean logic, such as conjunction, disjunction, negation or their combination. This logic can be applied, for example, on elements' sub-fields (e.g. first and last names in personal records) or characters.

The expression of Boolean logic as a polynomial over GF(2) is via the standard arithmetization techniques: expressing negation by $\mathsf{not}(a) = 1 - a$, conjunction by $\mathsf{conj}(a_1, \ldots, a_t) = \prod_{j=1}^{t} a_j$ and disjunction by $\mathsf{disj}(a, b) = a \oplus b \oplus a \cdot b$.

The degree d (respectively, overall multiplications μ) is the maximum composition length (respectively, overall number) of disjunction and conjunction operations. See concrete examples in the following sections.

5.3 Wild-Card Queries

Wild-card queries are specified by $q \in \{0, 1, *\}^w$. "Wild-card positions" are the entries j where q accepts $*$. Wild-card match returns is true when $x[i]$ and q agree on all the non wild-card positions.

In case the client is willing to leak the wild-card positions, we simply apply the equality test (see Section 2) on the substrings of $x[i]$ and q corresponding to the non wild-card positions. Complexity is only improved by this (compared to exact match on entire strings).

In case the client wishes to hide the wild-card positions, she can augment the query with the (encrypted) indicator vector $I \in \{0,1\}^w$ accepting 1 on all non wild-card positions, and 0 otherwise. The matching polynomial (to be homomorphically evaluated on encrypted values) is:

$$\mathsf{IsMatch}'\left(x[i],(q,I)\right) = \mathsf{IsMatch}\left(x[i]\cdot I, q\cdot I\right)$$

for $\cdot$ the entry-wise product.

Correctness follows as on wild-card entries both $x[i]$ and q are turned to 0 to guarantee equality, and they keep their original values on the non wild-card position.

Complexity overhead for the client is $|w|$ additional encryptions. The server evaluates a polynomial with degree increased by 1, and overall number of multiplications increased by an additive term of $2w$. Saving a factor of w in the overall multiplication is easy: by replacing each $*$ value in q with 0, and compute $\mathsf{IsMatch}\left(x[i]\cdot I, q\right)$.

5.4 Range Queries

Range queries specify lower and upper boundaries $(l,u$ respectively) to retrieve elements in the range (l,u). I.e., retrieving $(i,x[i])$ for $i = \min\{\, i\in[n] \mid l < x[i] < u\}$. Boundaries and data are encrypted by the client.

Range queries are easily implemented as the conjunction of two boundary-tests. Specifically, for elements $x[i]$ in $\{0,1\}^w$, the matching polynomial (to be homomorphically evaluated on encrypted values) is:

$$\mathsf{IsMatch}\left(x[i],(l,u)\right) = \mathsf{conj}\left(\mathsf{IsGrt}_w\left(x[i],l\right),\mathsf{IsGrt}_w\left(u,x[i]\right)\right)$$

for IsGrt_w and conj operators as in Sections 2 and 5.2.

The complexity of the client is dominated by encrypting $|l|+|u| = 2w$ bits for specifying the query, and by decrypting the outcome. The server's complexity grows with the matching degree and overall multiplications: $d = 2(w+1)$ and $\mu = 4w+1$, respectively.

5.5 Search In Sub-Array

In sub-array search for $\mathsf{IsMatch}$, the client specifies boundaries $l,u\in[n]$ together with the query q in order to retrieve from the server the first match for q in the sub-array $(x[l+1],\ldots,x[u-1])$. I.e. retrieving $(i,x[i])$ for $i = \min\{\, j\in(l,u) \mid \mathsf{IsMatch}(x[j],q)=1\}$. Boundaries, query and data are all encrypted by the client.

A sub-array search is easily computed as the conjunction of three requirements: $\mathsf{IsMatch}(x[i],q) = 1$, $i > l$, and $u > i$. Specifically, for indices specified by length $m = \log n$ binary representation, the matching polynomial (to be homomorphically evaluated on encrypted values) is:

$$\mathsf{IsMatch\_InSubArray}_n\left(x[i],(q,l,u)\right) =$$
$$\mathsf{conj}\left(\mathsf{IsMatch}\left(x[i],q\right),\mathsf{IsGrt}_m\left(i,l\right),\mathsf{IsGrt}_m\left(u,i\right)\right)$$

for IsGrt_w and conj operators as in Sections 2 and 5.2.

The complexity overhead compared to the underlying $\mathsf{IsMatch}$ (cf. Table 3) is as follows. The client's overhead is encrypting $|l|+|u| = 2\log(n)$ additional bits. The server evaluates a matching polynomial with degree and overall multiplications $d' = d+2(m+1)$ and $\mu' = \mu+4m$ respectively, for d,μ the degree and overall multiplications for the underlying matching criterion $\mathsf{IsMatch}$. Namely, an additive overhead of $\mathcal{O}(\log n)$.

We note that to search on a suffix $[x[l+1],\ldots,x[n]]$ of the array it suffices for the client to specify only the lower boundary l and for the server to compute the conjunction of only two conditions: $\mathsf{IsMatch}(x[i],q)=1$ and $i > l$. Analogously, for prefix search. This reduces the complexity overhead by a factor of 2.

5.6 Sequential Retrieval ("Fetch-Next")

We extend our secure search functionality to return, not only the first match, but also the next matching element (Fetch-Next), with further interaction. I.e. given a match $(i,x[i])$ retrieving the next match $(i',x[i'])$ for $i' = \min\{\, j\in[i+1,n] \mid \mathsf{IsMatch}(x[j],q)=1\}$. For this purpose we initiate the protocol with an augmented matching criteria that enables searching in an array suffix $[x[l+1],\ldots,x[n]]$ for boundary l specified by the client; see Section 5.5. Boundary, query and data are all encrypted by the client.

To issue a Fetch-Next query for q, after the client has already retrieved a match $(i,x[i])$, the client simply sets l to be i. This causes the search to be performed on indices $[i+1,\ldots,n]$ (without revealing the sub-array to the server). This routine can go on until the client receives the response that indicates that there are no more matches (possibly padding with dummy queries). To issue a "fresh" query, the client will simply set $l=0$.

The server cannot distinguish between a "fresh" query and a "fetch-next" query as in both cases the index l and query q are encrypted. The amount of elements that match a query q, out of the total number of queries, is not leaked to the server.

6 Extensions

We overview extensions to our secure search protocol.

6.1 Client-Side Amplification

To reduce the server's complexity load due to degree's growth in inverse-error $1/\varepsilon$, we can employ standard client-side amplification; See [53] Lemma 10.5. Specifically, by setting the protocols error parameter to $0 < \varepsilon_0 < 1/2$, repeating the protocol in parallel $\frac{-\log_2(1/\varepsilon)}{\log_2(4\varepsilon_0(1-\varepsilon_0))}$ time, and letting the client select the most frequent result, we get error ε but degree growth only with ε_0; See Table 3, row (v).

6.2 Dynamic Data Management

The client can have the benefits of dynamic data management: Insert, Update and Delete (see below). She can also execute search and the above commands multiple times and in any order that she wants.

Insert: Insertion of additional elements requires the server to append another ciphertext to the end of the encrypted array (here, and throughout this work, we assume that the size of the array n is known to the server, see Section 3.3).

Update: To update a specific element $x[i]$ the client first retrieves its index i using our *Secure Search*. Afterwards, to change the value of $x[i]$, denoted old, to a different value new the client submits the following tuple the server: $\big(\text{UPDATE}, [\![i]\!], [\![diff]\!] = [\![new - old]\!]\big)$. The server now homomorphically adds to each elements $[\![x[j]]\!]$ of the stored array the value $\mathsf{IsEqual}([\![i]\!], j) \cdot [\![diff]\!]$. This results in a new encrypted array $[\![x']\!]$ satisfying $x'[i] = new$ and $\forall j \neq i,\ x'[j] = x[j]$.

Delete: Deletion of elements can be achieved by updating them to a reserved "Deleted" symbol. Another option is switching the value of the element we wish to delete to that of the last element in the array and reducing the number of elements n by 1 (for cases when the dynamic value of n is either maintained by the client, or is not a secret and can be kept with the server).

7 Conclusions

In this work we presented a new and improved solution for secure search on FHE encrypted data. Our solution improves over the prior state-of-the-art of setup-free searching on FHE encrypted data in being: (1) post-processing free and with negligible error probability, (2) faster for both client and server, and (3) compatible with all FHE candidates. We implemented our secure search protocol and performed extensive benchmarks showing concrete run-time speedup by an order of magnitude.

Acknowledgment

This work was supported in part by the Center for Cyber Law & Policy at the University of Haifa, and by the BIU Center for Research in Applied Cryptography and Cyber Security. Both in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

References

[1] Mohamed Ahmed Abdelraheem, Tobias Andersson, and Christian Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. *IACR Cryptology ePrint Archive*, 2017:24, 2017.

[2] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure search via multi-ring sketch for fully homomorphic encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 985–1001. ACM, 2018.

[3] Omer Barkol and Yuval Ishai. Secure computation of constant-depth circuits with applications to database search problems. In *Annual International Cryptology Conference*, pages 395–411. Springer, 2005.

[4] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *International conference on the theory and applications of cryptographic techniques*, pages 506–522. Springer, 2004.

[5] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J Wu. Private database queries using somewhat homomorphic encryption. In *International Conference on Applied Cryptography and Network Security*, pages 102–118. Springer, 2013.

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

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[8] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106. IEEE Computer Society, 2011.

[9] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112. ACM, 1977.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679. ACM, 2015.

[11] Gizem S Çetin, Wei Dai, Yarkin Doröz, William J Martin, and Berk Sunar. Blind web search: How far are we from a privacy preserving search engine? *IACR Cryptology ePrint Archive*, 2016:801, 2016.

[12] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255. ACM, 2017.

[13] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *IEEE Transactions on Information Forensics and Security*, 11(1):188–199, 2016.

[14] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[15] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords*. Citeseer, 1997.

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

[17] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[18] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient pir from ntru. In *International Conference on Financial Cryptography and Data Security*, pages 195–207. Springer, 2014.

[19] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[20] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[21] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: Efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual Cryptology Conference*, pages 563–592. Springer, 2016.

[22] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009.

[23] Craig 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, 2009.

[24] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[25] Matthieu Giraud, Alexandre Anzala-Yamajako, Olivier Bernard, and Pascal Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *14th International Conference on Security and Cryptography SECRYPT 2017*. SCITEPRESS-Science and Technology Publications, 2017.

[26] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[27] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[28] Torbjrn Granlund et al. *GNU MP 6.1.2 Multiple precision arithmetic library*. Samurai Media Limited, 2016.

[29] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1353–1364. ACM, 2016.

[30] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 655–672. IEEE, 2017.

[31] S Halevi and V Shoup. The helib library, 2015.

[32] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *focs*, page 294. IEEE, 2000.

[33] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Ndss*, volume 20, page 12, 2012.

[34] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *Annual International Cryptology Conference*, pages 339–370. Springer, 2018.

[35] Myungsun Kim, Hyung Tae Lee, San Ling, Shu Qin Ren, Benjamin Hong Meng Tan, and Huaxiong Wang. Better security for queries on encrypted databases. *IACR Cryptology ePrint Archive*, 2016:470, 2016.

[36] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. Private compound wildcard queries using fully homomorphic encryption. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[37] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017(4):177–197, 2017.

[38] Hugo Krawczyk. Lfsr-based hashing and authentication. In *Annual International Cryptology Conference*, pages 129–139. Springer, 1994.

[39] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373. IEEE, 1997.

[40] Kristin Lauter, Adriana López-Alt, and Michael Naehrig. Private computation on encrypted genomic data. In *International Conference on Cryptology and Information Security in Latin America*, pages 3–27. Springer, 2014.

[41] Yehuda Lindell and Jonathan Katz. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[42] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[43] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[44] Rafail Ostrovsky and William E Skeith. A survey of single-database private information retrieval: Techniques and applications. In *International Workshop on Public Key Cryptography*, pages 393–411. Springer, 2007.

[45] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, volume 15, pages 515–530, 2015.

[46] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *USENIX Security Symposium*, volume 14, pages 797–812, 2014.

[47] David Pouliot and Charles V Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1341–1352. ACM, 2016.

[48] Alexander A Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[49] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[50] Sujoy Sinha Roy, Frederik Vercauteren, Jo Vliegen, and Ingrid Verbauwhede. Hardware assisted fully homomorphic function evaluation and encrypted search. *IEEE Transactions on Computers*, 66(9):1562–1572, 2017.

[51] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((logn) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214. Springer, 2011.

[52] Victor Shoup. Ntl: A library for doing number theory, 10.5.0. *http://www. shoup. net/ntl/*, 2017.

[53] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[54] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[55] Roman Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 77–82. ACM, 1987.

[56] Dawn Xiaoding Song, David Wagner, and Adrian 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.

[57] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[58] Haixu Tang, Xiaoqian Jiang, Xiaofeng Wang, Shuang Wang, Heidi Sofia, Dov Fox, Kristin Lauter, Bradley Malin, Amalio Telenti, Li Xiong, et al. Protecting genomic data analytics in the cloud: state of the art and opportunities. *BMC medical genomics*, 9(1):63, 2016.

[59] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313, 2017.

[60] David P Woodruff et al. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

[61] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

[62] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[63] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 65–76. ACM, 2013.

[64] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, pages 707–720, 2016.

A Protocols' Summary

See Figure 3 and Figure 4.

B Experimental Results

In this section we describe in detail the benchmarks performed to evaluate our *Binary Raffle* protocol and discuss our results. As a reference point, we executed benchmarks on an implementation of the *SPiRiT* protocol [2], the state-of-the-art secure search solution directly related to our FHE based protocol. We evaluated both the deterministic and randomized variants of *SPiRiT*.

We first present benchmarks with the same matching criteria predicate IsMatch as in [2]: IsEqual (see Equation 1, Section 2). Additionally, is Section B.3.5, we give selected benchmarks results for several other matching criteria (see more details in Section 5).

B.1 Experimental Setup

We executed the protocols on top of the HElib C++ library [31] that was compiled with NTL [52] running over GMP [28]. We utilized a single Ubuntu Server 16.04.4 LTS Linux machine with Intel Xeon E7-4870 CPU running at 2.40GHz on 16 cores, 30MB Cache and 16GB RAM.

Parallelization and SIMD In all experiments we utilized all available CPU cores by dividing the input array into equally sized segments that were processed by each core. After completing its execution, every core returned the first matched index candidate for its array segment.

Parameters (Shared Input): • Description of matching condition polynomial $\mathsf{IsMatch}\big(x, y\big) \in \{0,1\}$ with upper bound on the degree d. • Scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$. • Security parameter κ. • Error probability ε (upper-bound). • Array size n (upper-bound). • Element bit length w (upper-bound).
Inputs: The client's inputs are: • Plaintext array $x = \big(x[1], \ldots, x[n]\big)$, each element $x[i]$ given in binary representation of length w. • The query/lookup value q. The server has no input.
Outputs: With probability $1 - \varepsilon$, the client's output is $(b^*, x[i^*])$ for $i^* = \min\{\, i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$ and $b^* \in \{0,1\}^{\lceil \log_2 n \rceil + 1}$ the binary representation of the index i^*. The server has no output.

I **Keys generation.** The client performs the following: (1) Select the level $L = \lceil \log_2 \log_2(n/\varepsilon) + \log_2(d_{\mathtt{IsMatch}}) \rceil$. (2) Execute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$. (3) Keep pk, sk with the client. (4) Send pk to the server.

II **Data upload.** For each element $x[i]$ the client performs the following: (1) Pad element $x[i]$ with leading zeros until its size is w bits. (2) Encrypt the padded element bit by bit to receive $[\![x[i]]\!] = \mathsf{Enc}_{\mathsf{pk}}(x[i])$.
The client sends $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$ to the server.

III **Secure Search.** The following steps are executed whenever the client issues a search query.

(a) **Client's search query.** The client encrypts q bit by bit $[\![q]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(q)$ and sends it to the server.

(b) **Server's computation.** Using the public key pk the server computes the following:

(1) For every $i \in [n]$ evaluate the $\mathsf{IsMatch}$ polynomial on $[\![x[i]]\!]$ and the query value $[\![q]\!]$ to get the indicators array of size n: $[\![ind]\!] \leftarrow \big(\mathsf{IsMatch}([\![x[1]]\!], [\![q]\!]), \ldots, \mathsf{IsMatch}([\![x[n]]\!], [\![q]\!])\big)$

(2) Execute the following sub-routine (see Section 4.3.2) on the $[\![ind]\!]$ array: $[\![s]\!] \leftarrow \mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}\big([\![ind]\!]\big)$

(3) Compute a pairwise difference of adjacent indices in $[\![s]\!]$ (the derivative of $[\![s]\!]$):
$[\![s'[1]]\!] \leftarrow [\![s[1]]\!] \pmod 2$ and $\forall i \in [2, n] : [\![s'[i]]\!] \leftarrow [\![s[i]]\!] - [\![s[i-1]]\!] \pmod 2$

(4) Compute $[\![b^*]\!]$, the encrypted binary representation of the location of the single $[\![1]\!]$ in $[\![s']\!]$:
Let $B \in \{0,1\}^{(\lceil \log_2 n \rceil + 1) \times n}$ be the matrix that contains in each column $k \in [n]$ the binary representation of k.
Calculate $[\![b^*]\!] = B \cdot [\![s']\!]$.

(5) Compute for each index $j \in [n]$ and each bit $k \in [w]$: $[\![x[i^*][k]]\!] = \sum_{j=1}^{n} ([\![x[j][k]]\!] \cdot [\![s'[j]]\!])$ (see Section 4.3.3)

(6) Send $([\![b^*]\!], [\![x[i^*]]\!])$ to the client.

(c) **Client's decryption.** The client decrypts $b^* \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![b^*]\!])$, $x[i^*] \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![x[i^*]]\!])$ and outputs $(b^*, x[i^*])$.

Fig. 3. *Binary Raffle* Secure Search Protocol

We also took advantage of HElib's SIMD [54] capabilities and "packed" multiple plaintext values (at least 500) into each ciphertext. We remark that we did not attempt to optimize the SIMD factor besides setting its minimal required value. By slight modification of HElib's level parameter it is often possible to reach much higher SIMD factors, around several thousands.

To summarize, with each CPU core executed the protocol on an input array of n ciphertexts, the total amount of elements processed in each experiment is given by $n' = n \cdot \mathrm{SIMD} \cdot \mathrm{CORES}$ and the client obtained in the end of the experiment $\mathrm{SIMD} \cdot \mathrm{CORES}$ results.

B.2 Experiments Description

Binary Raffle. Our main focus in *Binary Raffle*'s benchmarks was to evaluate the running-time of the protocol as a parameter of total input array size (n'), word width (i.e. bit length w) and error probability (ε).

For word widths of $w > 1$ we use the equality operator (see Equation 1, Section 2) as the selected $\mathsf{IsMatch}$ predicate. Beyond that, we also experiment on elements with single bit ($w = 1$). These experiments on $w = 1$ were meant to filter out the running-time of evaluating the $\mathsf{IsMatch}$ polynomial on all elements (step (2.a) in Figure 3) from the remaining steps of protocol. This can be thought as using an $\mathsf{IsMatch}$ predicate that is the degenerate identity function that does nothing, in order to evaluate the performance of the rest of the protocol on a binary vector of indicators.

For $w \in \{16, 64\}$ input array sizes ranged up to $n' \approx 3 \cdot 10^6$ elements. For $w = 1$ input array sizes ranged up to $n' \approx 20 \cdot 10^6$ elements.

The failure probabilities we experimented with were $\varepsilon \in \{2^{-80}, 2^{-40}, 2^{-20}, 2^{-10}, 2^{-1}\}$. Regarding above failure probabilities, $\varepsilon \in \{2^{-80}, 2^{-40}\}$ can be viewed as a negligible error probability, and any $\varepsilon > 2^{-1}$ as an error probability that allows standard probability amplification (see Section 6.1).

SPiRiT. As mentioned, we used an implementation of *SPiRiT* as a reference point. On array of sizes $n' = n \cdot \mathrm{SIMD} \cdot \mathrm{CORES}$, the deterministic variant was evaluated using $k = \lceil \log^2 n / \log\log n \rceil$ sequential executions for different primes larger then $\log n$. We would like to mention that we did not parallelize these k executions as we already exhausted all available employed parallelism to partition the input array into segments assigned to each CPU core.

Similarly to *Binary Raffle*, *SPiRiT* was executed on elements with $w = 1$. Additionaly, due to relatively low

Parameters: • Integer $n \in \mathbb{N}$. • Failure probability ε.
Input: • Array $v = \big(v[1], \ldots, v[n]\big) \in \{0,1\}^n$.
Output: The array is array $t \in \{0,1\}^n$ as follows. If $v \neq (0,\ldots,0)$, then with probability $1-\varepsilon$, $t[0] = \ldots = t[i^*-1] = 0$ and $t[i^*] = \ldots = t[n] = 1$ for $i^* = \min\{i \in [n] \mid v[i] = 1\}$ (step function). Else (if $v = (0,\ldots,0)$), $t = (0,\ldots,0)$ (with probability 1).
Algorithm:

1. Set $N(\varepsilon) = \lceil \log_2(n/\varepsilon) \rceil$ and sample $N(\varepsilon)$ uniformly random arrays $r_1, \ldots, r_{N(\varepsilon)} \leftarrow\!\!\$\ \{0,1\}^n$
2. Recall that for $v \in \{0,1\}^n$ and $k < n$ we denote $\mathsf{prefix}_k(v) = (v_1, \ldots, v_k)$.
   Compute $N(\varepsilon) \cdot n$ random partial prefix sums: $\forall j \in [N(\varepsilon)], \forall k \in [n] :\ S[j,k] = \langle \mathsf{prefix}_k(v), \mathsf{prefix}_k(r_j) \rangle$
3. Compute the binary step function array $t \in \{0,1\}^n$ where each $k \in [n]$, $t[k]$ is the OR of the values in the k-th column of S:
   $\forall k \in [n] :\ t[k] = 1 - \left(\prod_{j=1}^{N(\varepsilon)} \left(1 - S[j,k]\right) \right) \pmod 2$
4. Return array t

Fig. 4. $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ Algorithm

amounts of RAM (16GB) in our test machine we were unable to execute *SPiRiT* over elements with $w = 64$ for sufficiently large array sizes (n') and had to settle for $w = 16$ only. Given this amount of RAM the maximum array sizes that we were able to process ranged between $n' \approx 0.5 \cdot 10^6$ for $w = 16$ and $n' \approx 2 \cdot 10^6$ for $w = 1$.

The running-time of the randomized variant of *SPiRiT* with error probability $\varepsilon = 2^{-1}$ was obtained by taking the mean and standard deviation over the running-time of $2k = 2 \cdot \lceil \log^2 n / \log\log n \rceil$ executions for different primes larger then $\log n$, as required by the protocol. In some cases, due to RAM restrictions, executions for less than $2k$ (although at least k) primes were performed leading to an outcome of error probability higher than 2^{-1}.

B.3 Experimental Results

Our experimental results are presented below, showing the server's running time for different executions of both *Binary Raffle* and *SPiRiT* protocols. The client's running time for encrypting the query and decrypting the result can be ignored as it is negligible in comparison to the server's operations.

B.3.1 *Binary Raffle* with Negligible Error Probability (vs. *SPiRiT* Deterministic)

First we compare the performance *Binary Raffle* with $\varepsilon = 2^{-80}$ to the deterministic variant of *SPiRiT* for both $w \in \{1, 16\}$ (Figures 5a and 5b).

It can be immediately observed from both graphs that *Binary Raffle* achieves faster execution time in an order of magnitude compared to *SPiRiT*. Also we can observe that for *SPiRiT* with $w = 1$ and $w = 16$ there is an approximate $\times 10$ increase in run time between the first and the second. In comparison, for *Binary Raffle* with $w = 1$ and $w = 16$ the increase in run time is relatively minor.

Notice that the *SPiRiT* curves in both graphs are terminated for smaller array sizes than the ones of *Binary Raffle*. The reason for this is that *SPiRiT* executions for larger array sizes were unable to complete successfully. This occurs due to the increase in required levels for larger primes in HElib and the penalty on RAM that is associated with it.

B.3.2 *Binary Raffle* with Error Probability Half (vs. *SPiRiT* Randomized)

Now we compare the performance *Binary Raffle* with $\varepsilon = 2^{-1}$ to the randomized variant of *SPiRiT* (also with error probability half) for both $w \in \{1, 16\}$ (Figures 5c and 5d).

Again, it can be seen in both graphs that *Binary Raffle* achieves faster execution time in an order of magnitude compared to *SPiRiT*. And again, the *SPiRiT* curves in both graphs are terminated for smaller array sizes than the ones of *Binary Raffle*. This happens, similarly to the described in previous section, because working with large primes in HElib increases RAM consumption.

B.3.3 Impact of Error Probability (ε) on *Binary Raffle*

We executed *Binary Raffle* with different error probabilities $\varepsilon \in \{2^{-80}, 2^{-40}, 2^{-20}, 2^{-10}, 2^{-1}\}$ and observed the effect on run time performance (Figures 5e and 5f).

One can see in the graphs that although we jump from half error probability to a negligible error probability ($\varepsilon = 2^{-80}$) the difference in execution time is around $\times 20 - \times 50$ for words with a single bit and around $\times 2 - \times 3$ for words with 64 bits.

This can be explained by the logarithmic dependence between $1/\varepsilon$ and both degree and overall multiplications of the polynomial executed by the server during the *Binary Raffle* protocol.

B.3.4 Impact of Word Size (w) on *Binary Raffle*

We executed *Binary Raffle* with both $w \in \{1, 64\}$ and observed the effect on run time performance (Figures 5g and 5h).

As opposed to the previous section, in this section the change in execution time is more noticeable as word size w increases. When going from a single bit to 64 bit words the difference in execution is around $\times 20 - \times 40$ for error probability half and around $\times 2$ for negligible error probability ($\varepsilon = 2^{-80}$).

This can be explained by the linear dependence between word size w and both degree and overall multiplications of the polynomial executed by the server during the *Binary Raffle* protocol.

This observation brings into being our improvement to the *Binary Raffle* protocol specified in section 5.1.

B.3.5 Other Matching Criteria

The benchmarks below were performed with the same setup described in Section B.1 with the exception of using a server with different CPU (less cores yet each core more powerful): Intel Core i7-4790 CPU running at 3.60GHz on 8 cores, 8MB Cache and 16GB RAM.

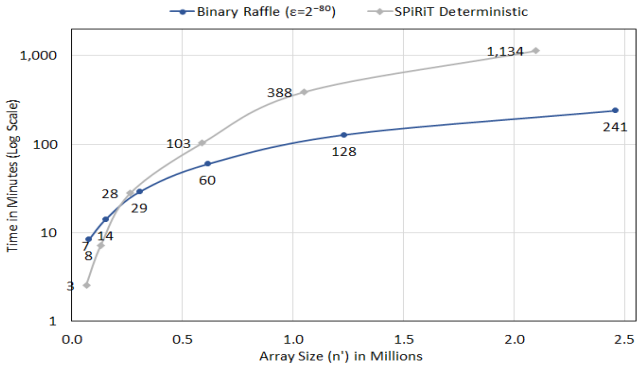
Experiment 1 was the conjunction between two equality queries over an array with two $w = 64$ bit subfields (see Section 5.2). Experiment 2 was a range query, with lower and upper values restriction, over an array with $w = 64$ bit unsigned integers (see Section 5.4). Experiment 3 was of equality queries over an array with $w = 64$ bit with 32 publicly known wildcard positions (see Section 5.3). The results are presented in Table 4.

In the results one can clearly see the relative overhead of the additional multiplications performed during the execution of IsGrt_{64} (Equation 2, Section 2) in the second experiment compared to $\mathsf{IsEqual}_{64}$ (Equation 1, Section 2) in the first experiment. Similarly, the third experiment, in which the matching criteria includes execution of solely $\mathsf{IsEqual}_{32}$, is even faster than the two previous experiments.

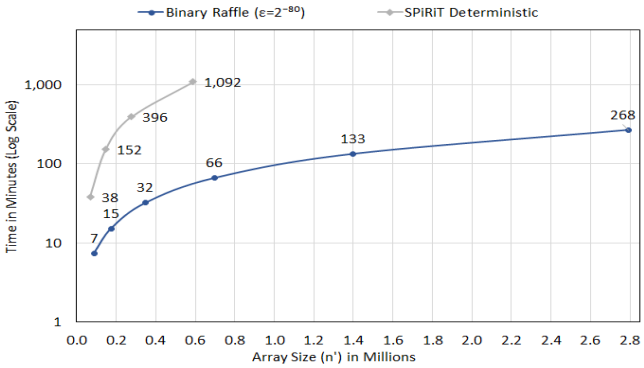
	(i) Conjunction		(ii) Range		(iii) Wild-cards	
Error Prob.	Array Size	Time (min.)	Array Size	Time (min.)	Array Size	Time (min.)
2^{-1}	38K	5	44K	23	38K	1
	77K	10	87K	50	77K	3
	154K	19	175K	98	154K	5
	307K	37	349K	195	307K	10
2^{-40}	44K	7	47K	50	44K	3
	87K	15	92K	99	87K	6
	175K	30	184K	199	175K	11
	349K	60	369K	396	349K	23
2^{-80}	44K	8	46K	55	44K	4
	87K	17	92K	111	87K	5
	175K	36	184K	223	175K	16
	349K	68	369K	443	349K	32

Table 4. Binary Raffle benchmarks for various matching criteria:
(i) Conjunction of two $w = 64$ bits equality-test;
(ii) Range queries over $w = 64$ bit unsigned integers;
(iii) Wild-card queries over $w = 64$ bits with 32 wild-card positions. Array sizes are in thousands of elements (denoted, K).

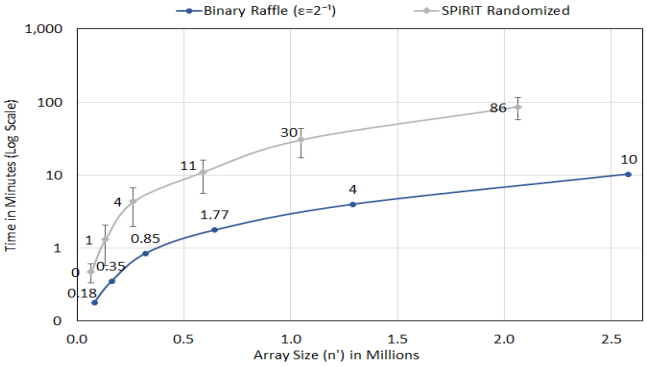
(a) *Binary Raffle* With Negligible Error Probability ($\varepsilon = 2^{-80}$) Versus *SPiRiT* Deterministic for Word Width $w = 1$



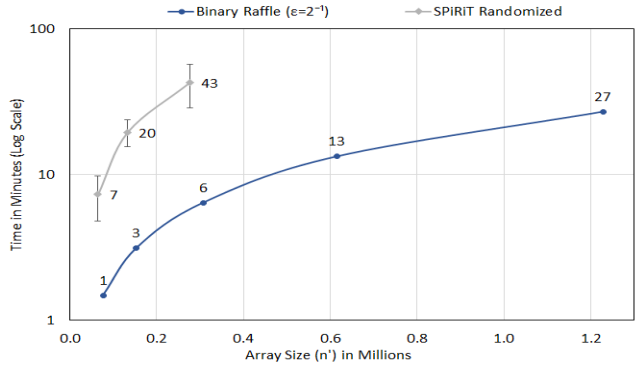
(b) *Binary Raffle* With Negligible Error Probability ($\varepsilon = 2^{-80}$) Versus *SPiRiT* Deterministic for Word Width $w = 16$



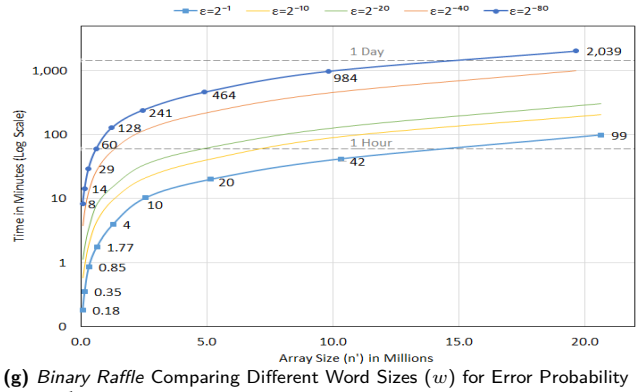
(c) *Binary Raffle* Versus *SPiRiT* Randomized Both With Error Probability $\varepsilon = 2^{-1}$ for Word Width $w = 1$



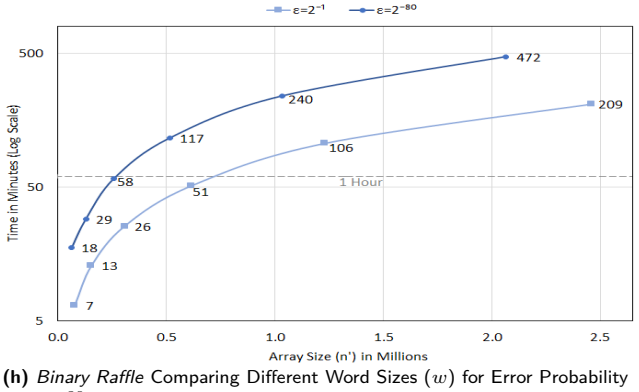
(d) *Binary Raffle* Versus *SPiRiT* Randomized Both With Error Probability $\varepsilon = 2^{-1}$ for Word Width $w = 16$



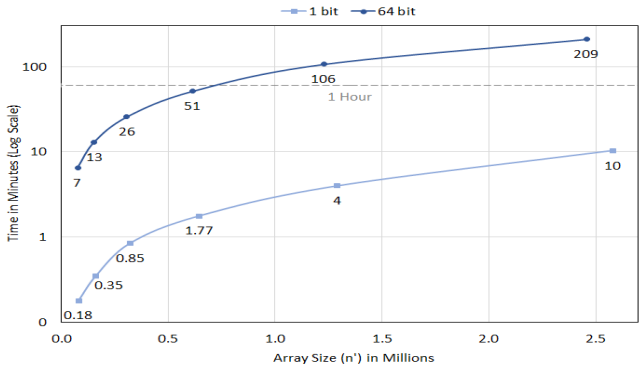
(e) *Binary Raffle* Comparing Different Failure Probabilities (ε) for Word Width $w = 1$



(f) *Binary Raffle* Comparing Different Failure Probabilities (ε) for Word Width $w = 64$



(g) *Binary Raffle* Comparing Different Word Sizes (w) for Error Probability $\varepsilon = 2^{-1}$



(h) *Binary Raffle* Comparing Different Word Sizes (w) for Error Probability $\varepsilon = 2^{-80}$

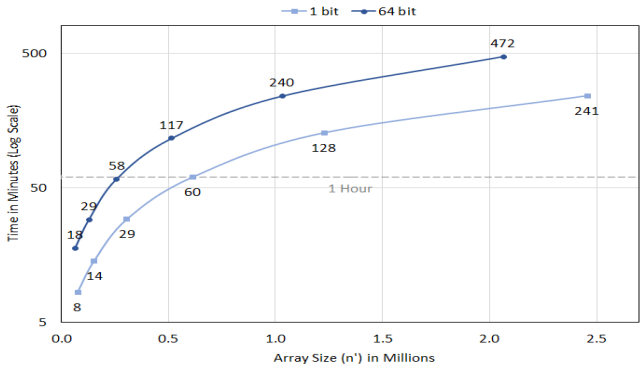


Fig. 5. Server's execution time for different experiments. Y axis – minutes in logarithmic scale; X axis – array size in millions.