Secure Search on Encrypted Data: Faster & Post-Processing Free

by

Max Leibovich

Submitted to the School of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

THE ACADEMIC COLLEGE OF TEL AVIV-YAFFO

August 2018

Author .....
School of Computer Science
August 03, 2018

Certified by .....
Adi Akavia
Doctor
Thesis Supervisor

Secure Search on Encrypted Data:

Faster & Post-Processing Free

by

Max Leibovich

Submitted to the School of Computer Science
on August 03, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Outsourcing storage and computations to the cloud, despite its many celebrated advantages, raises serious privacy concerns. Encrypting the data with Fully Homomorphic Encryption (FHE) prior to uploading to the cloud makes it possible for the client to securely outsource computations to the server. This is because FHE allows processing the underlying clear-text data while it still remains encrypted (and without giving away the secret key). While theoretically feasible, the practical run-time overhead of FHE is notoriously high, with the primary FHE bottlenecks emanating from the degree and overall number of multiplications in the polynomial specifying the desired computation. The challenge therefore is to specify the desired computation via a polynomial of low degree and low number of multiplications.

In this work we focus on the problem of *Secure Search* on FHE encrypted data, which is useful in numerous data analysis and retrieval tasks. Specifically, we focus on settings where the server holds an unsorted array of previously uploaded encrypted elements $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$, the client sends to the server an encrypted query $[\![q]\!]$, and the server returns to the client the encrypted index and element pair $[\![y]\!] = \big([\![i^*]\!], [\![x[i^*]]\!]\big)$ for the first element satisfying the agreed matching condition. We present a novel secure search solution for these settings, improving the prior state-of-the-art (Akavia, Feldman, Shaul, CCS'2018) in being:

- The first secure search solution that simultaneously achieves: (1) full security; (2) efficient server; (3) efficient client; (4) efficient communication; (5) unrestricted search functionality; (6) retrieval of both index and element; (7) post-processing free client; (8) negligible error probability. In contrast, all prior works achieved only a strict subset of these desired properties.

- Faster than the prior-state-of-the-art, attaining:
  (a) optimal client time ($\log^2 n / \log\log n$ speedup), and (b) considerably improved server time (degree reduced from cubic to linear in $\log n$, and overall multiplications reduce by up to $\log n$ factor). Moreover, (c) our solution requires only computa-

tions over GF(2) (in contrast to GF(p) for $p \gg 2$), thus making it compatible with all current FHE candidates.

- We implemented our protocol and ran extensive benchmarks showing promising results: an order of magnitude speedup over the prior state-of-the-art.

Thesis Supervisor: Adi Akavia
Title: Doctor

Acknowledgments

I would like to thank my supervisor Dr. Adi Akavia for her invaluable and inspiring guidance. Next, I would like to thank my companion and partner in life Ira Vitenzon for her unwavering encouragement and assistance. Finally, I would like to thank all my family members and friends for their love and support.

Contents

Chapter 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 clear-text (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 clear-text data in any meaningful way.

Fully Homomorphic Encryption (FHE, [31], [14]) is an encryption scheme that allows processing the underlying clear-text data while it still remains in encrypted form, and without giving away the secret key. 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 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 $[\![y]\!] \leftarrow \mathsf{Eval}_{\mathsf{ek}}(f, [\![x]\!])$ (see section 2.2) 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.

For example, for data in binary representation, bitwise operations on plaintext bits (addition/multiplication modulo 2) can be replaced by their homomorphic coun-

terparts on encrypted bits (homomorphic-addition/homomorphic-multiplication). In general, the homomorphic computations achievable by the known FHE candidates [5, 30, 13, 16] are specified by a polynomial over a finite ring (i.e. by repeated application of homomorphic-addition and homomorphic-multiplication for that ring).

The key factors that influence the run time of such homomorphic computations is the degree and overall multiplications of the polynomial. This gives birth to the main 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 the computation of $y = f(x)$ to the server has the benefits of requiring only a single round of communication, and 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. Indeed, an abundance of proposed solutions were presented to solve it using different cryptographic tools (see section 1.3 and table 1.1). In particular, Gentry [14] proposed using FHE to securely search on encrypted data. A natural and simple definition for *secure search on FHE encrypted data* (*Secure Search*) is as 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.

1.1 Our Contributions

In this work we present a new and improved solution for secure search on FHE encrypted data, analyze its efficiency compared to previous solutions and demonstrate its concrete run-time performance by providing an implementation (built on top of HElib C++ library [19]) together with extensive experiments.

Contribution 1. Our *Secure Search* solution is the first to simultaneously attain all desired advantages below (see section 5.1.1): (1) full security; (2) efficient server; (3) efficient client; (4) efficient communication;[1] (5) unrestricted search functionality; (6) retrieval of both index and element; (7) post-processing free client; (8) negligible error probability;. In contrast, all prior secure search solutions achieve only a strict subset of these properties (see table 1.1).

Contribution 2. Our *Secure Search* solution is faster than the prior secure search solutions on FHE encrypted data with full security, unrestricted search functionality and both index and element retrieval. In particular, we attain: (1) Optimal client run-time: 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 5.1).

Contribution 3. Our *Secure Search* solution requires computations solely over GF(2) (instead of GF(p) for primes $p > 2$). This leads to compatibility with all currently known candidate FHE schemes (including GSW [16]) unlike [1, 2, 3]. This also allows further run-time speedup when using current FHE schemes implementations, including HElib [19] that implements BGV [5] scheme. The reason for the speedup is that in all current FHE schemes, working over GF(p) for larger primes $p > 2$ cause a general slowdown of all the homomorphic operations and size inflation of the keys and ciphertexts (see section 5.1.3).

[1] i.e single round communication, transmitting only the encrypted index and encrypted output.

Contribution 4. We implemented our Secure Search solution based on the FHE HELib C++ library [19]. We performed extensive run-time benchmarks on a mid-range Linux server (16 CPU cores and 16GB RAM). We achieve faster run-time in an order of magnitude compared to previous leading solution with similar characteristics. A few examples of comparing our results to [1, 2, 3] on same server and with similar parameters for bits per element, execution time and error probability follow (see more details in section 6.3):

(i) We securely search on $\approx 3 \times 10^6$ (in contrast to $\approx 0.1 \times 10^6$) 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 $\approx 0.2 \times 10^6$) 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) 64-bit elements in 1 hour, with error probability 1/2.

(iv) We securely search on $> 10 \times 10^6$ (in contrast $\approx 0.5 \times 10^6$) 1-bit elements in 1 hour, with error probability 1/2.

1.2 Our Techniques Highlights

When considering *Secure Search* over unsorted FHE encrypted data, approaches like binary-search are rejected immediately. More so, even if the data is sorted, binary-search leaks which elements were ignored and does not provide secure search.[2] Namely, any secure search solution must "touch" every element in the array.

The approach proposed by Akavia et. al. [1, 2, 3] 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

[2]We note that when permitting multiple rounds of interaction (which is beyond the focus of this work), secure binary search is feasible, e.g., using ORAM.

	SE	Folklore	PIR	PSI	SPM	2PC*	SPiRiT Det.	SPiRiT Rand.	**BR**
Full security	×	✓	✓	✓	✓	✓	✓	✓	✓
Efficient server	✓	×	✓	✓	✓	✓	✓	✓	✓
Efficient client	✓	✓	✓	✓	×	✓	✓	✓	✓
Efficient communication	✓	✓	✓	✓	×	×	✓	✓	✓
Unrestricted search functionality	✓	✓	×	×	✓	✓	✓	✓	✓
Retrieval of both index and element	✓	✓	✓	×	×	✓	✓	✓	✓
Post processing free client	✓	✓	✓	✓	×	✓	×	✓	✓
Negligible error probability	✓	✓	✓	✓	✓	✓	✓	×	✓

Table 1.1: Comparison with prior secure search solutions. Rows list desired properties, columns list prior works. A cell containing ✓ denotes that the property in that row is attained by the work in that column (× means not attained). See section 5.1.1 for properties specification, see section 1.3 for details on prior works.
SE – Searchable Encryption (section 1.3.1); Folklore – FHE Based Folklore Solutions (section 1.3.2); PIR – Private Information Retrieval (section 1.3.4); PSI – Private Set Intersection (section 1.3.5); SPM – Secure Pattern Matching (section 1.3.6); 2PC* – Secure Two-Party Computation Protocols Without the Use of FHE (section 1.3.7); SPiRiT – Secure Search via Multi-Ring Sketch (section 1.3.8); **BR – This work: *Binary Raffle***.

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).

Realizing this approach is challenging as already step (2) is extremely expensive: It requires working over plaintext spaces larger than the array size. Or alternatively, working with binary plaintext space and requiring the use of full-adders. Both of these options cause the polynomials realizing them to have large degrees and extensive amounts of overall multiplications.

To address this challenge Akavia et. al. [1, 2, 3] 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 a new alternative for the probabilistic test combining steps (2)+(3). Our probabilistic test is over GF(2), attains negligible error probability, and requires no post-processing. Specifically, 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.

1.3 Related Works

In this section we present a literature survey of the works most relevant to our *Secure Search* problem. This literature survey focuses on works that execute on FHE encrypted data, alongside a limited survey of works that utilize other cryptographic techniques.

1.3.1 Searchable Encryption (SE)

Searching over encrypted data is the main goal for works in the field of *Searchable Encryption* (SE) (see thorough survey in [4]) that emerged around 20 years ago in the seminal work of Song et. al. [36]. This field focuses on the inherent trade-off of efficiency versus security. Specifically, all the protocols in this approach deliberately leak some information to enable the search functionality over encrypted data. Therefore they do not satisfy the full security requirement (see (i) in 5.1.1).

1.3.2 FHE Based Folklore Solutions (Folklore)

The concept of FHE was first suggested in 1978 by Rivest, Adleman and Dertouzos [31] providing the ability to process encrypted data without giving away access to it (using "privacy homomorphisms"). The first candidate FHE construction came thirty years later with Gentry's seminal work in 2009 [14]. Since then, many works followed towards constituting alternative FHE scheme candidates (for example: BGV [5], GSW [16], LTV [30]). Over time concrete implementations of FHE schemes started to appear with the prominent example of HELib [19] implementing the BGV scheme. For an extensive survey of FHE see [20].

Already in Gentry's work [14] appeared the proposition to utilize FHE for securely searching on encrypted data. In contrast to *Searchable Encryption* (see section 1.3.1), FHE based protocols benefit from the strong property of semantic security of the underlying FHE scheme. This results in full security for the stored data and access pattern, submitted queries and the returned results, both during the execution of the protocol and afterwards. Nevertheless, folklore secure search FHE solutions for

unrestricted settings (as opposed to restrictions that will be described below) suffered from inefficient server runtime due to evaluating degree $\Omega(n)$ polynomials (for n the number of elements).

1.3.3 Relaxed Settings

We note that there are FHE based solutions in alternative settings, where the data array itself is stored in a non encrypted form and only the lookup value and returned result stay hidden from the server, e.g. [32]. This is in contrast to the focus of this work that is the secure outsourcing of encrypted data while preserving the ability of secure search over it.

1.3.4 Private Information Retrieval (PIR)

Another relaxation is when we enforce uniqueness constrain on the stored elements. For these settings FHE based PIR solutions (see [6, 12, 8]) with low degree polynomials are known. Specifically, the degree is essentially the degree of equality comparison polynomial (see 2.3) which is in turn $\mathcal{O}(\log n)$ when the lookup value is the unique index $i \in [n]$.

1.3.5 Private Set Intersection (PSI)

FHE based *Private Set Intersection* (PSI) [9] can also be applied to solve a problem that is similar to the secure search problem. In the PSI solution settings the server is able to compute homomorphiclly the intersection between the set S of stored values and the set Q that contains the submitted lookup value by the client. Because FHE based PSI solutions work on sets (rather than multi-sets) it produces a uniqueness constraint on the stored values, exactly as in FHE based PIR solutions. Furthermore, the output of these protocols only solves the decision problem of whether the lookup value appears in the database with no ability to retrieve the actual record itself later on.

1.3.6 Secure Pattern Matching (SPM)

Additional variation on our settings is in the case of *Secure Pattern Matching* (SPM) [10, 11, 37, 25, 26, 28, 40]. In these works the data, submitted lookup value, and returned result indeed remain hidden from the server during the execution of the protocol. The difference lies in the result that is given as a binary vector of indicators for the matched positions with length proportional to the number of elements n. Once the client receives this indicators vector she can now decrypt it and commence another round of interaction with the server. In this round the client will engage in a PIR protocol with the server to retrieve the actual record at the desired position. The main drawbacks of these protocols are the communication complexity and client running time that are proportional to the number of stored elements $\Omega(n)$.

1.3.7 Secure Two-Party Computation (2PC*)

Classical works on *secure two-party computation* from the 80s [39, 17] 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. Since then, various works showed how to solve the secure search problem by allowing the client and server to engage in a secure two-party computation multi-round interactive protocol that does not necessarily involve the use of FHE (2PC*). These kind of protocols usually require a communication complexity that is at least polynomially dependent in the complexity of computing the search function itself. This is in contrast to being solely dependent in the size of the lookup value and returned result as in efficient FHE based solutions that are discussed in this article.

1.3.8 Secure Search via Multi-Ring Sketch (SPiRiT)

Recent work by Akavia et al. [1, 2, 3] provides a FHE based secure search protocol that addresses all aforementioned drawbacks (sections 1.3.1-1.3.7) and attains desired advantages 1-6 (see section 1.1, contribution 1). Their suggested protocol includes

both deterministic and randomized variants. The deterministic variant uses modern data summarization techniques known as sketches [38] 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.

As an efficiency improvement, a randomized variant was also suggested there. This variant is similar to the deterministic one except for working over a single random ring instead of several different rings. This randomized variant achieves an error probability that is only polynomially small in n rather than a negligible error probability.

SPiRiT Deterministic Variant (SPiRiT Det.) The deterministic variant of the *SPiRiT* protocol assumes that each of the n elements are stored as a tuple of $k = \mathcal{O}\left(\log^2 n / \log\log n\right)$ ciphertexts. For any given record this tuple consist of encryptions of the same value under different plaintext moduli $p_1, \ldots, p_k$. These moduli are chosen to be the first k primes larger then $\log n$, all of magnitude $\mathcal{O}\left(\log^2 n\right)$.

To find the index of the first match for a lookup value q the client sends k encryptions of q with respect to the k different prime plaintext moduli $p_1, \ldots, p_k$. The next steps in the protocol (described below) are repeated k times (possibly in parallel) with respect to each of these different plaintext moduli: First, the server evaluates the desired IsMatch polynomial over the lookup value and the stored elements to obtain an encrypted indicator vector with $[\![1]\!]$ in positions where matches occurred and $[\![0]\!]$ in the remaining positions. Second, the server applies the *SPiRiT* First Positive sketch on the encrypted indicator vector to calculate the binary representation of index of the first match. Finally, this encrypted binary representation of the index is returned to the client. A list of candidate elements is also obtained by evaluating *PIR* polynomials on the encrypted candidate indices and stored elements.

At the termination of the above computation, the client has received k encrypted candidate index and element pairs. Finally, the client recovers the actual first match

by decrypting the (index, element) pairs and choosing the matched element with the minimal index.

This protocol achieves full security due to the fact that the stored elements, lookup value and returned response are encrypted with FHE. The protocol has unrestricted search functionality and does not enforce the constraint of uniquely identifiable elements. It achieves communication complexity proportional to the size of k encrypted lookup values and k result candidates. The clients running time is proportional to k encryption/decryption operations. The server evaluates k times (in parallel) the composition of the desired $\mathsf{IsMatch}$ polynomial, $SPiRiT$ and PIR polynomials. The task of computing the index of the first match is realized by polynomials of degree $\mathcal{O}\left(\log^3 n\right)$ and the total number of multiplications over all polynomials is $\mathcal{O}\left(n \log^2 n\right)$.

SPiRiT Randomized Variant (SPiRiT Rand.) The randomized variant of the $SPiRiT$ protocol, in which only a single prime is used, can achieve only polynomially small error probability. For error probability $\varepsilon \in (0, 1)$ this prime is chosen uniformly at random from the set of first k/ε primes larger then $\log n$. In this randomized version if we would like to have a negligible error probability the chosen prime needs to be selected from the set of $n^{\omega(1)}$ primes larger then $\log n$ which eventually causes the overall degree to go up to an infeasible value of $n^{\omega(1)}$ (which is worse even than the folklore solution).

Chapter 2

Preliminaries

2.1 Notations

For natural numbers $k < n$, denote $[n] = \{1, \ldots, n\}$ and $[k, n] = \{k, \ldots, n\}$. For array v denote $v[i]$ the i-th element in v. 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 $M[i, j]$ and $\mathsf{row}_i(M)$, $\mathsf{col}_j(M)$ will denote the i-th row and j-th column of M respectively.

For a field $\mathbb{F}$, vectors $v, u \in \mathbb{F}^n$ and $k \in [n]$, denote:

- $\mathsf{weight}(v) = |\{i \in [m] \mid v[i] = 1\}|$
- $\langle v, u \rangle = \sum_{i=1}^{n} v[i] \cdot u[i] \pmod 2$
- $\mathsf{prefix}_k(v) = (v_1, \ldots, v_k) \in \mathbb{F}^k$
- $\mathsf{suffix}_k(v) = (v_{k+1}, \ldots, v_n) \in \mathbb{F}^{n-k}$
- $|v|$ – size (length, dimension) of v $(= n)$

Let $k, n \in \mathbb{N}$, denote $r_1, \ldots, r_k \leftarrow\!\!\$\, \{0,1\}^n$ as sampling k arrays independently at random from the uniform distribution over $\{0,1\}^n$.

2.2 Fully Homomorphic Encryption

Definition 2.2.1. *A leveled homomorphic (public-key) encryption scheme* $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *is a quadruple of probabilistic polynomial time (*PPT*) algorithms as follows.*

- ***Key generation.*** *The algorithm* $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$ *takes a unary representation of the security parameter* κ *in addition to a unary representation of the circuit depth upper-bound* L *and outputs a public encryption key* pk*, a public evaluation key* ek *and a secret decryption key* sk*.*

- ***Encryption.*** *The algorithm* $[\![x]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x)$ *takes the public key* pk *and plaintext message* $x \in \{0,1\}$ *and outputs a ciphertext message* $[\![x]\!]$*. Also, we abuse the notation and for any plaintext array* $y \in \{0,1\}^n$ *the algorithm* $[\![y]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(y)$ *takes the public key* pk *and plaintext array* $y \in \{0,1\}^n$ *and performs the encryption bit by bit resulting in ciphertext array* $[\![y]\!] = \big([\![y[1]]\!], \ldots, [\![y[n]]\!]\big)$*.*

- ***Decryption.*** *The algorithm* $x' \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![x]\!])$ *takes the secret key* sk *and ciphertext message* $[\![x]\!]$ *and outputs a plaintext message* $x' \in \{0,1\}$*. Also, we abuse the notation and for any ciphertext array* $[\![y]\!] = \big([\![y[1]]\!], \ldots, [\![y[n]]\!]\big)$ *the algorithm* $y' \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![y]\!])$ *takes the private key* sk *and ciphertext array* $[\![y]\!]$ *and performs the decryption ciphertext by ciphertext resulting in plaintext array* $y' = \big(y'[1], \ldots, y'[n]\big)$*.*

- ***Homomorphic evaluation.*** *The algorithm* $[\![r]\!] \leftarrow \mathsf{Eval}_{\mathsf{ek}}(f, [\![x[1]]\!], \ldots, [\![x[t]]\!])$ *takes the evaluation key* ek*, a function* $f : \{0,1\}^t \rightarrow \{0,1\}$ *represented as an arithmetic circuit over* $GF(2)$ *and a set of* t *ciphertexts* $[\![x[1]]\!], \ldots, [\![x[t]]\!]$*, and outputs a ciphertext* $[\![r]\!]$ *such that* $\mathsf{Dec}_{\mathsf{sk}}([\![r]\!]) = f(x[1], \ldots, x[t])$*.*

2.3 Implementing Comparison Operators with FHE

We present implementations of several binary comparison operators later to be used in our protocols. For $x \in \{0,1\}^w$ the least significant bit (LSB) is located in $x[1]$ and the most significant bit (MSB) is located in $x[w]$.

- For $x_1, x_2 \in \{0,1\}^w$ equality operator $(x_1 = x_2)$ can be implemented as

$$\mathsf{IsEqual}_w(x_1, x_2) = \prod_{i \in [w]} \Big(1 + x_1[i] + x_2[i]\Big) \pmod 2$$

  Can be evaluated by a polynomial with degree of w, and $w - 1$ overall multiplications.

- For $x_1, x_2 \in \{0,1\}^w$ "greater-then" operator $(x_1 > x_2)$ can be implemented as

$$\mathsf{IsGreaterThen}_w(x_1, x_2) = \sum_{i \in [w-1]} \Big[\big(x_1[i] \cdot (x_2[i]+1)\big) \cdot \mathsf{IsEqual}_{w-i}\big(\mathsf{suffix}_i(x_1), \mathsf{suffix}_i(x_2)\big)\Big]$$

$$+\big(x_1[w] \cdot (x_2[w]+1)\big) \pmod 2$$

  Can be evaluated by a polynomial with degree of $w + 1$, and $\frac{1}{2}w(w+1)$ overall multiplications.

2.4 BGV FHE Scheme & HElib Implementation

The HElib [19] implementation of the BGV [5] homomorphic encryption scheme includes Smart-Vercauteren [35] Single Instruction Multiple Data (SIMD) optimization alongside many other optimizations [15, 21, 22]. We now point out several characteristics of BGV and HElib that will play key roles in the run-time efficiency analysis of our implementations. More details on these characteristics and general design of HElib can be obtained from [23, 19].

Plaintext Space This BGV variant in HElib is defined over polynomial rings of the form $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$ where m is a parameter and $\Phi_m(X)$ is the m'th cyclotomic polynomial. Let $\mathbb{A}_q = \mathbb{A}/q\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X), q)$ for integer q, and identify $\mathbb{A}_q$ as the set of integer polynomials of degree up to $\phi(m) - 1$ reduced module q (where $\phi(m)$ is Euler's totient function). The "native" plaintext space for HElib is the ring $\mathbb{A}_2$, although after HElib incorporated additional optimizations, other plaintext spaces, including $\mathbb{A}_p$ for any arbitrary prime p, are also available. Enlarging the plaintext

space $\mathbb{A}_p$ to larger prime values leads to a general slowdown in all the homomorphic operations and size inflation of the keys and ciphertexts.

Plaintext Packing & SIMD Smart-Vercauteren optimization allows to "pack" many plaintext elements in a single ciphertext and apply to them operations in a SIMD manner. We refer to the different plaintext values in a single ciphertext as the "plaintext slots" of that ciphertext. This is achievable by factoring the polynomial $\Phi_m(X)$ into s irreducible factors modulo 2, $\Phi_m(X) = F_1(X) \cdot \ldots \cdot F_s(X) \pmod 2$, all of degree $d = \phi(m)/s$. Now we can view a polynomial $a \in \mathbb{A}_2$ as representing a vector $(a \mod F_i)_{i=1}^s$ that holds s encodings of plaintext values. Notice that the amount of plaintext slots s is dependent on the value $\phi(m)$ (the degree of the cyclotomic polynomial $\Phi_m(X)$), thus higher values of $\phi(m)$ will usually enable more plaintext slots. It is important to mention that in addition to enabling more plaintext slots, incrementing $\phi(m)$ will also cause a general slowdown of all the homomorphic operations and size inflation of the keys and ciphertexts.

Leveled FHE As BGV is a leveled FHE scheme, the ciphertext space for this scheme consists of vector over $\mathbb{A}_q$, where q is a large odd modulus that evolves with the homomorphic evaluation. Specifically, the system is parametrized by a "chain" of moduli of decreasing size, $q_0 < q_1 < \ldots < q_L$ and freshly encrypted ciphertexts are defined over $\mathbb{A}_{q_L}$. During homomorphic evaluation, after each multiplication, to handle the increasing "noise", a switching to smaller and smaller moduli is preformed until a ciphertext over $\mathbb{A}_{q_0}$ is obtained, on which further computations are impossible.

When working over plaintext space $\mathbb{A}_p$ for $p > 2$ the above operation of ciphertext "refresh" after each multiplication can "consume" several such levels. This is caused by the rounding operation during modulus switching where additional "noise", proportional to $\mathsf{poly}(p)$, is added to the ciphertext. Therefore, when working with plaintext space with higher value of p, additional levels (and additional primes q_i) are necessary in order to enable successful computation.

Security Parameter We set the security parameter of HElib to 80 bit (same as Akavia et al. [1, 2, 3] and other works [24, 29, 25]). Changing this parameter is possible in the initialization phase of HElib.

Another aspect of the security parameter is that it brings about a linear increase in $\phi(m)$, which in term will cause a longer computation time of homomorphic operations and increase in ciphertexts and keys sizes.

2.5 Universal Family of Hash Functions

Carter and Wegman [7] defined the notion of a universal family of hash functions:

Definition 2.5.1 ([7])**.** *A family of hash functions $\mathcal{H} : \mathcal{W} \to \mathcal{V}$ is said to be* ***strongly universal*** *if for every $h \in \mathcal{H}$ and for all $x \neq y \in \mathcal{W}$,*

$$\Pr_{h \in \mathcal{H}} \left[h(x) = h(y) \right] \leq \frac{1}{|\mathcal{V}|}$$

Theorem 2.5.2. *Given a hash function h sampled randomly from a family of* ***strongly universal*** *hash functions $\mathcal{H} : \mathcal{W} \to \mathcal{V}$ and n distinct values $x_1 \ldots, x_n \in \mathcal{W}$ ($\forall i \neq j \in [n] : x_i \neq x_j$), the hashed values $h(x_1), \ldots, h(x_n)$ are all distinct with probability*

$$\Pr_{h \in \mathcal{H}} \left[\forall i \neq j \in [n] : h(x_i) \neq h(x_j) \right] > 1 - \frac{\binom{n}{2}}{|\mathcal{V}|}$$

Proof. There are exactly $\binom{n}{2}$ possible value pairs among $x_1, \ldots, x_n$. From the property of the hash function being strongly universal we get that for each value pair $x_i \neq x_j$ for $i \neq j$ the probability for a collision ($h(x_i) = h(x_j)$) is at most $|\mathcal{V}|^{-1}$. Finally, by applying the union bound over all possible value pairs we get that the total probability for any collision is at most $\binom{n}{2} \cdot |\mathcal{V}|^{-1}$ as required. □

Corollary 2.5.1. *Given security parameter κ, and suppose that $|\mathcal{V}| = 2^v$, if $v = 2\log_2(n) + \omega(\log_2(\kappa))$ then we get*

$$\Pr_{h \in \mathcal{H}} \left[\forall i \neq j \in [n] : h(x_i) \neq h(x_j) \right] = 1 - \mathsf{negl}(\kappa)$$

Proof. Immediate by using $2^{-\omega(\log_2(\kappa))} = \mathsf{negl}(\kappa)$ and assigning $v = 2\log_2(n) + \omega(\log_2(\kappa))$ in Theorem 2.5.2. □

Theorem 2.5.3 ([7]). *Let $\mathcal{W} = \{0,1\}^w$ and $\mathcal{V} = \{0,1\}^v$. We now provide the following construction of hash function family: Sample random matrix $A \in \{0,1\}^{w \times v}$ and vector $b \in \{0,1\}^v$, for any $x \in \{0,1\}^w$ the hash value $h(x) \in \{0,1\}^v$ will be*

$$h(x) = Ax + b \pmod 2$$

*This is a **strongly universal** family of hash functions that contains $2^{(w+1)v}$ functions.*

In the above construction, describing the hash function requires $\mathcal{O}(w \cdot v)$ bits. We can reduce this storage overhead with the corollary below.

Definition 2.5.4 (Toeplitz Matrix). *$A \in \{0,1\}^{w \times v}$ is defined as following: Fill the first row $A_{1,1}, \ldots, A_{1,w}$ and the first column $A_{1,1}, \ldots, A_{v,1}$ with random bits. For every other entry $A_{i,j}$ for $i > 1$ and $j > 1$ define $A_{i,j} = A_{i-1,j-1}$. So all entries in each "northwest-southeast" diagonal in A are the same.*

Corollary 2.5.2 ([27]). *Let $\mathcal{W} = \{0,1\}^w$ and $\mathcal{V} = \{0,1\}^v$. The construction of the following hash function family will be identical to the one in Theorem 2.5.3 with the sole difference that $A \in \{0,1\}^{w \times v}$ will be a random Toeplitz Matrix. This is a **strongly universal** family of hash functions that contains $2^{(w+v-1)+v}$ functions, its description requires only $\mathcal{O}(w+v)$ bits.*

Chapter 3

Problem Statement

Suppose a client (Alice) wants to use a 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 FHE Encrypted Data

In this paper we focus on the problem of secure search on FHE encrypted data. In the *Secure Search* problem (see definition 3.1.1 below) 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 $\forall j < i^* : \mathsf{IsMatch}(x[j], q) = 0$).

In order for the *Secure Search* to be generic and applicable to different problems we emphasize that it can be instantiated with various IsMatch predicates. In general the predicate IsMatch is defined as a function that receives two elements and returns a binary indicator whether they are considered matching in the context of the given problem. The IsMatch predicate can have various domain specific implementations like: exact equality operator, conjunction/disjunction query, range query, edit

distance, Hamming distance, Euclidean distance, and so forth.

For example, our conducted run-time experiments use the exact equality (see section 2.3) as the concrete $\mathsf{IsMatch}$ implementation. We also present several other instantiations to $\mathsf{IsMatch}$ that achieve: (1) performance improvement of exact equality using hash functions (see section 7.1); (2) searching in sub-array (see section 7.2); (3) sequential retrieval of additional results besides the first match (see section 7.3).

Definition 3.1.1 (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:*

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

Putting it all together A possible use case scenario for a solution to the *Secure Search* problem is as follows (see figure 3-1):

- *Keys Generation*: Alice initializes the scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ and generates the keys $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek})$. Alice keeps pk, sk and sends ek to Bob.
- *Array Upload*: Alice gradually, over time, encrypts and uploads elements to Bob. At any given moment, Bob holds an encrypted and unsorted array $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$.
- *Secure Search:* At any time, Alice may issue search query q by encrypting it and sending $[\![q]\!]$ to Bob. After homomorphic evaluation Bob returns encrypted search outcome of the index of the first match $[\![i^*]\!]$ and corresponding element $[\![x[i^*]]\!]$ to Alice. Finally, Alice decrypts both and obtains i^* and $x[i^*]$.

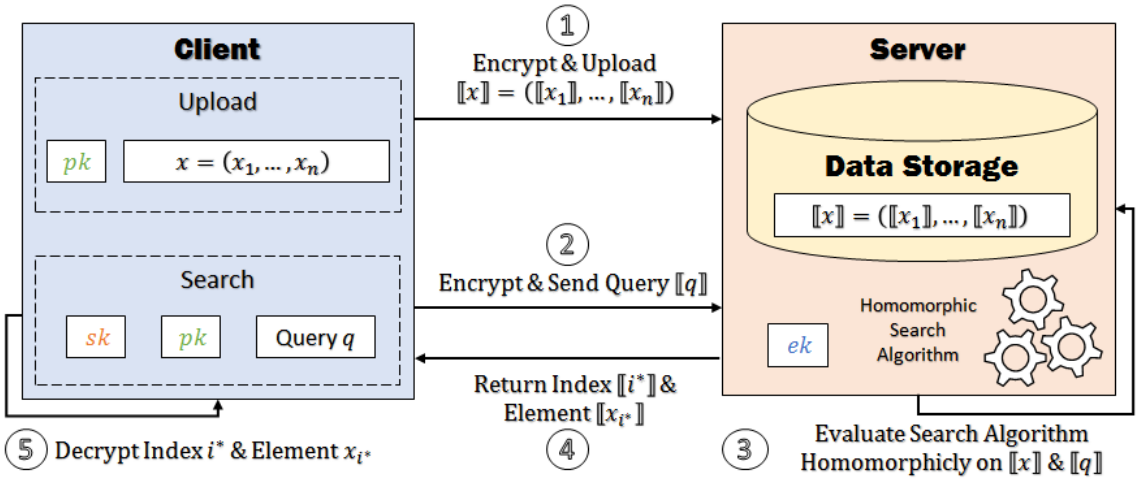


Figure 3-1: Depiction of *Secure Search* on FHE encrypted data

3.2 Extensions

Although, for the sake of clarity, our problem statement above is phrased in a limited form, we present several extensions that augment it with additional functionality.

3.2.1 Search In Sub-Array

The client (Alice) can perform the *Secure Search* on a sub-array. This is by using an augmented IsMatch predicate that, in addition to performing the matching logic, includes a homomorphic check against provided encrypted array's lower and upper bounds (see section 7.2).

3.2.2 Search & Retrieve Multiple Elements

The client (Alice) can retrieve a sequence of multiple matches for a given query q. This is achieved by adding interaction and repeated invocation of a variant to the *Secure Search* protocol. This *Secure Search* variant includes an enhanced IsMatch predicate the uses auxiliary data to fetch the next match for query q every time it is being invoked (see section 7.3).

3.2.3 Dynamic Data Management

The client (Alice) 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 implemented 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 size of the data is either maintained by the client, or is not a secret and can be maintained by the server).

3.2.4 Multiple Clients with Multiple Roles

The client (Alice) can be segmented into multiple parties:

1. *Keys Generation Authority* – Responsible for keys generation and their distribution between other parties.

2. *Data Source* – Provides the array of elements. Responsible for their encryption and for uploading the encrypted elements to the *Search Server*. Needs only the public key pk.

3. *Search Client* – Issues encrypted search queries to the *Search Server* and receives encrypted responses from it. Needs both public key pk and secret key sk.

Another extension can be having multiple instances of the above parties (except the *Keys Generation Authority*) that simply share their keys: multiple *Search Clients* that share both pk and sk, multiple *Data Sources* that share pk.

3.3 Threat Model

The untrusted party in our scenario is the *honest-but-curious* server. For the upload functionality the server provides the storage facility but is prohibited from modifying or destroying stored elements. For the search functionality the server receives queries and is obligated to follow the protocol and return search outcomes accordingly. On the other hand, the server can try to derive sensitive information from the stored elements, received queries, data access patterns and search outcomes.

We say that a protocol *provides security for the client* against the above server if any probabilistic polynomial time (PPT) adversary controlling the server has no more than a negligible advantage over a random guess in winning games (1) or (2) below.

Game 1 – Protecting Queries and Access Patterns The adversary chooses two queries q_1, q_2 of same size $|q_1| = |q_2|$. The client chooses a uniformly random $b \in \{0,1\}$ and executes the secure search protocol with query q_b. The adversary outputs a bit $b' \in \{0,1\}$, and wins if $b' = b$.

Game 2 – Protecting Stored Data and Access Patterns The adversary chooses two data arrays $x_1 = \big(x_1[1], \ldots, x_1[n]\big)$ and $x_2 = \big(x_2[1], \ldots, x_2[n]\big)$ with all elements of the same size in both arrays. The client chooses a uniformly random $b \in \{0,1\}$ and a query q and initiates the upload and secure search protocols on input x_b and q respectively. The adversary outputs $b' \in \{0,1\}$ and wins if $b' = b$.

Looking ahead, we note that security as defined above follow immediately from the semantic security of the underlying fully homomorphic encryption system.

Leakage We point out that protocols satisfying the above security definition are allowed to leak the following information (as is indeed the case for our protocol): (1) plaintext space; (2) array size upper-bound; (3) element size upper-bound; (4) overall count of executed queries. We stress that the overall count of executed queries does not reveal any information regarding the content or distribution of stored elements. In particular, the server is unable to distinguish between queries that request for "fresh" results and queries that request the next match for a previously issued query (see more details in 7.3).

Chapter 4

Protocols

In this section we describe our protocols for keys generation (section 4.1), array upload (section 4.2) and secure search (section 4.3).

4.1 Keys Generation Protocol (figure 4-1)

In the keys generation protocol the client executes the key generation algorithm of the leveled scheme $\mathcal{FHE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ (see definition 2.2). 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).

In details, the level L will depend 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 $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$ where public key pk and secret kept sk are kept by the client and the evaluation key ek and ε are sent to the server.

4.2 Array Upload Protocol (figure 4-2)

In the array upload protocol 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. The encryption of each element is then performed bit by bit.

4.3 Secure Search Protocol (figures 4-3, 4-4, 4-5)

We now give an overview of our secure search protocol (see figures 4-3, 4-4). In this protocol the client issues search query q by encrypting it and sending $[\![q]\!]$ to the server. After homomorphic evaluation the server returns encrypted search outcome of the index of the first match $[\![i^*]\!]$ and element $[\![x[i^*]]\!]$ to the client. In the end of the protocol, the client decrypts both and obtains i^* and $x[i^*]$.

Sections 4.3.1-4.3.2 describe how the protocol returns the index of the first match. Section 4.3.1 gives the high level of the protocol, and section 4.3.2 describe the key algorithm in this protocol. In section 4.3.3 we describe how to augment this protocol to return also the element on top of the index.

4.3.1 Binary Raffle: Returning the Index (figure 4-3)

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

The secure search protocol proceeds as follows (see figure 4-3): First the client encrypts her search query, $[\![q]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(q)$, and sends it to the server. Next, the server evaluates the steps specified below on the stored array $[\![x]\!]$ and the lookup value $[\![q]\!]$. The server obtains $[\![b^*]\!]$ for $b^* \in \{0,1\}^{\lceil \log_2 n \rceil + 1}$ which is the binary representation of the index of the first match $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$. Afterwards, the

server sends $[\![b^*]\!]$ to the client. The protocol concludes after the client decrypts $[\![b^*]\!]$ to obtain the desired output.

The steps executed by the server are as follows: First, 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, 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

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

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

Now, 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 $[\![1]\!]$.

Finally, the server computes $[\![b^*]\!] = B \cdot [\![s']\!]$ for $B \in \{0,1\}^{(\lceil \log_2 n \rceil + 1) \times (n+1)}$ the matrix that contains in each column $k \in [n]$ the binary representation of k and in column $n+1$ the binary representation of 0. The resulting array $[\![b^*]\!]$ will hold the binary representation of the index i^* of the single $[\![1]\!]$ in $[\![s']\!]$. 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 $j = n+1$).

The outcome $[\![b^*]\!]$ is sent to client, who decrypts it to obtain the binary representation b^* of the first match to his submitted query.

4.3.2 $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ Algorithm (figure 4-4)

We next describe the algorithm that 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 4-4). In addition we provide an illustration for the main steps of the algorithm in figure 4-5.

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.

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 Extending Binary Raffle: Returning Element on Top of Index

Since the problem of privately retrieving a uniquely identifiable element from an encrypted array has efficient FHE based solutions, we first focused above (sections 4-3-4.3.2) 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 1.3.4) 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 $d_{\texttt{PIR}} = \log n$.

Instead we suggest a more efficient alternative for retrieving the matched element $[\![x[i^*]]\!]$. This is by re-using 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:

$$[\![x[i^*]]\!] = \sum_{j=1}^{n} ([\![x[j]]\!] \cdot [\![s'[j]]\!])$$

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.

4.3.4 Protocols Figures

Client Inputs:

- Security parameter κ.
- Error probability ε (upper-bound).
- Array size n (upper-bound).
- Degree of the desired matching polynomial $d_{\texttt{IsMatch}}$ (upper-bound).

Server Inputs:

- None.

Client Outputs:

- The secret key sk and public key pk.

Server Outputs:

- The evaluation key ek.
- Error probability ε.

Protocol:

1. The client preforms the following steps:
   (a) Select the level $L = \lceil \log_2 \log_2(n/\varepsilon) + \log_2(d_{\texttt{IsMatch}}) \rceil$.
   (b) Execute $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek}) \leftarrow \mathsf{KGen}(1^\kappa, 1^L)$.
   (c) Keep pk, sk with the client.
   (d) Send ek and ε to the server.

Figure 4-1: Keys Generation Protocol

Client Inputs:

- Public key pk (see protocol 4.1).
- Plaintext array $x = \big(x[1], \ldots, x[n]\big)$, each element $x[i]$ given in binary representation.
- Element bit length upper-bound w.

Server Inputs:

- None.

Client Outputs:

- None.

Server Outputs:

- Encrypted array $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$.

Protocol:

1. The client preforms the following steps:
   (a) For each element $x[i]$:
      i. Pad element $x[i]$ with leading zeros until its size is w bits.
      ii. Encrypt the padded element bit by bit to receive $[\![x[i]]\!] = \mathsf{Enc}_{\mathsf{pk}}(x[i])$ (as described in section 2.2) .
   (b) Send $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$ to the server.

Figure 4-2: Array Upload Protocol

Client Inputs:

- Secret key sk and public key pk (see protocol 4.1). • The query/lookup value q.

Server Inputs:

- Failure probability ε. • The evaluation key ek (see protocol 4.1).
- Description of the required pattern matching polynomial with binary results $\mathsf{IsMatch}\big([\![x]\!], [\![y]\!]\big) \in \{[\![0]\!], [\![1]\!]\}$.
- An array of n unsorted and not necessarily distinct elements, previously encrypted and uploaded to the server by the client $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$ (see section 4.2 and figure 4-2).

Client Outputs:

- Binary representation $b^* \in \{0,1\}^{\lceil \log_2 n \rceil + 1}$ of the index of the first match $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$ with probability $1 - \varepsilon$.

Server Outputs: • None.
Protocol:

1. The client encrypts the lookup value bit by bit $[\![q]\!] \leftarrow \mathsf{Enc}_{\mathsf{pk}}(q)$ (as described in section 2.2) and sends it to the server.
2. The server preforms the following steps:
   (a) 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:
   $$[\![ind]\!] \leftarrow \big(\mathsf{IsMatch}([\![x[1]]\!], [\![q]\!]), \ldots, \mathsf{IsMatch}([\![x[n]]\!], [\![q]\!])\big)$$
   (b) Execute the following sub-routine (see section 4.3.2) on the $[\![ind]\!]$ array
   $$[\![s]\!] \leftarrow \mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}\big([\![ind]\!]\big)$$
   (c) Compute a pairwise difference of adjacent indices in $[\![s]\!]$ (the derivative of $[\![s]\!]$):
   $$\forall i \in [2, n]: \quad [\![s'[i]]\!] \leftarrow [\![s[i]]\!] - [\![s[i-1]]\!] \pmod 2 \quad \text{and}$$
   $$[\![s'[1]]\!] \leftarrow [\![s[1]]\!], \quad [\![s'[n+1]]\!] \leftarrow [\![1]\!] - [\![s[n]]\!]$$
   (d) Compute $[\![b^*]\!]$, the encrypted binary representation of the location of the single $[\![1]\!]$ in $[\![s']\!]$: Let $B \in \{0,1\}^{(\lceil \log_2 n \rceil + 1) \times (n+1)}$ be the matrix that contains in each column $k \in [n]$ the binary representation of k and in column $n+1$ the binary representation of 0. Calculate $[\![b^*]\!] = B \cdot [\![s']\!]$.
   (e) Send $[\![b^*]\!]$ to the client.
3. The client decrypts $b^* \leftarrow \mathsf{Dec}_{\mathsf{sk}}([\![b^*]\!])$ and outputs b^*.

Figure 4-3: *Binary Raffle* Secure Search Protocol

Parameters:

- Integer $n \in \mathbb{N}$.
- Failure probability ε.

Input:

- Array $v = \big(v[1], \ldots, v[n]\big) \in \{0,1\}^n$.

Output:

- If $v \neq (0, \ldots, 0)$ denote $i^* = \min\{i \in [n] \mid v[i] = 1\}$. With probability $1 - \varepsilon$, the output is the array $t \in \{0,1\}^n$ defined by $t[0] = \ldots = t[i^* - 1] = 0$ and $t[i^*] = \ldots = t[n] = 1$ (step function).

  If $v = (0, \ldots, 0)$ the output is the array $t = (0, \ldots, 0)$ of size n (with probability 1).

Algorithm:

1. Set $N(\varepsilon) = \lceil \log_2(n/\varepsilon) \rceil$ and sample $N(\varepsilon)$ uniformly random arrays

$$r_1, \ldots, r_{N(\varepsilon)} \xleftarrow{\$} \{0,1\}^n$$

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:

$$\forall j \in [N(\varepsilon)], \forall k \in [n]: \quad S[j,k] = \langle \mathsf{prefix}_k(v), \mathsf{prefix}_k(r_j) \rangle$$

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:

$$\forall k \in [n]: \quad t[k] = 1 - \left(\prod_{j=1}^{N(\varepsilon)} \Big(1 - S[j,k]\Big) \right) \pmod 2$$

4. Return array t

Figure 4-4: $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ Algorithm

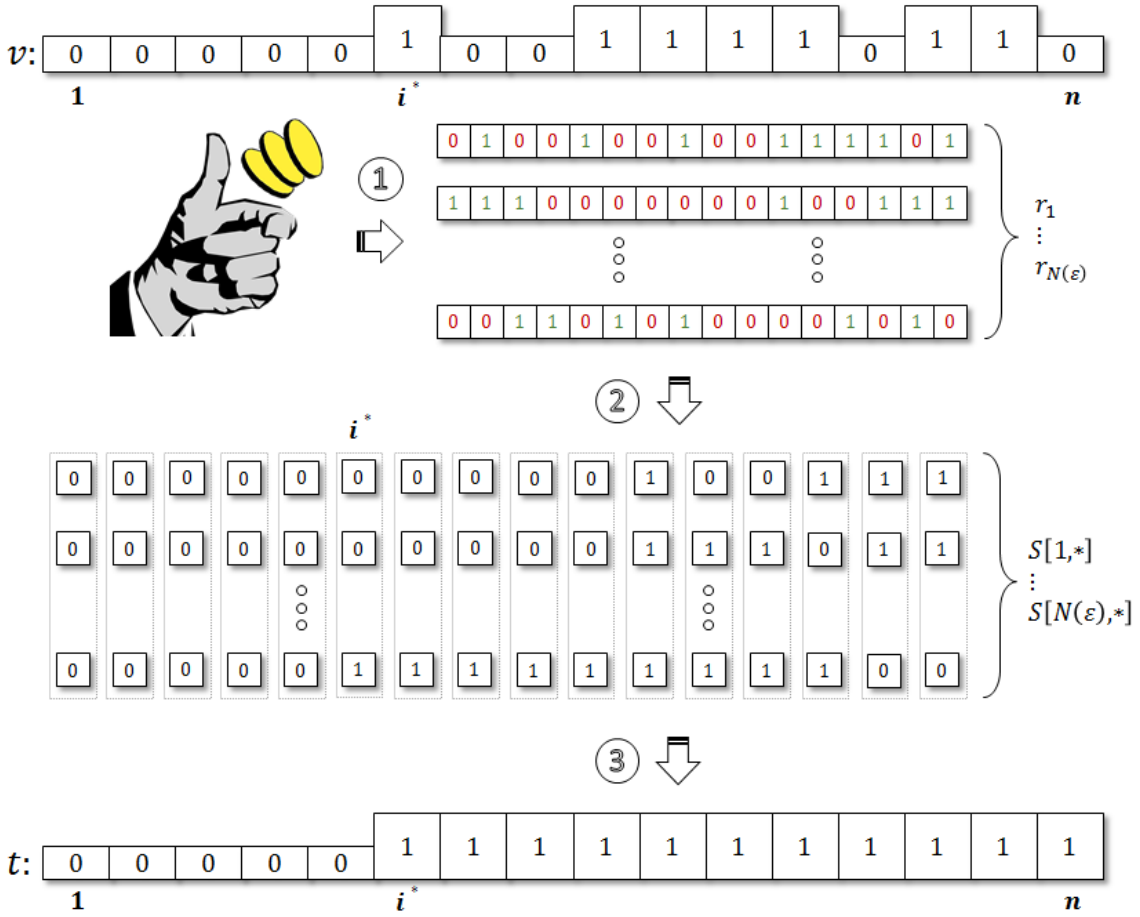


Figure 4-5: An illustration of the main steps in $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm: On input array v the following steps are preformed: (1) Sample $N(\varepsilon)$ random binary arrays $r_1, \ldots, r_{N(\varepsilon)}$ of length n; (2) Matrix S will hold $N(\varepsilon) \cdot n$ 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.

Chapter 5

Theoretical Results

In this section we present our theoretical contribution starting with an overview (section 5.1), and then giving the formal theorems and proofs for the analysis of *Binary Raffle* protocol (section 5.2) and $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm (section 5.3).

5.1 Theoretical Results Overview

In this work we present a new and improved secure search solution attaining the following key contributions.

5.1.1 Contribution 1: *Secure Search* with More Desirable Advantages

Presenting the first *Secure Search* solution that simultaneously attains all the desired advantages specified below:

1. *Full security:* All the data the server is exposed to is encrypted with leveled FHE achieving the strong property of semantic security both for data at rest and during searching.

2. *Efficient server:* The server evaluates a polynomial of degree $\log(n/\varepsilon) \cdot d$ and

with overall multiplications $n \cdot (\log(n/\varepsilon) + \mu + 1)$, for d, μ the degree and overall multiplications for IsMatch.

3. *Efficient client:* The client's running time is proportional to $|q|$ encryptions for the query and $\log(n)$ decryptions for the result (for queries of size $|q|$ bits).

4. *Efficient communication:* Single round protocol with low communication complexity proportional only to the size of the query and returned result.

5. *Unrestricted search functionality:* Our solution is applicable to any data array and query, with no restrictions on number of elements in array that match the query. Moreover, the protocol is modular and can be used with generic matching criteria IsMatch (for examples, see section 7).

6. *Retrieval of both index and element:* The client's output is both index and element. As opposed to retrieval of index alone or solely a binary indicator of whether the element exists in the array.

7. *Post-processing free client:* There is no need for post-processing on the client-side to obtain the final result.

8. *Negligible error probability:* The result returned to the client is correct with overwhelming probability.

This is in contrast to prior works where only a strict subset of these advantages was attained (see table 1.1).

In particular, our starting point was the recent work [1, 2, 3] presenting a secure search protocol called *SPiRiT* that achieved advantages 1-6 above together with either advantage 7 or 8 but not both (see table 1.1). Namely, the output returned by the server either requires post processing on the client's side, or has noticeable error probability. This is problematic, for example, when the *Secure Search* is utilized as a sub-component (possibly invoked repeatedly several times) in a larger computation performed by the server. For the reason that enabling client's post-processing necessitates another round of interaction for each sub-component to obtain its outcome.

Extra rounds of interaction are a major disadvantage in scenarios with intermittent, expensive or high latency communication.

In this work we improve over *SPiRiT* in achieving all advantages 1-8, thus resolving the aforementioned problem. An example use-case that benefits from the above is a server side update of an element that satisfies an issued query without further interaction with the client.

5.1.2 Contribution 2: Faster *Secure Search*

Our solution is faster than the prior secure search solutions on FHE encrypted data with full security, unrestricted search functionality and both index and element retrieval (i.e. *SPiRiT* Rand. and *SPiRiT* Det.). In particular, (1) we attain optimal client run-time, and (2) considerably improve the server's run-time: we reduce the degree of the evaluated polynomial from cubic to linear in $\log n$, and reduce the overall multiplications by up to $\log n$ factor.

In more details:

- First, when comparing solutions that support a post-processing free client with $\log(n)$ complexity on the client side (rows (ii) and (iii) in table 5.1), our solution improves the server's complexity by reducing the degree from cubic to linear in $\log n$, reducing the degree from linear to logarithmic in $1/\varepsilon$, and reduce the overall multiplications by a factor of $\log\log n$.

- Second, when comparing solutions that include additional post-processing done by the client resulting in a $\mathsf{poly}(\log(n))$ complexity on the client side (rows (i) and (v) in table 5.1), our solution improves the server's complexity by reducing the degree from cubic to linear in $\log n$, and reduce the overall multiplications by a factor of $(k/\alpha)\cdot\log(n) = \mathcal{O}\left(\frac{\log^3 n}{\log\log(n)\cdot\log(1/\varepsilon)}\right)$. We state however that whereas our solution is correct with overwhelming probability, *SPiRiT* Det. is correct with probability 1.

- In an extension of our solution for the case that $\mathsf{IsMatch}$ is the equality operator

(row (iv) in table 5.1) we show how to eliminate the dependence of the server's complexity on the parameters of the $\mathsf{IsMatch}$ polynomial d, μ (as appearing in prior works, rows (i)-(ii)) and achieve $\mathsf{poly}(\log(n/\varepsilon))$ degree and $\mathcal{O}(n \cdot \log(n/\varepsilon))$ overall multiplications.

Solution (iv) is especially appealing for scenarios with long binary stored elements and search queries (large files, DNA sequences, etc.). Specifically for strings of length w, *Binary Raffle* (row (iii)) achieves degree $\log(n/\varepsilon) \cdot w$ and overall multiplications $n \cdot (\log(n/\varepsilon) + w + 1))$. By employing Universal Hash technique (row (iv)), we completely get rid of this dependency on w while still preserving the efficiency on the server side (more details in section 7.1).

5.1.3 Contribution 3: Wider FHE Compatibility & Further Speedup

All our protocols require computing only over GF(2). This leads to the following two advantages:

First, our solution is compatible with all currently known candidate FHE schemes. Specifically, our solution can use as a black box any FHE scheme that enables homomorphic additions and multiplications over encrypted plaintext values in GF(2) (i.e. $\oplus, \wedge$ over plaintext bits). This is opposed to requiring homomorphic additions and multiplications over rings GF(p) for $p > 2$ as in *SPiRiT* (See section 1.3.8). Thus, for example, our solution is compatible with the GSW [16] FHE scheme, whereas *SPiRiT* is not.

Second, we achieve further run-time speedup. This is because (to the best of our knowledge) in all current FHE schemes implementations, including HElib [19] that implements BGV [5] scheme, homomorphic computations over GF(2) are considerably faster. The reason for that in BGV and HElib is the additional costs induced by ciphertext refresh after each multiplication in plaintext space GF(p) for $p > 2$ (see [23] and section 2.4 for further details).

	Server's Degree & Overall Multiplications	**Client's Decryptions**
(i) ***SPiRiT* Det.**	$\log^3(n) \cdot d$ $k \cdot n \cdot \left(\log^2(n) + \mu + 1\right)$	$k \cdot \log(n)$
(ii) ***SPiRiT* Rand.**	$\frac{c}{2\varepsilon} \log^3(n) \cdot d$ $n \cdot \left(\log\left(\frac{n}{\varepsilon} \cdot \frac{c}{2} \cdot \log n\right) + \mu + 1\right)$	$\log(n)$
(iii) ***Binary Raffle***	$\log(n/\varepsilon) \cdot d$ $n \cdot (\log(n/\varepsilon) + \mu + 1)$	$\log(n)$
(iv) ***Binary Raffle* + Universal Hash**	$2 \cdot \log^2(2n/\varepsilon)$ $n \cdot (3 \cdot \log(2n/\varepsilon) + 1)$	$\log(n)$
(v) ***Binary Raffle* + Client Probability Amplification**	$\log(3n) \cdot d$ $\alpha \cdot n \cdot (\log(3n) + \mu + 1)$ for $\alpha = \mathcal{O}(\log(1/\varepsilon))$	$\alpha \cdot \log(n)$

Table 5.1: Comparison of secure search protocols on FHE encrypted data that support full semantic security, unrestricted search functionality and both index and element retrieval. The metrics are: server's degree, server's overall multiplications and client's decryptions count to obtain the result index. Rows (i)-(ii) address previous works and rows (iii)-(v) address is this work.
Notations: d, μ – degree and overall multiplications of $\mathsf{IsMatch}$; n – array size; ε – failure probability; $k = \log^2 n / \log\log n$; c – constant that depends on the density of prime numbers;

5.2 Analysis of *Binary Raffle* Protocol

In this section we analyze the *Binary Raffle* protocol (see section 4.3.1). The *Binary Raffle* protocol, on client's inputs $(\mathsf{sk}, \mathsf{pk}, q)$ and server's inputs $(\varepsilon, \mathsf{ek}, \mathsf{IsMatch}, [\![x]\!])$ as specified in figure 4-3, satisfies the following two theorems (5.2.1, 5.2.2).

Theorem 5.2.1 (Correctness)**.** *With probability* $1-\varepsilon$*, the client's output is the binary representation* $b^* \in \{0,1\}^{\lceil \log_2 n \rceil + 1}$ *of the index of the first match* $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$*. The server has no output.*

Theorem 5.2.2 (Complexity)**.** *The client's running time consists of* $|q|$ *encryptions and* $|b^*| = \log(n)$ *decryptions. The server evaluates a polynomial of degree* $\log(n/\varepsilon) \cdot d_{\texttt{IsMatch}}$ *which performs* $n \cdot \log(n/\varepsilon) + n \cdot \mu_{\texttt{IsMatch}}$ *overall multiplications for* $d_{\texttt{IsMatch}}$ *and* $\mu_{\texttt{IsMatch}}$ *the degree and overall multiplications of* IsMatch*.*

Proof of Theorem 5.2.1 (Correctness). Once the client has encrypted its query and sent $[\![q]\!]$ to the server the first step (2.a) performed by the server is the evaluation of the specified chosen pattern matching polynomial IsMatch on stored array $[\![x]\!]$ and query $[\![q]\!]$. This results in an encrypted array $[\![ind]\!]$ that contains in every index $i \in [n]$ the encrypted boolean result $\mathsf{IsMatch}([\![x[i]]\!], [\![q]\!]) \in \{[\![0]\!], [\![1]\!]\}$.

Next, in step (2.b) the server converts the boolean array $[\![ind]\!]$ to a step function $[\![s]\!] = ([\![0]\!], \ldots, [\![0]\!], [\![1]\!], \ldots, [\![1]\!])$ where the first $[\![1]\!]$ is located in index i^* (or rather only encryptions of zeros if $[\![ind]\!]$ contains no matches).By theorem 5.3.1 with probability $1-\varepsilon$ the array $[\![s]\!]$ computed in protocol step (2.b) and returned after the algorithm call $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}([\![ind]\!])$ satisfies that it contains an encrypted step function as described above.

In the next step (2.c) the server computes the pairwise difference oa adjacent indices in $[\![s]\!]$ (its derivative) this results in an array $[\![s']\!]$ that contains $[\![0]\!]$ in every index except in the index of the first match i^* where it contains $[\![1]\!]$ (or in index $n+1$, is $s = (0, \ldots, 0)$).

In the final step performed by the server (2.d) it computes $[\![b^*]\!] = B \cdot [\![s']\!]$ for $B \in \{0,1\}^{(\lceil \log_2 n \rceil + 1) \times (n+1)}$ the matrix that contains in each column $k \in [n]$ the binary representation of k and in column $n+1$ the binary representation of 0. The array $[\![b^*]\!]$ will hold the index i^* of the single $[\![1]\!]$ in $[\![s']\!]$. This is because multiplying the matrix B by any vector of size $n+1$ that contains a single 1 bit in some index $j \in [n]$ results in a vector of size $\lceil \log_2 n \rceil$ that holds the binary representation of j (and a vector of zeros if $j = n+1$).

In this case the output computed in step (2.d) and sent to the client in step (2.e) is the encryption of $[\![b^*]\!]$. Because $b^* \in \{0,1\}^{\lceil \log_2 n \rceil + 1}$ is indeed the binary representation of the index of the first match $i^* = \min\{i \in [n] \mid \mathsf{IsMatch}(x[i], q) = 1\}$ decryption of $[\![b^*]\!]$ by the client guarantees correctness. □

Proof of Theorem 5.2.2 (Complexity). The only operations performed by the client are encrypting the query q and decrypting the result b^*.

The server's evaluated polynomial includes n executions of IsMatch (step 2.a), a single execution of $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ (step 2.b), another $n+1$ homomorphic additions for the pairwise differences (step 2.c), and at most $n\log(n)$ homomorphic additions for the calculation of the binary representation of the result index (step 2.d).

All the homomorphic multiplications come from the n executions of IsMatch and the single execution of $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$. Regarding degree, the first introduces a degree of $d_{\texttt{IsMatch}}$ and the second introduces a degree of $\log(n/\varepsilon)$ (see theorem 5.3.2). Regarding multiplications, the first introduces $n \cdot \mu_{\texttt{IsMatch}}$ multiplications (see theorem 5.3.2) and the second introduces additional $n \cdot \log(n/\varepsilon)$ multiplications. The total degree is the product of both degrees above. The overall multiplications are the sum of all multiplications above. □

5.3 Analysis of $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ Algorithm

In this section we analyze BinaryRaffleStepFunction, the key step in the *Binary Raffle* protocol (see section 4.3.2 and figures 4-4, 4-5).

Theorem 5.3.1 (Correctness)**.** *Let $v \in \{0,1\}^n$ be binary vector and $t = \big(t[1],\ldots,t[n]\big)$ be the vector returned after executing $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}(v)$. Then the following holds:*

1. *If $\forall i \in [n] : v[i] = 0$, then with probability 1 it holds that $\forall j \in [n] : t[j] = 0$*
2. *If $\exists i \in [n] : v[i] = 1$, then with probability $1-\varepsilon$ t it the step function*

   $t[1] = \ldots = t[i^*-1] = 0$ *and* $t[i^*] = \ldots = t[n] = 1$ *for* $i^* = \min\{i \in [n] \mid v[i] = 1\}$

Proof. The first case is trivial: if $v = (0,\ldots,0)$ then all the random partial prefix sums are zero (i.e., $\forall j \in [N(\varepsilon)], \forall k \in [n] : S[j,k] = 0$). This hold for all samples $r_1,\ldots,r_{N(\varepsilon)}$, so the resulting binary step function vector is $t[1] = \ldots = t[n] = 0$.

Next we analyze the second case, namely, when $\exists i \in [n] : v[i] = 1$, we denote $i^* = \min\{i \in [n] \mid v[i] = 1\}$. In Lemmas A.1-A.3 we prove the following:

1. For any $\eta \in [N(\varepsilon)]$, the probability of a random partial prefix sum in index $\ell \in [i^*, n]$ (the element $S[\eta, \ell]$ in the matrix) to be either zero or one is exactly half (see proof of lemma A.1).

2. For any index $\ell \in [i^*, n]$, the probability that $t[\ell]$ equals to zero is exactly $2^{-N(\varepsilon)}$ (see proof of lemma A.2).

3. The probability that there exists an index $\ell \in [i^*, n]$ so that $t[\ell] = 0$ is at most $n \cdot 2^{-N(\varepsilon)}$ (see proof of lemma A.3).

Assigning $N(\varepsilon) = \log_2(n/\varepsilon)$ in (3) above concludes the proof. □

In case we wish for the algorithm to succeed with an overwhelming probability we can employ the following corollary.

Corollary 5.3.1 (Negligible Error Probability)**.** *Given security parameter κ, by choosing $N(\varepsilon) = \log_2(n) + \omega(\log_2(\kappa))$ we get*

$$\Pr\left[\exists \ell \in [i^*, n] \text{ s.t. } t[\ell] = 0\right] = \mathsf{negl}(\kappa)$$

Proof. Immediate by using $2^{-\omega(\log_2(\kappa))} = \mathsf{negl}(\kappa)$ and assigning $N(\varepsilon) = \log_2(n) + \omega(\log_2(\kappa))$ in Lemma A.3. □

Suppose that we settled for a non-negligible error probability of $0 < \varepsilon_0 < 1/2$, we could use standard probability amplification technique to achieve negligible error probability by simply repeating the protocol 2α times and letting the client select the most frequent result (see [34] Lemma 10.5). This method gives birth to following corollary.

Corollary 5.3.2 (Probability Amplification)**.** *For any $\varepsilon > 0$, if we repeat α times the Binary Raffle protocol with error probability ε_0 (say $\varepsilon_0 = 1/3$) for $\alpha = \frac{-\log_2(1/\varepsilon)}{\log_2(4\varepsilon_0(1-\varepsilon_0))}$, and let the client select the result by taking the most frequent vote, then this result is correct with probability $1 - \varepsilon$.*

Proof. Using Lemma 10.5 from [34] which states that the error probability after selecting the most frequent vote of the α repetitions is at most ε. □

Next we analyze the complexity of BinaryRaffleStepFunction algorithm (figure 4-4).

Theorem 5.3.2 (Complexity)**.** *The* $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ *algorithm is realized by a polynomial of degree* $\log_2(n/\varepsilon)$ *which performs* $n \cdot \log_2(n/\varepsilon)$ *overall multiplications.*

Proof. The only multiplications performed in the algorithm are during the computation of OR between the elements in each column of the matrix S (step 3). Thus, the degree of the polynomial equals to on $N(\varepsilon) = \log_2(n/\varepsilon)$ (the length of each column in S), and the overall number of multiplications is $n \cdot N(\varepsilon)$ ($N(\varepsilon)$ multiplications for each of the cells of t).

□

Chapter 6

Experimental Setup and Results

In this section we describe in detail the benchmarks preformed to evaluate our *Binary Raffle* protocol and discuss our results. As a reference point, we executed benchmarks on an implementation of the *SPiRiT* protocol and evaluated both its deterministic and randomized variants.

6.1 Experimental Setup

We executed the protocols on top of the HElib C++ library [19] that was compiled with NTL [33] running over GMP [18]. 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 & 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.

We also took advantage of HElib's SIMD capabilities and "packed" multiple plaintext values (at least 500) into each ciphertext. We remark that we did not attempt to optimized 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 $n' = n \cdot \text{SIMD} \cdot \text{CORES}$ and the client obtained in the end of the experiment $\text{SIMD} \cdot \text{CORES}$ results.

6.2 Experiments Description

6.2.1 *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 section 2.3) as the selected 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 IsMatch polynomial on all elements (step (2.a) in figure 4-3) from the remaining steps of protocol. This can be thought as using an 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 (see Corollary 5.3.1), and any $\varepsilon > 2^{-1}$ as an error probability that allows standard probability amplification (see Corollary 5.3.2).

6.2.2 *SPiRiT*

As mentioned, we used an implementation of *SPiRiT* as a reference point. On array of sizes $n' = n \cdot \text{SIMD} \cdot \text{CORES}$, 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 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 $n' \approx 0.25 \cdot 10^6$ for $w = 16$ and $n' \approx 1 \cdot 10^6$ 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 (see section 1.3.8). 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}.

6.3 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.

6.3.1 *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 6-1 and 6-2).

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 for $w = 16$ and array of size $n' \approx 0.5 \cdot 10^6$ (appears as a dashed line in the graph) we were unable to complete the execution of the protocol on all the required primes for deterministic *SPiRiT* (7 top primes out of the total 36 are missing). This occurs due to the increase in required levels for larger primes in HElib and the penalty on RAM that is associated with it (see more details in section 2.4). Therefore, this value presented in the graph is actually an under-approximation for the total running time on all primes.

Figure 6-1: *Binary Raffle* With Negligible Error Probability ($\varepsilon = 2^{-80}$) Versus *SPiRiT* Deterministic for Word Width $w = 1$

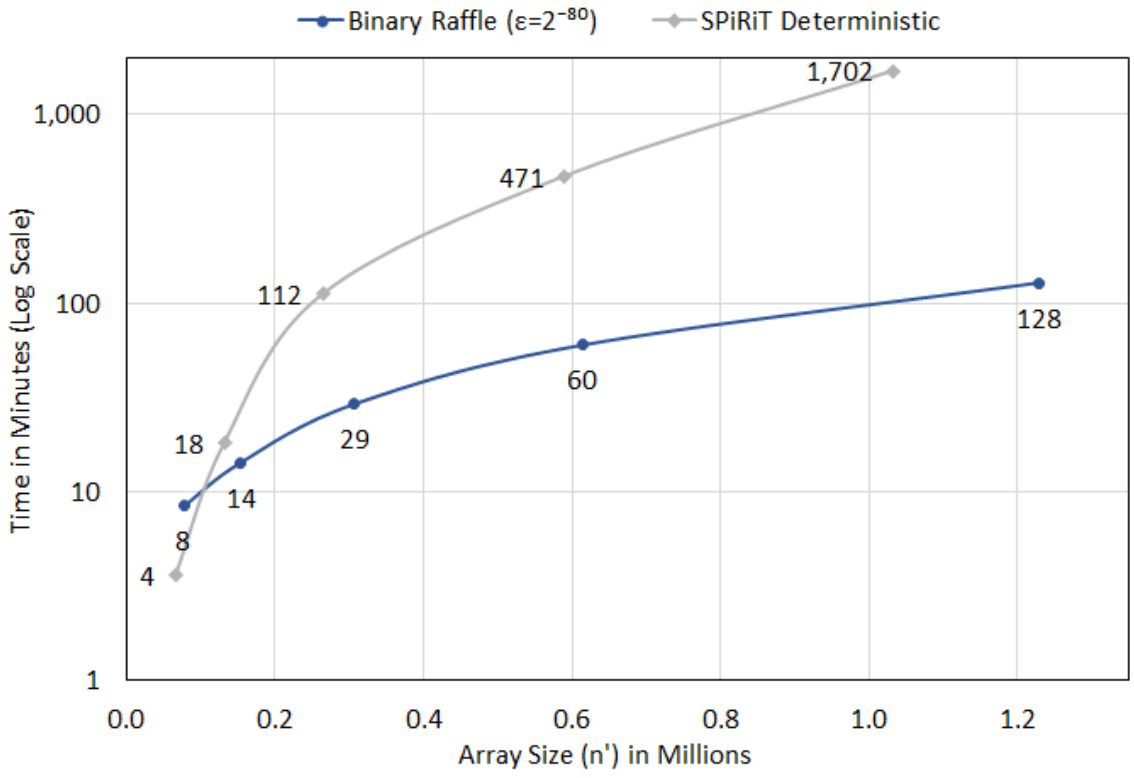
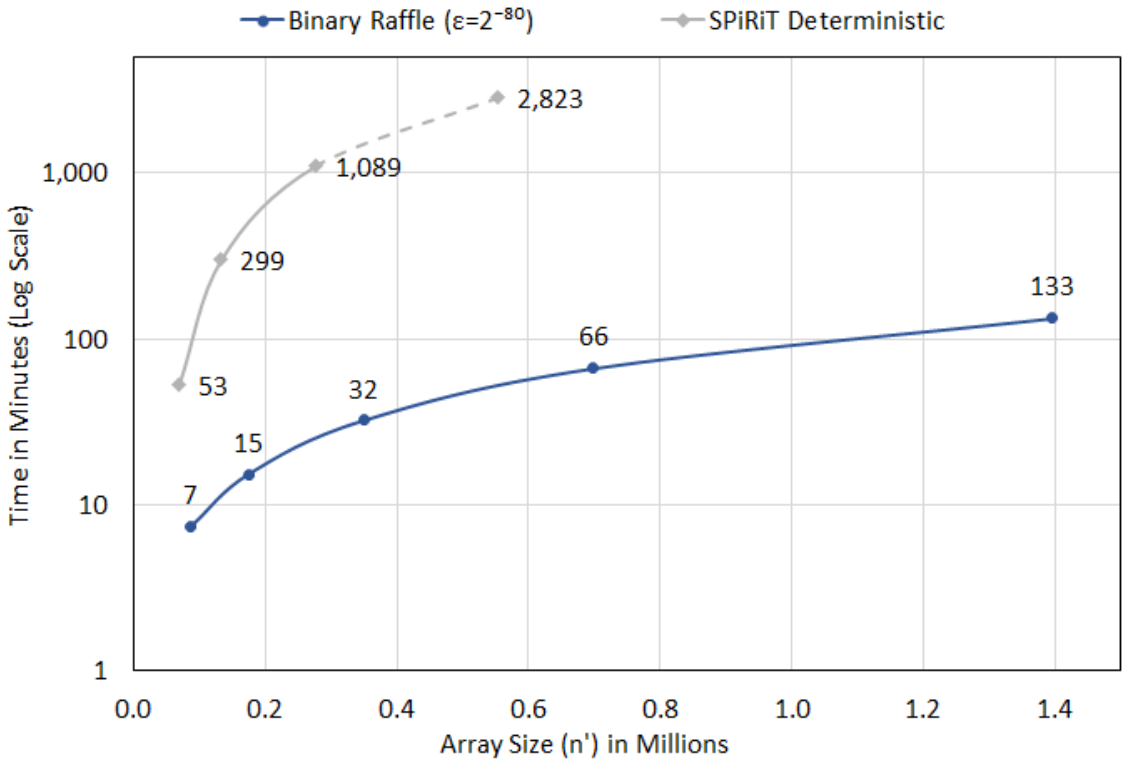


Figure 6-2: *Binary Raffle* With Negligible Error Probability ($\varepsilon = 2^{-80}$) Versus *SPiRiT* Deterministic for Word Width $w = 16$



6.3.2 *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 6-3 and 6-4).

Again, it can be seen in both graphs that *Binary Raffle* achieves faster execution time in an order of magnitude compared to *SPiRiT*. Another key detail that appears in both graphs is the inability to execute *SPiRiT* with all $2k$ primes for large enough array sizes (dashed lines in the graph). This happens, similarly to the described in previous section, because working with large primes in HElib increases RAM consumption (see more details in section 2.4). Therefore, these values presented in the graph are actually an under-approximation for the mean and standard deviation of the running time on all $2k$ primes.

Figure 6-3: *Binary Raffle* Versus *SPiRiT* Randomized Both With Error Probability $\varepsilon = 2^{-1}$ for Word Width $w = 1$

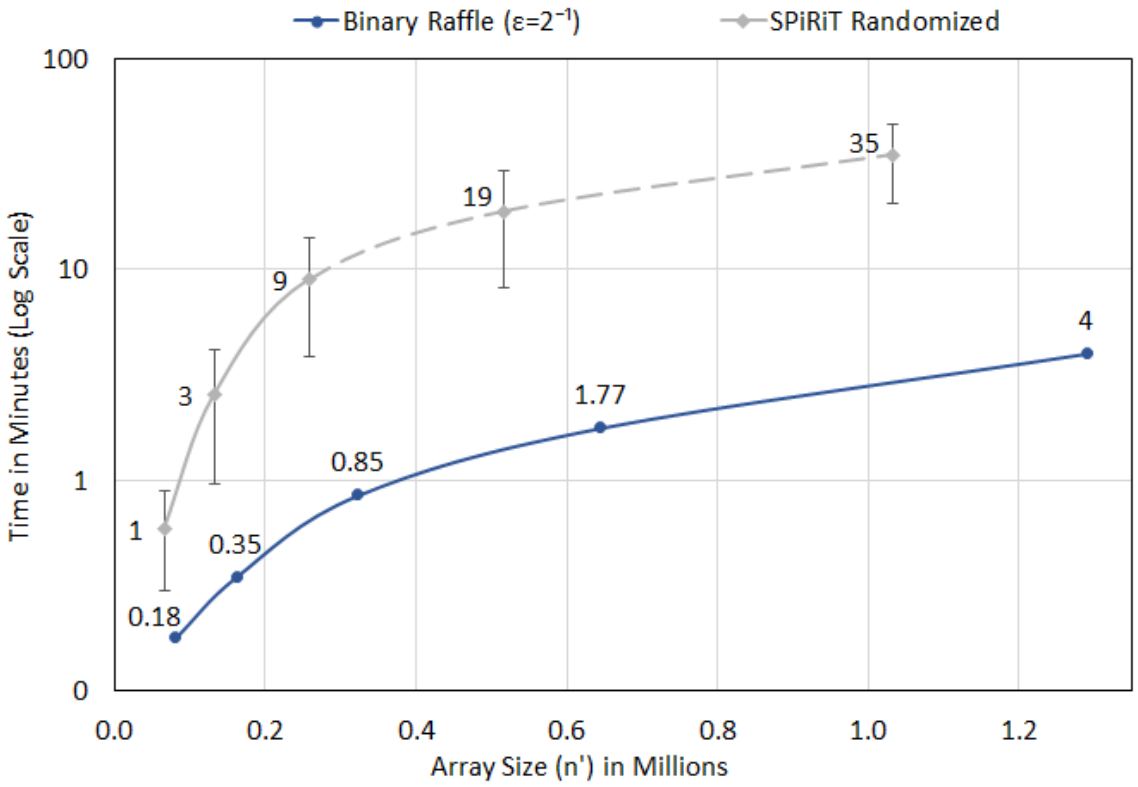
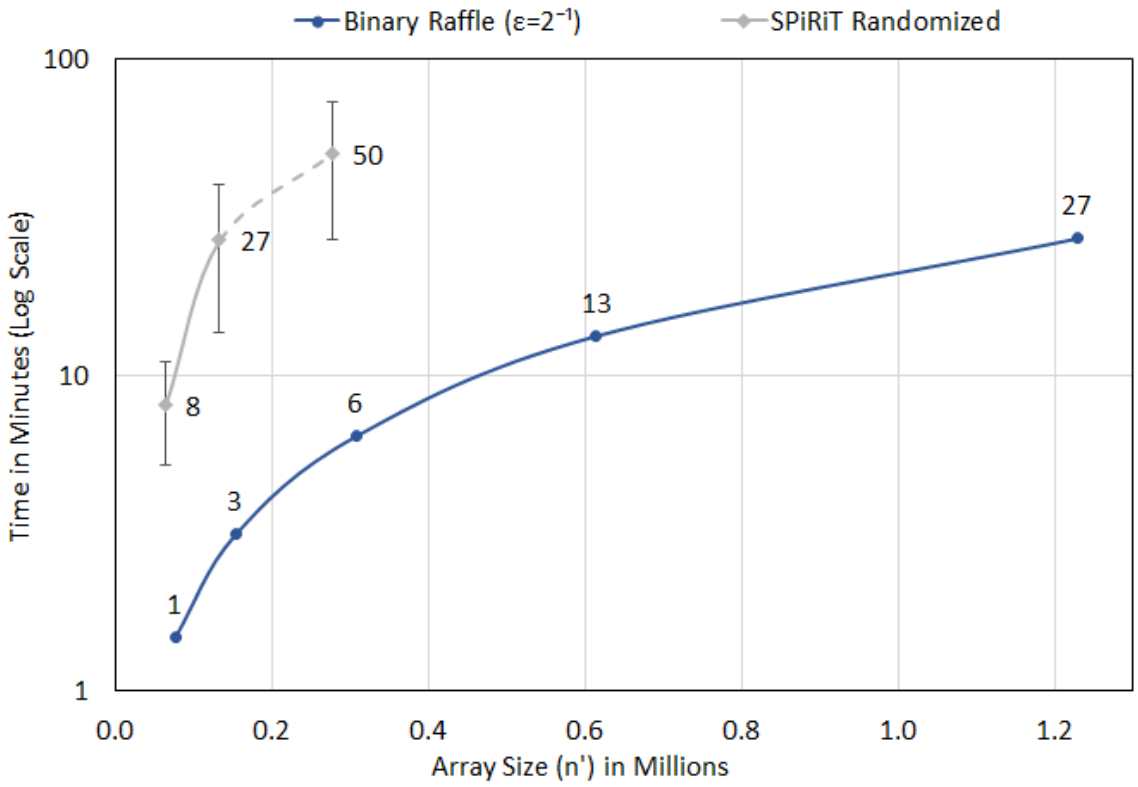


Figure 6-4: *Binary Raffle* Versus *SPiRiT* Randomized Both With Error Probability $\varepsilon = 2^{-1}$ for Word Width $w = 16$



6.3.3 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 6-5 and 6-6).

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.

Figure 6-5: *Binary Raffle* Comparing Different Failure Probabilities (ε) for Word Width $w = 1$

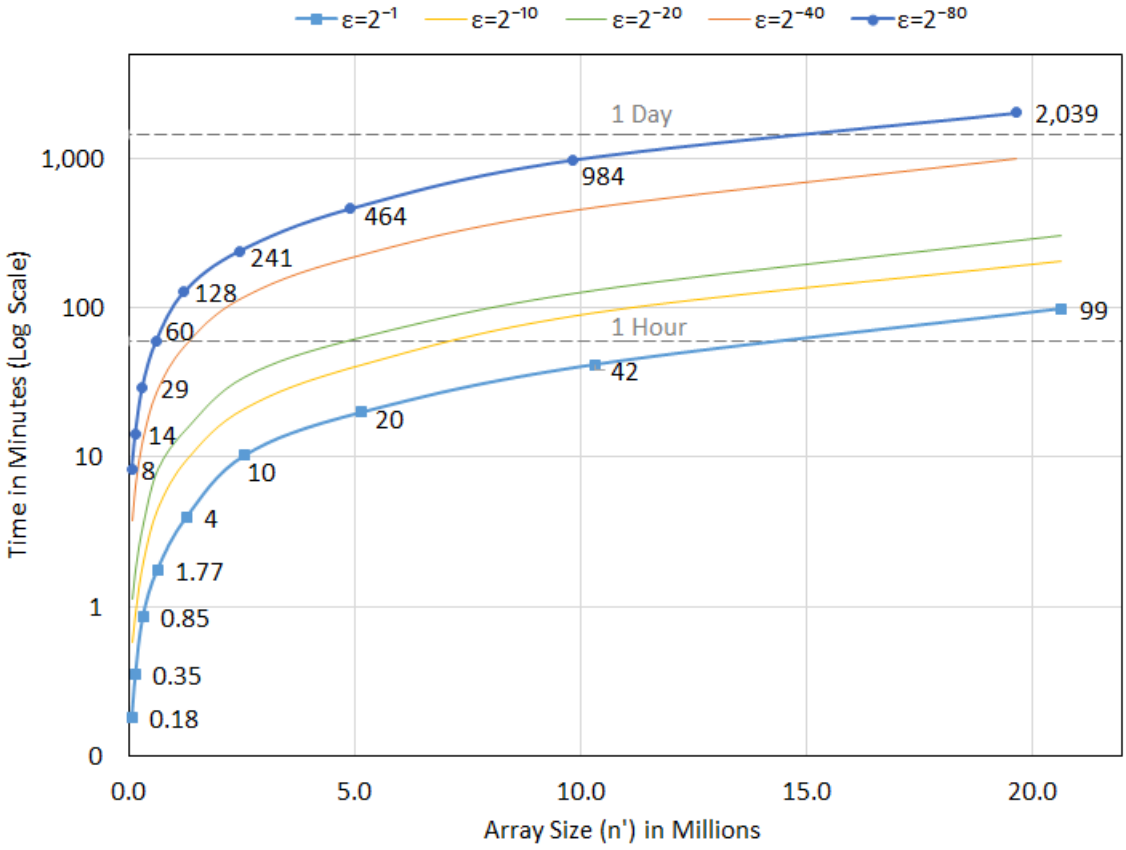
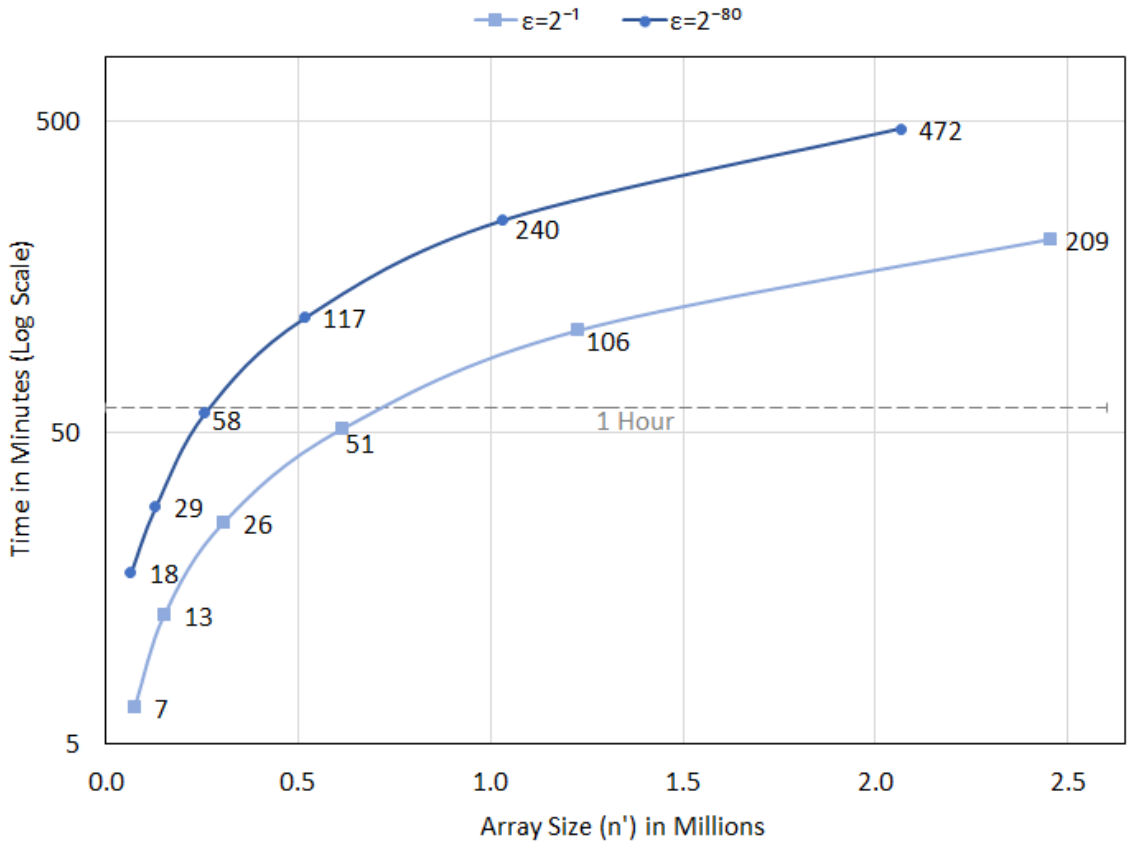


Figure 6-6: *Binary Raffle* Comparing Different Failure Probabilities (ε) for Word Width $w = 64$



6.3.4 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 6-7 and 6-8).

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 7.1.

Figure 6-7: *Binary Raffle* Comparing Different Word Sizes (w) for Error Probability $\varepsilon = 2^{-1}$

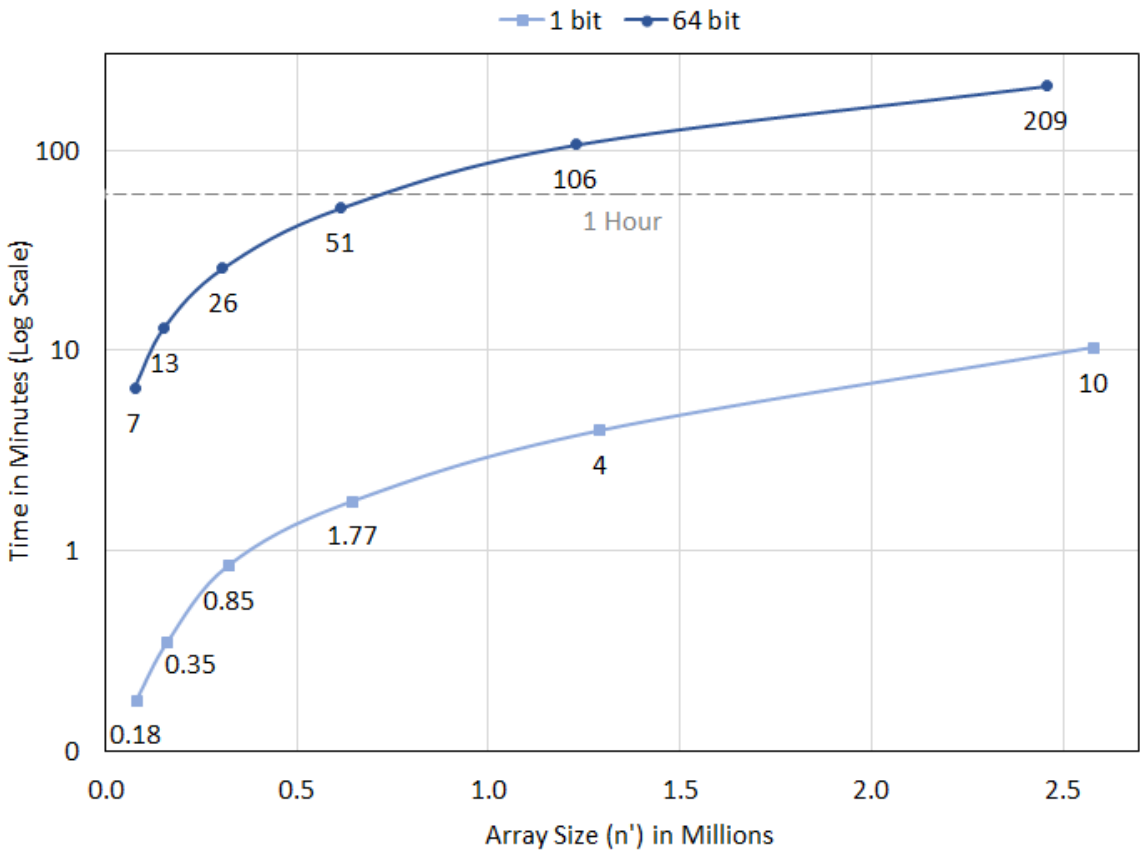
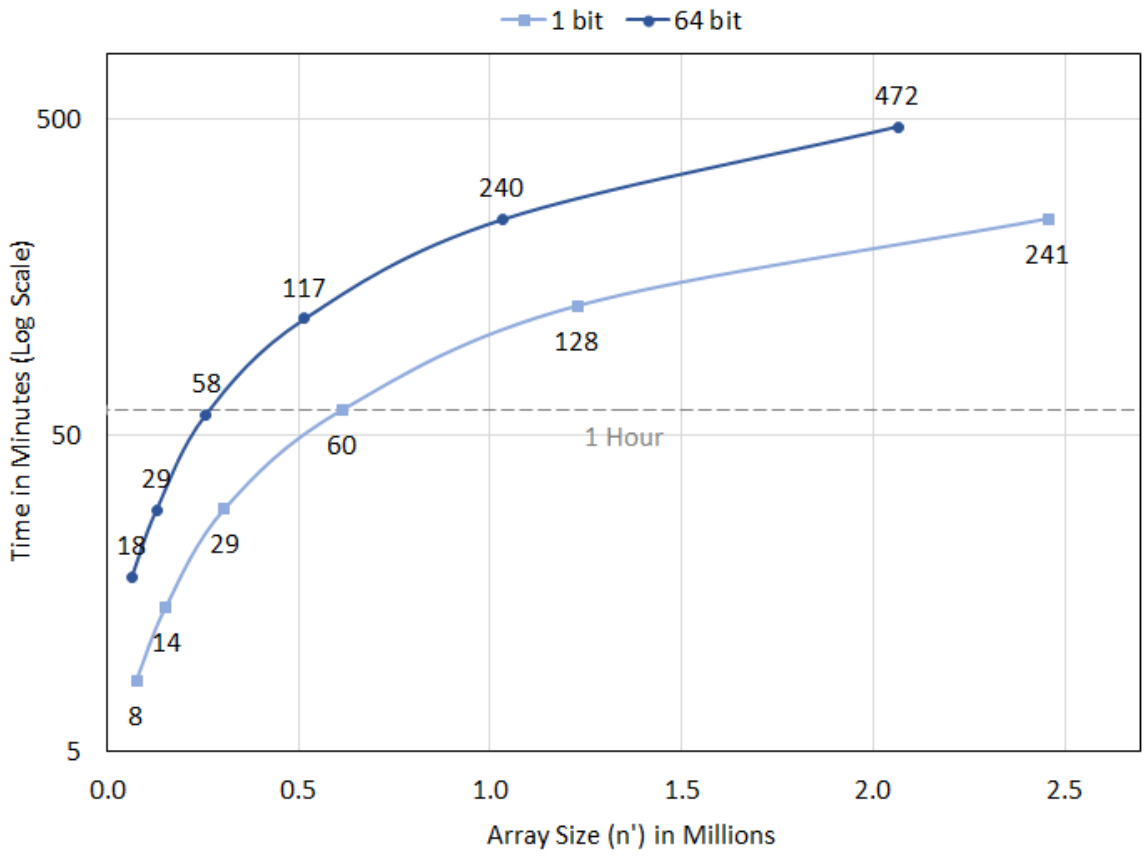


Figure 6-8: *Binary Raffle* Comparing Different Word Sizes (w) for Error Probability $\varepsilon = 2^{-80}$



Chapter 7

Make IsMatch Great Again!

In this section we show how specific implementations of the IsMatch predicate can improve run-time performance and introduce additional functionality to the *Binary Raffle* protocol. Specifically, faster IsMatch using Universal Hashing (section 7.1), augmenting IsMatch to search in a sub-array (section 7.2), and a version of IsMatch that can be used within a protocol that achieves sequential retrieval of additional matches (section 7.3) .

7.1 Faster IsMatch Using Universal Hashing

In our protocol (and also in *SPiRiT*) the first steps preformed by the server are n executions of the IsMatch pattern matching polynomial comparing each element in the stored array $[\![x[i]]\!]$ to the lookup value $[\![q]\!]$. This step is generic and can use a variety of domain specific IsMatch functions. Typically, there is at least a linear dependency[1] between the word length (w) of each element and the degree (also the overall multiplications) of the polynomial realizing the chosen IsMatch function.

For the case of equality operator (see section 2.3) we propose to reduce the degree and overall multiplications of IsMatch, making them independent of the word length w. This is achieved by applying a *Universal Hash Function* (see details in section

[1]Other use cases include sub-linear implementations for the IsMatch function that use probabilistic sampling. We do not address these implementations in our discussion in this section.

2.5) chosen at random to shrink the operands of the IsMatch predicate prior to its execution. To shrink these operands our chosen universal hash function will include solely homomorphic additions and its output length will be logarithmic in the number of elements n.

Parameters:

- Failure probability ε.
- Array size upper-bound n.
- Word size w and hashed word size $v = \lceil 2\log_2(n/\varepsilon) \rceil$.
- Description of the standard equality polynomial on binary vectors of length v $\mathsf{IsEqual}_v(x, y) \in \{0,1\} \in \{0,1\}$ (see section 2.3).

Initialization (occurs only once prior to first call):

1. Generate and store a random Toeplitz Matrix $A \in \{0,1\}^{w \times v}$ (see section 2.5) and a random vector $b \in \{0,1\}^v$.

Input:

- Elements $x_1, x_2 \in \{0,1\}^w$.

Output:

- with probability $1 - \varepsilon$, the output is 1 if $\mathsf{IsEqual}_w(x_1, x_2) = 1$ and 0 otherwise.

Algorithm:

1. Define the hash function $h(x) = Ax + b \pmod 2$.
2. Calculate $h(x_1), h(x_2) \in \{0,1\}^v$.
3. Return $\mathsf{IsEqualBinary}_v\big(h(x_1), h(x_2)\big)$.

Figure 7-1: $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}$ Algorithm

Theorem 7.1.1 (Correctness). *For array of n elements $x = \big(x[1], \ldots, x[n]\big)$ the following holds with probability $1-\varepsilon$: for all pairs $i, j \in [n]$,* $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}(x[i], x[j]) = \mathsf{IsEqualBinary}_w\big(x[i], x[j]\big)$.

Proof. Immediate by combining theorem 2.5.2 and corollary 2.5.2. The first theorem specifies the conditions required for a strongly universal family of hash functions to achieve an error probability ε given an array of at most n distinct values. The second corollary describes the use of Toeplitz Matrix to construct a strongly universal family of hash functions that is identical to the construction in the algorithm above. □

Theorem 7.1.2 (Complexity). *The* $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}$ *algorithm is realized by a polynomial of degree* $2\log_2(n/\varepsilon)$ *which performs* $2\log_2(n/\varepsilon)$ *overall multiplications.*

Proof. Notice that this algorithm needs to preform only homomorphic additions to calculate the hash of a value, so this step does not increases the degree or the amount of multiplications. All the multiplications the algorithm performs come from the invocation of exact equality for binary arrays of length $v = \lceil 2\log_2(n/\varepsilon) \rceil$. This step introduces polynomial of degree v and v overall multiplications. □

7.1.1 Applying UniversalHashIsEqual to *Binary Raffle* Protocol (4.3)

In cases when the bit length of array elements $x[i]$ and lookup value q are at least $2\log(n/\varepsilon)$ we can apply the $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}$ algorithm to speed up the running-time of the *Binary Raffle* protocol.

First the server will execute the initialization step fo algorithm $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}$ to generate and store in the clear (i.e unencrypted) a random Toeplitz Matrix $A \in \{0,1\}^{w \times v}$ (see section 2.5) and a random vector $b \in \{0,1\}^v$ using $w + 2v - 1$ bits for

$$v = \lceil 2\log_2(n/\varepsilon) \rceil$$

Now during each call to $\mathsf{IsMatch}(x[i], q)$ in the first step performed by the server (see figure 4-3) instead of using $\mathsf{IsEqualBinary}_w(x[i], q)$ use $\mathsf{UniversalHashIsEqual}_{w,n,\varepsilon}(x[i], q)$.

When an additional requirement is that the error probability will be negligible in the security parameter κ the value of v will be $v = 2\log_2(n) + \omega(\log_2(\kappa))$ (see corollary 2.5.1).

The correctness of *Binary Raffle* still stands as we merely introduced a concrete implementation for IsMatch. Regarding complexity, we present the theorem below.

Theorem 7.1.3 (Complexity)**.** *When executing the Binary Raffle protocol with* UniversalHashIsEqual *as the embedded* IsMatch *predicate the following holds:*

The client's running time consists of to w encryptions and $\log(n)$ decryptions.

For error probability ε, the server evaluates a polynomial of degree $2 \cdot \log_2{}^2(2n/\varepsilon)$ which performs $3 \cdot n \cdot \log_2(2n/\varepsilon)$ overall multiplications.

Proof. The proof will stem from combining the use of union bound ($\varepsilon' = \varepsilon/2$) and the statements below. We simply take the complexity formulas specified in theorem 5.2.2 and place the concrete complexity of UniversalHashIsEqual algorithm inside them: $d_{\texttt{IsMatch}} = 2\log_2(2n/\varepsilon)$, $\mu_{\texttt{IsMatch}} = 2\log_2(2n/\varepsilon)$. □

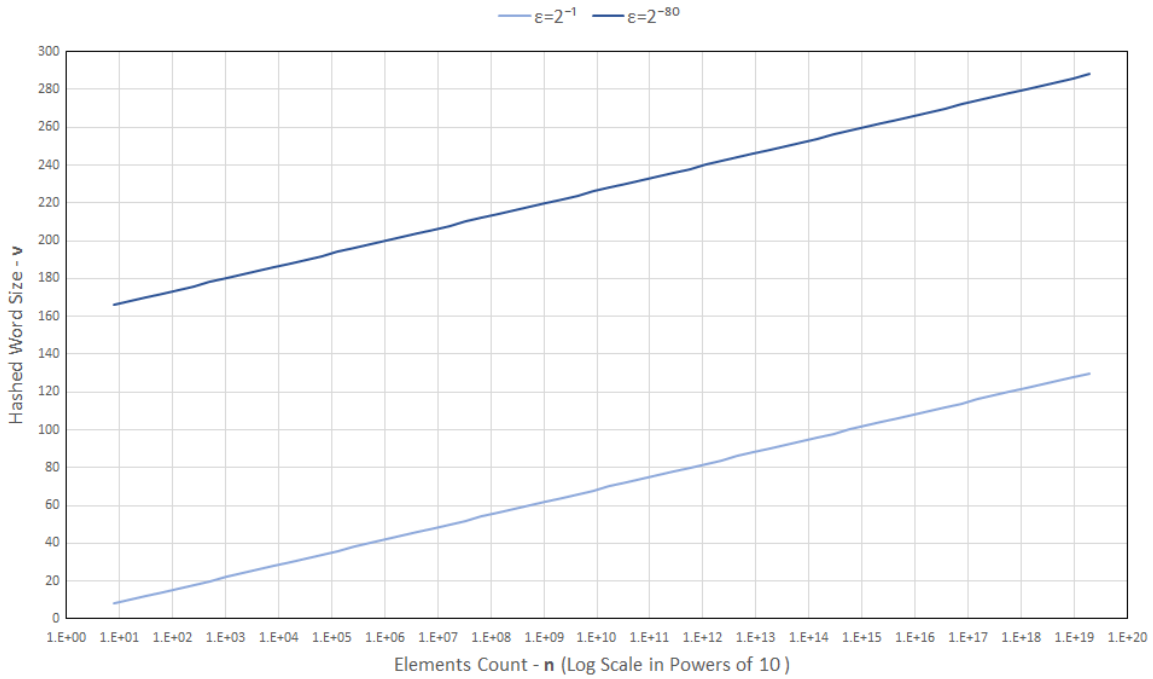


Figure 7-2: Required hashed word size $\left(v = \lceil 2\log_2(n/\varepsilon) \rceil\right)$ for different elements count (n) and error probabilities $\varepsilon \in \{2^{-1}, 2^{-80}\}$

7.2 Search In Sub-Array

A natural improvement to *Secure Search* is the ability to search for elements in a sub-array rather than in the whole array $[\![x]\!] = \big([\![x[1]]\!], \ldots, [\![x[n]]\!]\big)$. The client wishes to restrict the part of the array to search on to be between a lower-bound $l \in \{0,1\}^{\lceil \log_2(n) \rceil}$ and an upper-bound $u \in \{0,1\}^{\lceil \log_2(n+1) \rceil}$, namely indicating that only elements in indices $[l+1, \ldots, u-1]$ should be considered for a match. To do so the client encrypts the lower-bound l and upper-bound u and sends the augmented query $[\![q']\!] = \big([\![q]\!], [\![l]\!], [\![u]\!]\big)$ to the server.

At the server's side the above functionality is achieved by using an augmented implementation of $\mathsf{IsMatch}$ (step (2.a) in protocol 4-3). This implementation, $\mathsf{IsMatchInRange}_n$ (see also algorithm in figure 7-3) in addition to receiving encrypted lookup value $[\![q]\!]$ and the current array element $[\![x_i]\!]$ to compare it to, will also receive the lower bound $[\![l]\!]$ and an upper bound $[\![u]\!]$. As stated, these additional values indicate that a match should occur only if the original matching logic succeeds and also the index i is located within the boundaries $l < i < u$.

During the execution of $\mathsf{IsMatchInRange}_n\Big([\![x[i]]\!], \big([\![q]\!], [\![l]\!], [\![u]\!]\big)\Big)$, when element $[\![x[i]]\!]$ is examined for a match to $[\![q]\!]$, besides evaluation the original matching logic $c_0 = \mathsf{IsMatch}([\![x[i]]\!], [\![q]\!])$, an additional boundaries inspection is performed by invoking $c_1 = \mathsf{IsGreaterThen}_{\lceil \log_2(n) \rceil}(i, [\![l]\!])$ and $c_2 = \mathsf{IsGreaterThen}_{\lceil \log_2(n) \rceil}([\![u]\!], i)$ (see section 2.3). Finally, multiplying all three $c_0 \cdot c_1 \cdot c_2$ will give te required result.

Notice that when a client wishes to perform the search on the whole array of size n he simply sets $l = 0$ and $u = n + 1$.

Alternative versions for the above $\mathsf{IsMatchInRange}$ algorithm can include only a lower-bound or only an upper-bound restriction. These alternatives will perform only a single invocation of $\mathsf{IsGreaterThen}$ which reduces the degree and overall multiplications associated with the second invocation of $\mathsf{IsGreaterThen}$.

Parameters:

- Array size upper-bound n.
- Description of the required pattern matching polynomial $\mathsf{IsMatch}$.
- Description of "Greater-Then" operator $(x > y)$ on binary vectors of length w $\mathsf{IsGreaterThen}_w$ (see section 2.3).

Input:

- Array element $x[i] \in \{0,1\}^w$ and an index $i \in \{0,1\}^{\lceil \log_2(n) \rceil}$.
- The query/lookup value $q \in \{0,1\}^w$.
- Lower-bound $l \in \{0,1\}^{\lceil \log_2(n) \rceil}$ $(0 \leq l \leq n-1)$
- Upper-bound $u \in \{0,1\}^{\lceil \log_2(n+1) \rceil}$ $(l < u \leq n+1)$.

Output:

- The output is 1 if $\big(\mathsf{IsMatch}(x[i], q) = 1\big) \wedge (i > l) \wedge (u > i)$ and 0 otherwise.

Algorithm:

1. Calculate the matching criteria $c_0 = \mathsf{IsMatch}(x[i], q)$.
2. Calculate the lower-bound restriction $c_1 = \mathsf{IsGreaterThen}_{\lceil \log_2(n) \rceil}(i, l)$.
3. Calculate the upper-bound restriction $c_2 = \mathsf{IsGreaterThen}_{\lceil \log_2(n+1) \rceil}(u, i)$.
4. Return $c_0 \cdot c_1 \cdot c_2$.

Figure 7-3: $\mathsf{IsMatchInRange}_n$ Algorithm

The correctness of *Binary Raffle* still stands as we merely introduced a concrete implementation for $\mathsf{IsMatch}$. Regarding complexity, we present the theorem below.

Theorem 7.2.1 (Complexity)**.** *When executing the Binary Raffle protocol with* $\mathsf{IsMatchInRange}$ *as the embedded* $\mathsf{IsMatch}$ *predicate the following holds:*

The client's running time consists of $|q'| = |q|+|l|+|u| = |q|+2\log(n)$ *encryptions and* $|b^*| = \log(n)$ *decryptions.*

For error probability ε*, the server evaluates a polynomial of degree* $\log(n/\varepsilon) \cdot d_{\texttt{IsMatch}} \cdot \log^2(n)$ *which performs* $n \cdot \log(n/\varepsilon) + n \cdot \mu_{\texttt{IsMatch}} + 2n \cdot \mathcal{O}\left(\log^2(n)\right)$ *overall multiplications for* $d_{\texttt{IsMatch}}$ *and* $\mu_{\texttt{IsMatch}}$ *the degree and overall multiplications of* IsMatch.

Proof. When comparing to theorem 5.2.2 the additional operations performed by the client are encrypting the lower-bound and upper-bound each of size $\log(n)$.

Regarding the server, the additional operations are two invocations of IsGreaterThen (see section 2.3) and multiplication of the results of these invocations with the result of IsMatch. These additional operation elevate the degree of the polynomial evaluated by the server by a factor of $\log^2(n)$, and introduce additional $2n \cdot \mathcal{O}\left(\log^2(n)\right)$ overall multiplications. □

7.3 Sequential Retrieval of Additional Matches ("Fetch-Next")

Until now we presented our *Secure Search* solution as being able to search and retrieve only the first match. Now we present a simple extension that will allow retrieval of additional matches to a given query q.

The protocol will be initialized with the $\mathsf{IsMatchInRange}_n$ algorithm (see figure 7-3) variant that include only a lower-bound restriction[2] as selected IsMatch predicate.

Every time the client issues a "fresh" query he will simply initialize $l = 0$. Otherwise, suppose the client already issued a query q and received a response indicating that the first match is located in index i_1. In the next round, to fetch the next match, the client will initialize $l = i_1$ which will cause the search to be performed on indices

[2]We do not need the upper-bound restriction so we do not include it to save the associated degree and multiplications.

$[i_1 + 1, \ldots, n]$ (without revealing it to the server). This routine can go on until the client receives the response that indicates that there are no more matches.

Notice that the server cannot distinguished between a "fresh" query and a "fetch-next" query as in both cases the index l and query q are encrypted. This guarantees that the amount of elements that match any given query q is not leaked to the sever.

Chapter 8

Conclusions

In this work we presented a new and improved solution for secure search on FHE encrypted data. Our solution is the first to simultaneously achieve a slew of desired properties (full security, efficient server client and communication, unrestricted search functionality, retrieval of both index and element, post-processing free client, and negligible error probability). Furthermore, our solution is faster than the prior state-of-the-art, achieving optimal client complexity, and improving server's complexity by reducing the degree of the polynomial it evaluates from cubic to linear in $\log n$ (for n the number of searched data elements), and reducing the overall number of multiplications by a factor of up to $\log n$. Moreover, our solution is compatible with a wider range of FHE candidates. We implemented our secure search protocol and performed extensive run-time benchmarks showing concrete running time speedup by an order of magnitude.

Bibliography

[1] Adi Akavia, Dan Feldman, and Hayim Shaul. "Secure Search on the Cloud via Coresets and Sketches". In: *arXiv preprint arXiv:1708.05811* (2017).

[2] Adi Akavia, Dan Feldman, and Hayim Shaul. "Secure Search via Multi-Ring Fully Homomorphic Encryption." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 245.

[3] 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*. ACM. 2018.

[4] Christoph Bösch et al. "A survey of provably secure searchable encryption". In: *ACM Computing Surveys (CSUR)* 47.2 (2015), p. 18.

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

[6] 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*. IEEE Computer Society. 2011, pp. 97–106.

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

[8] Gizem S Çetin et al. "Blind Web Search: How far are we from a privacy preserving search engine?" In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 801.

[9] 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*. ACM. 2017, pp. 1243–1255.

[10] Jung Hee Cheon, Miran Kim, and Myungsun Kim. "Optimized search-and-compute circuits and their application to query evaluation on encrypted data". In: *IEEE Transactions on Information Forensics and Security* 11.1 (2016), pp. 188–199.

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

[12] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. "Bandwidth efficient PIR from NTRU". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 195–207.

[13] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptology ePrint Archive* 2012 (2012), p. 144.

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

[15] Craig Gentry, Shai Halevi, and Nigel P Smart. "Homomorphic evaluation of the AES circuit". In: *Advances in cryptology–crypto 2012*. Springer, 2012, pp. 850–867.

[16] 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*. Springer, 2013, pp. 75–92.

[17] 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*. ACM. 1987, pp. 218–229.

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

[19] S Halevi and V Shoup. *The HElib library*. 2015.

[20] Shai Halevi. "Homomorphic encryption". In: *Tutorials on the Foundations of Cryptography*. Springer, 2017, pp. 219–276.

[21] Shai Halevi and Victor Shoup. "Algorithms in helib". In: *International cryptology conference*. Springer. 2014, pp. 554–571.

[22] Shai Halevi and Victor Shoup. "Bootstrapping for helib". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 641–670.

[23] Shai Halevi and Victor Shoup. "Design and implementation of a homomorphic-encryption library". In: *IBM Research (Manuscript)* 6 (2013), pp. 12–15.

[24] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. "SHIELD: scalable homomorphic implementation of encrypted data-classifiers". In: *IEEE Transactions on Computers* 65.9 (2016), pp. 2848–2858.

[25] Myungsun Kim et al. "Better Security for Queries on Encrypted Databases." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 470.

[26] Myungsun Kim et al. "Private Compound Wildcard Queries using Fully Homomorphic Encryption". In: *IEEE Transactions on Dependable and Secure Computing* (2017).

[27] Hugo Krawczyk. "LFSR-based hashing and authentication". In: *Annual International Cryptology Conference*. Springer. 1994, pp. 129–139.

[28] 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*. Springer. 2014, pp. 3–27.

[29] Kristin E Lauter. "Practical applications of homomorphic encryption". In: *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*. ACM. 2012, pp. 57–58.

[30] 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*. ACM. 2012, pp. 1219–1234.

[31] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[32] Sujoy Sinha Roy et al. "Hardware assisted fully homomorphic function evaluation and encrypted search". In: *IEEE Transactions on Computers* 66.9 (2017), pp. 1562–1572.

[33] Victor Shoup. "NTL: A library for doing number theory, 10.5.0". In: *http://www.shoup. net/ntl/* (2017).

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

[35] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, codes and cryptography* 71.1 (2014), pp. 57–81.

[36] 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*. IEEE. 2000, pp. 44–55.

[37] Haixu Tang et al. "Protecting genomic data analytics in the cloud: state of the art and opportunities". In: *BMC medical genomics* 9.1 (2016), p. 63.

[38] David P Woodruff et al. "Sketching as a tool for numerical linear algebra". In: *Foundations and Trends® in Theoretical Computer Science* 10.1–2 (2014), pp. 1–157.

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

[40] Masaya Yasuda et al. “Secure pattern matching using somewhat homomorphic encryption”. In: *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*. ACM. 2013, pp. 65–76.

Appendix A

$\mathsf{BinaryRaffleStepFunction}_{\eta,\varepsilon}$ Correctness Lemmas

In this appendix we provide lemmas that are used in the correctness proof of $\mathsf{BinaryRaffleStepFunction}_{n,\varepsilon}$ algorithm (theorem 5.3.1).

Lemma A.1 (Single Sample + Single Index)**.** *Let $\eta \in [N(\varepsilon)]$, $\ell \in [i^*, n]$ be some indices and $S[\eta, \ell]$ a random partial prefix sum at row η and up to column ℓ. It holds that $\Pr[S[\eta, \ell] = 1] = \frac{1}{2}$ (and by the complement rule of probability $\Pr[S[\eta, \ell] = 0] = \frac{1}{2}$).*

Proof. The proof is by induction on $\mathsf{weight}(v)$.

Base case:
For $\mathsf{weight}(v) = 1$. It holds that $S[\eta, \ell] = 1$ only when $r_\eta[i^*] = 1$. Because r_η is sampled at random from the uniform distribution over $\{0,1\}^n$ we get that $\Pr\left[r_\eta[i^*] = 1\right] = \frac{1}{2}$ and thus $\Pr\left[S[\eta, \ell] = 1\right] = \frac{1}{2}$.

Hypothesis:
For $\mathsf{weight}(v) = c - 1$ it holds that $\Pr\left[S[\eta, \ell] = 1\right] = \frac{1}{2}$.

Inductive step:
For $\mathsf{weight}(v) = c$ any random partial prefix sum $S[\eta, \ell]$ can be divided into two

partial sums $S[\eta, \ell] = s' + s'' =$

$$\langle \mathsf{prefix}_{i^*}(v), \mathsf{prefix}_{i^*}(r_\eta) \rangle + \langle \mathsf{suffix}_{i^*}(v), \mathsf{suffix}_{i^*}(r_\eta) \rangle \pmod 2$$

From the induction basis we get $\Pr\Big[s' = 1\Big] = \frac{1}{2}$ and from the induction hypothesis we get $\Pr\Big[s'' = 1\Big] = \frac{1}{2}$. From the complement rule of probability we get $\Pr\Big[s' = 0\Big] = \frac{1}{2}$ and $\Pr\Big[s'' = 0\Big] = \frac{1}{2}$. The sum $S[\eta, \ell]$ equals to 1 if either one of the following two cases hold: (1) $s' = 1$ and $s'' = 0$ or (2) $s' = 0$ or $s'' = 1$ and thus we get that $\Pr\Big[S[\eta, \ell] = 1\Big] = \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$.

□

Lemma A.2 (Multiple Samples + Single Index)**.** *Let $\ell \in [i^*, n]$ be some index and $t[\ell]$ the binary step function vector at this index. It holds that* $\Pr\Big[t[\ell] = 0\Big] = 2^{-N(\varepsilon)}$.

Proof. Lets choose a single matrix row $\eta \in [N(\varepsilon)]$ and define

$$t'[\ell] = 1 - \big(1 - S[\eta, \ell]\big) \pmod 2$$

Observe that $t'[\ell]$ is similar to $t[\ell]$ with the only difference that we include only a single cell of the matrix S ($S[\eta, \ell]$) in the calculation of $t'[\ell]$ rather than the full product of ℓ-th column $S[1, \ell], \ldots, S[N(\varepsilon), \ell]$ as in the calculation of $t[\ell]$. A direct property of this definition is the following equivalence

$$\Big(t'[\ell] = 0\Big) \leftrightarrow \Big(S[\eta, \ell] = 0\Big)$$

From Lemma A.1 we know that for a single sample r_η it holds that $\Pr\Big[S[\eta, \ell] = 0\Big] = \frac{1}{2}$ and from the equivalence above we can conclude that $\Pr\Big[t'[\ell] = 0\Big] = \frac{1}{2}$.

In a similar manner for $N(\varepsilon)$ samples we get

$$\Big(t[\ell] = 0\Big) \leftrightarrow \forall j \in [N(\varepsilon)] : \Big(S[j, \ell] = 0\Big) \tag{A.1}$$

Because $r_1, \ldots, r_{N(\varepsilon)}$ are sampled independently we get that $\forall j \in [N(\varepsilon)] : \Big(S[j, \ell] =$

$0\Big)$ are mutually independent events and thus

$$\Pr\Big[\bigcap_{j=1}^{N(\varepsilon)}\Big(S[j,\ell]=0\Big)\Big]=\prod_{j=1}^{N(\varepsilon)}\Pr\Big[S[j,\ell]=0\Big] \tag{A.2}$$

Finally, again from Lemma A.1 we get that

$$\prod_{j=1}^{N(\varepsilon)}\Pr\Big[S[j,\ell]=0\Big]=2^{-N(\varepsilon)} \tag{A.3}$$

and by combining the above statements (1), (2) and (3) we get $\Pr\Big[t[\ell]=0\Big]=2^{-N(\varepsilon)}$ as required. □

Lemma A.3 (Multiple Samples + Multiple Indices)**.** *It holds that*

$$\Pr\Big[\exists\ell\in[i^*,n]\text{ s.t. }t[\ell]=0\Big]\leq n\cdot 2^{-N(\varepsilon)}$$

Proof.

$$\begin{aligned}\Pr\Big[\exists\ell\in[i^*,n]\text{ s.t. }t[\ell]=0\Big]\leq\\ \sum_{\ell=i^*}^{n}\Pr\Big[t[\ell]=0\Big]\leq\\ \sum_{\ell=1}^{n}\Pr\Big[t[\ell]=0\Big]\leq n\cdot 2^{-N(\varepsilon)}\end{aligned}$$

The first inequality is from union-bound on all the indices $\ell\in[i^*,n]$. The second inequality is simply because we are adding additional non-negative probabilities to the sum. The last inequality is due to Lemma A.2. □