Srinivas Vivek, Shyam Murthy*, and Deepak Kumaraswamy

Integer Polynomial Recovery from Outputs and its Application to Cryptanalysis of a Protocol for Secure Sorting[†]

Abstract: We investigate the problem of recovering integer inputs (up to an affine scaling) when given only the integer monotonic polynomial outputs. Given n integer outputs of a degree-d integer monotonic polynomial whose coefficients and inputs are integers within known bounds and $n \gg d$, we give an algorithm to recover the polynomial and the integer inputs (up to an affine scaling). A heuristic expected time complexity analysis of our method shows that it is exponential in the size of the degree of the polynomial but polynomial in the size of the polynomial coefficients. We conduct experiments with real-world data as well as randomly chosen parameters and demonstrate the effectiveness of our algorithm over a wide range of parameters.
Using only the polynomial evaluations at specific integer points, the apparent hardness of recovering the input data served as the basis of security of a recent protocol proposed by Kesarwani et al. for secure k-nearest neighbour computation on encrypted data that involved secure sorting. The protocol uses the outputs of randomly chosen monotonic integer polynomial to hide its inputs except to only reveal the ordering of input data. Using our integer polynomial recovery algorithm, we show that we can recover the polynomial and the inputs within a few seconds, thereby demonstrating an attack on the protocol of Kesarwani et al.

Keywords: Polynomial Reconstruction, Somewhat Homomorphic Encryption, Sorting, Low-depth Circuit

† The final published version of this paper appears in the Journal of Mathematical Cryptology, Volume 16 Issue 1, with DOI : 10.1515/jmc-2021-0054.
An earlier version of this work was titled "Cryptanalysis of a Protocol for Efficient Sorting on SHE Encrypted Data", and appeared in the Proceedings of 17th IMACC, 2019. The current work subsumes the earlier work and provides new results.

Srinivas Vivek, Shyam Murthy, IIIT Bangalore, Electronics City, Hosur Road, Bangalore, 560100, Karnataka, India; Email: srinivas.vivek@iiitb.ac.in; Email: shyam.sm@iiitb.ac.in
Deepak Kumaraswamy, National Institute of Technology, Surathkal, Karnataka, India; Email: deepakkumaraswamy99@gmail.com

1 Introduction

The problem of polynomial reconstruction, posed in different flavours, has received good attention in the past [21, 32]. Guruswami and Sudan [21] present a list decoding algorithm for Reed-Solomon codes which is closely related to polynomial reconstruction over a field, Naor and Pinkas [32] give a new cryptographic primitive based on intractibility assumptions related to noisy polynomial reconstruction problem. Given n points (i.e., input/output pairs) on a d-degree polynomial and $n > d$, a well-known technique is the Lagrange interpolation that uses $(d+1)$ points to output a unique polynomial of degree at most d that fits the points. This problem also occurs in the context of decoding other error-correcting codes in general, with many proposed techniques to recover polynomials even when a sufficiently small fraction of the input-output pairs are error prone [2, 19].

Section 2 discusses some of the work available on the problem of polynomial reconstruction. We would like to emphasize that, to the best of our knowledge, in all the previous works both the input to and the output of the polynomials are given. In the present setting, we look at the study of recovery of monotonic polynomials with non-negative integer coefficients (sampled from a certain interval) when *only* the integer outputs are provided and we are *not* provided the inputs (except that we only know they are integers within a given range and the degree of polynomial). The goal is to recover the inputs. The motivation comes from the field of cryptanalysis in a setting that is used to perform computation on encrypted data, where the computation involves homomorphic evaluation of a monotonic integer polynomial at specific points such that the ordering of these points are preserved to enable sorting. It is easy to see that in the case of polynomials over the real or the complex numbers, there will typically be infinitely many satisfying polynomials to the above mentioned polynomial recovery problem. Hence, it seems quite intuitive to expect the same for integer polynomials and this apparent hardness of recovering the original polynomial has found applications in areas such as privacy-preserving machine learning to hide the input data [23]. Formally, the problem is stated as given below.
Let $\mathbb{N}$ be the set of natural numbers (including 0), $\mathbb{Z}$ be the set of integers, $\mathbb{Z}^+$ be the set of positive integers and $\mathbb{Q}$ be the set of rational numbers.

Problem Statement 1. Polynomial reconstruction given only outputs:
Let $p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d \in \mathbb{N}[x]$ *(deg*$(p(x)) = d$ *and* $p(x)$ *is monotonic). The coefficients* a_i*,* $0 \leq i \leq d$*, are independent and uniformly random in the range* $[1, 2^\alpha - 1]$*,* $\alpha \in \mathbb{N}$*.*
Given **only** n *integer outputs* $y_1 = p(x_1), y_2 = p(x_2), \ldots, y_n = p(x_n)$ *evaluated at* n

distinct values with $n \gg d$, recover x_i, or its affine shifts, on which $p(x)$ is evaluated, $x_i \in \mathbb{N}$, $x_i \in [0, 2^\beta - 1]$ and $\beta \in \mathbb{N}$. The values of α, β and d are known.

The paper is organized as follows. Section 2 discusses some of the works related to polynomial reconstruction. Section 3 introduces the problem setting in detail with the help of a motivating application followed by contributions of the paper. Section 4 describes our algorithm for polynomial recovery. Section 5 discusses the experiments and results obtained. Section 6 gives a variation of the original protocol and a possible attack, followed by a section on conclusion and future work.

2 Related Work on Polynomial Reconstruction

Polynomial reconstruction has applications in coding theory in the context of decoding received codewords. A block error-correcting code $\mathcal{C}$ is a collection of strings called codewords, all of which have the same length, over some finite alphabet Σ [21]. It is defined such that any pair of codewords in the range of $\mathcal{C}$ differ in at least d locations out of, block length, n. The decoding problem for such an error-correcting code is the problem of finding a codeword in Σ^n which is at a distance at most e (the error bound) from a received codeword $R \in \Sigma^n$.
In the Reed-Solomon code, the alphabet Σ is a finite field $\mathbb{F}$, the message specifies a univariate polynomial of degree-d. The codewords are evaluations of the polynomial at n distinct points chosen from $\mathbb{F}$ [39].

Definition 1. Polynomial reconstruction [21, pp. 3]: *Given integers k, t and n points (x_i, y_i), $1 \leq i \leq n$, where $x_i, y_i \in \mathbb{F}$, output all univariate polynomials p of degree at most k such that $y_i = p(x_i)$, for at least t values of $i \in [1, n]$.*

For Reed-Solomon family of codes, the decoding problem reduces to the polynomial reconstruction problem [21]. List decoding formalizes the notion of error-correction.

Definition 2. List decoding problem for a code $\mathcal{C}$ [39, pp. 3]: *Given a received word $r \in \Sigma^n$ and error bound e, output a list of all codewords $c_1, \ldots, c_m \in \mathcal{C}$ that differ from r in at most e places.*

Hence, the list decoding problem for (generalized) Reed-Solomon code is a variant of the polynomial reconstruction problem over a field $\mathbb{F}$.

Definition 3. List decoding for Reed-Solomon Codes [39, pp. 13]: *Given integers e, k, d and n pairs $(x_1, r_1), \ldots, (x_n, r_n)$, find all degree-$(k-1)$ polynomials p such that $p(x_i) = r_i$ for at least $n - e$ values of i.*

Sudan [39] gives an algorithm that works for $t \geq \sqrt{2dn}$, whereas, the classical algorithm of Berlekamp and Massey [2] solves for $t \geq \frac{n+d}{2}$. Guruswami and Sudan [21] give a solution for $t \geq \sqrt{dn}$ for the same problem.

Gopalan *et al* [20] study the problem of polynomial reconstruction for low-degree multivariate polynomials over $\mathbb{F}[2]$, where, given a set of points $x \in \{0,1\}^n$ and their evaluations $f(x)$ at each of those points, the goal is to find a degree d polynomial that has good agreement with f at $1-\epsilon$ fraction of the points. They show that it is NP-hard to find a polynomial that agrees with f on more than $1 - 2^{-d} + \delta + \epsilon$ fractions of points for any $\epsilon, \delta > 0$.

Naor and Pinkas [32] describe the noisy polynomial reconstruction problem, the description of which closely resembles that of the list decoding problem of Reed-Solomon codes since the two problems are closely related.

Definition 4. Noisy polynomial reconstruction [32, pp. 8]: *Given integers k and t, and n points (x_i, y_i), $1 \leq i \leq n$, where $x_i, y_i \in \mathbb{F}$, output any univariate polynomial p of degree at most k such that $p(x_i) = y_i$ for at least t values $i \in [1, n]$.*

Kiayias and Yung [24] give a short survey on cryptography based on polynomial reconstruction and describe the cryptographic primitive "Oblivious Polynomial Evaluation (OPE)" of Noar and Pinkas [32]. OPE is based on the intractability assumption of the noisy polynomial reconstruction. In OPE, Bob has a polynomial P and lets Alice compute $P(x)$ where x is private to Alice in such a way that Bob does not learn anything about x and Alice does not gain any additional information about p. Bleichenbacher and Nguyen [3] present new methods to solve noisy polynomial interpolation using lattice-based methods.

Definition 5. Noisy polynomial interpolation [3, pp. 2]: *Let p be a k-degree polynomial over $\in \mathbb{F}$. Given $n > k+1$ sets $S_1, \ldots, S_n$ and n distinct elements $x_1, \ldots, x_n \in \mathbb{F}$ such that $S_i = \{y_{i,j}\}_{1 \leq j \leq m}$ contains $m-1$ random elements and $p(x_i)$, recover the polynomial p, provided that the solution is unique.*

We note here in all the scenarios presented above, both the evaluation points (inputs) as well as the corresponding outputs are available for polynomial reconstruction and so seemingly do not apply to our problem.

3 Motivating Application

In this section we describe the background leading to the problem setting, adversarial model and contributions of the paper.

We look at the client-server setting wherein a client has large amount of data on which some computation is to be performed and the server has computation power to perform the needed operations. One such operation is to obtain k-Nearest Neighbours described more in the following section. In order to protect privacy, a client may choose to encrypt its data thus limiting the server's ability to perform operations on the same. Encrypting data using homomorphic schemes is one way to allow computation on encrypted data but at the cost of efficiency. A number of works have been proposed in the literature which aim to improve efficiency while allowing computation on encrypted data. We look at cryptanalysis of one such work and in the process look at the problem of integer polynomial reconstruction from only outputs, details of which is the raison d'être of this paper.

3.1 k-Nearest Neighbour Protocol

k-Nearest Neighbour (k-NN) algorithm is a basic method used in data mining, machine learning and pattern recognition used for classification [26, 33]. k-NN is used to search, as per a given metric, k neighbours closest to a given Δ-tuple query point in a database containing n many Δ-tuples. In the privacy preserving version of the k-NN algorithm, the query point is an encrypted Δ-tuple and the database contains n encrypted Δ-tuples. Many *efficient* solutions have been proposed for determining k-NN on encrypted data [12, 15, 23, 37], to name a few. Among these [23] is a solution in a non-colluding federated two-cloud setting more described in Section 3.3.

3.2 Computation on Encrypted Data

Homomorphic Encryption schemes enable computation on encrypted data such that the result of computation is available only after decryption. Current Fully/Somewhat Homomorphic Encryption (F/SHE) [11, 16, 17] schemes come with a price in terms of the large size of ciphertexts and are inefficient when high multiplicative depth circuits like search and sort operations are involved, and hence are impractical when handling a few hundred data elements [7–9]. One way to overcome this is a solution that employs two servers, Server A where encrypted data is stored and Server B that has keys needed to decrypt and perform efficient plaintext computations. Server A does some basic operations on encrypted data followed by some transformation so as to "obscure" it before sharing it with Server B, Server B then decrypts the received information, performs operations on plaintext data efficiently, re-encrypts the results and shares it with Server A to give the processed data back to the end-user client. In these models, it needs to be ensured that Servers A and B do not collude with one another. Moreover, protocols between them should ensure that Server A obscures the data in such a way that Server B is provided only as much data as required for the required computation and in a form from which Server B cannot glean anything more. There are many examples of non-colluding multi-server solutions in the literature deployed in disparate settings, we give a couple of example scenarios. Aono et al. [1] consider a 2-cloud model to compute logistic regression securely and a solution by Hardy et al. [22] uses a 3-cloud federated setting to do learning on vertically partitioned data, which are examples in a machine learning environment; while pRide [30] and PSRide [41] are two solutions in a ride-hailing environment that use 2-cloud models.

3.3 k-NN Protocol over Encrypted Data [23]

At EDBT 2018, Kesarwani et al. [23] present a new Secure k-Nearest Neighbour protocol in the two-party honest-but-curious federated cloud model for computing k-NN on encrypted data. Their solution uses a semantically secure (Levelled) FHE (LFHE) [6] to compute squared Euclidean distances directly on encrypted data. To compute the ranking among the distances, the distances are suitably transformed to preserve the order. A federated non-colluding public cloud having the secret key performs the comparison using the transformed data. Since this cloud has access only to transformed results of computations performed on plaintext data, the paper claims that public cloud does not learn anything useful about the original database, the results or the query. The federated cloud setting consisting of two Servers A and B is depicted in Figure 1.

The paper claims to provide an asymptotically faster solution than the current state-of-the-art protocol. They implement their protocol and run experiments on two real-world datasets from UCI Machine Learning Repository [14], one from healthcare domain and another from financial domain having a relatively large number of dimensions (32 and 23, respectively) and containing 858 and 30000 data points respectively; both of which contain personally identifiable sensitive information like gender, age, marital status etc. The paper states that the data was pre-processed to remove negative integer values. They run their experiments for k = 2, 8 and 16. While the datatype in each dimension is not explicitly mentioned in the paper, by examining the datasets we find that they are non-negative integers of size $\leq$ 16 bits.

The paper makes the following security guarantee regarding the leakage profile for the Server B as given in [23, Theorem 4.2], reproduced here for reference.

"**Theorem 4.2.** Secure k-NN Guarantee: Server B: Our secure k-nearest neighbour protocol leaks no information to Server B except the number of nearest neighbours k to be returned by the protocol and the number of equidistant points in the database with respect to a given query point q".

It may be noted that though the protocol of [23] has been described specifically in the context of securely evaluating k-NN, their technique of transforming through a random monotonic polynomial has applications in many settings where sorting of SHE or LFHE encrypted data is needed. We remark here that if sorting is the only functionality required, then order-preserving or order-revealing encryption schemes would suffice for the purpose [4, 5, 28].

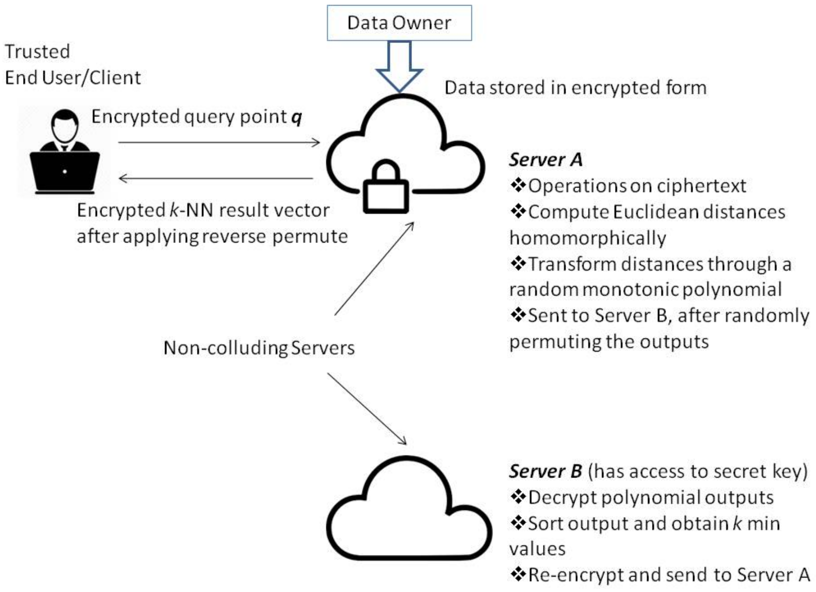
Data Owner
Trusted
End User/Client
Data stored in encrypted form
Encrypted query point q
Server A
❖Operations on ciphertext
❖Compute Euclidean distances homomorphically
❖Transform distances through a random monotonic polynomial
❖Sent to Server B, after randomly permuting the outputs
Encrypted k-NN result vector after applying reverse permute
Non-colluding Servers
Server B (has access to secret key)
❖Decrypt polynomial outputs
❖Sort output and obtain k min values
❖Re-encrypt and send to Server A

Fig. 1: k-NN Setting from [23]

3.4 Problem Setting

We refer to Figure 1 where the data owner outsources his/her database in an encrypted form using a semantically secure homomorphic encryption scheme for storage in Server A. Each data item in the database is of dimension Δ. Server A does not have access to any secret keys and operates only on encrypted data. Server A provides storage for the database and provides services on the encrypted database homomorphically. One of these services is the computation of the k-NN of a given query point q, of dimension Δ, received from an end-user or client. Server A homomorphically computes squared Euclidean Distances (ED) between the query point and each of the n data points in the database. Because a single ED^2 computation is of multiplicative depth 1, it can be efficiently evaluated using an LFHE scheme. Since Server A does not possess the decryption key, it will not be able to efficiently uncover the underlying plaintext information of either the query point or the data points.

Computing Euclidean Distance is of low multiplicative depth, but sorting is not. Hence Server A transforms the distances while preserving its order. It picks a monotonic polynomial $p(x)$ of degree d of the form $a_0+a_1\cdot x+a_2\cdot x^2+\ldots+a_d\cdot x^d$ for some chosen $d \in \mathbb{N}$, where each of the *integer* coefficients a_i are picked uniform randomly and independently in the range $[1, 2^{\alpha}-1]$, for example, in the range $[1, 2^{32}-1]$ as done in [23, Section 3.4]. This polynomial is then evaluated homomorphically for each of the Euclidean distances and the output ciphertexts are re-ordered using a permutation σ picked uniformly at random, before sending them to the Server B for sorting.

Server B possesses the decryption key using which it will decrypt the values received from Server A and sorts them. As the decrypted values are outputs of a random polynomial, the original distances as computed by Server A are “hidden” from Server B. Server B then sends the indices of k-NNs to Server A which then applies σ^{-1} to the received ordering of the indices and forwards the same to the client. We refer to the security guarantee with respect to Server B given in [23, Theorem 4.2] in Section 3.3 and ask the question whether Server B can glean more information about the polynomial chosen by Server A and, eventually, the plaintext integer inputs used by Server A for its homomorphic polynomial evaluation.

Remark: The plaintext data points and the query point are encoded as tuples of integers, since in the context of F/SHE schemes, fixed-point values too are (exactly) encoded using essentially the scaled-integer representation [13]. The only exception is the CKKS FHE scheme [10] that natively supports floating-point arithmetic but was not considered in [23]. However, it may be noted that a successful key recovery attack on CKKS was published in EUROCRYPT 2021 [29]. We leave it for future work to extend our attack to also include any implementation based on the CKKS scheme.

3.5 Adversarial Model

Servers A and B are assumed to be honest but curious. Each will perform the computations correctly but is keen to learn more about the distances between the query point and the data set points. Server A has access only to encrypted data. The semantic security of the underlying encryption scheme will guarantee the protocol is secure against Server A.

Using the decryption key, Server B decrypts the permuted results of computation performed by Server A. After decryption, the Server B would observe only the outputs of the polynomial evaluation (and *not* the input squared distances). That is, it only sees the values on the L.H.S. of the following set of equations:

$$\begin{aligned} p(x_1) &= a_0 + a_1 \cdot x_1 + a_2 \cdot x_1^2 + \ldots + a_d \cdot x_1^d \\ p(x_2) &= a_0 + a_1 \cdot x_2 + a_2 \cdot x_2^2 + \ldots + a_d \cdot x_2^d \\ &\vdots \\ p(x_n) &= a_0 + a_1 \cdot x_n + a_2 \cdot x_n^2 + \ldots + a_d \cdot x_n^d \end{aligned} \tag{1}$$

It is assumed that the Server B, seen here as the adversary, knows the degree d which is usually small since the homomorphic evaluation of polynomials in encrypted form are efficient only for small degrees. It also knows the range $[1, 2^\alpha - 1]$ for the unknown coefficients a_i, and the range $[0, 2^\beta - 1]$ for the unknown inputs x_i. For our attack, we need not know the exact values for the above three parameters, just an upper bound on them would suffice. Also, note that all the parameters above take non-negative integer values.

As noted before, Server A homomorphically evaluates the polynomial $p(x)$ at n (squared Euclidean distance) integer values $x_1, \ldots, x_n$, and we can assume without loss of generality that $0 \leq x_1 \leq x_2 \leq \ldots \leq x_n$. Since $p(x)$ is monotonic, the ordering of $p(x_i)$ is identical to the ordering of x_i. If $a_0 \leq x_1$, then for any given tuple of coefficients $(a_0, a_1, \ldots, a_d)$, there will be a set of positive real roots $(\chi_1, \chi_2, \ldots, \chi_d)$ to (1). Hence, the authors of [23] seem to argue that if the range for a_i is large enough, then it will be infeasible to search for all possible polynomials satisfying (1). The authors claim that the probability that Server B successfully recovers the coefficients a_i, followed by x_i, is approximately $1/2^{\alpha \cdot (d+1)}$, which is negligible when α is large. Referring to the example given in [23, Section 4.2], for α = 16 and d = 9, this probability is approximately 2^{-160}, which is negligible. Hence, the protocol leaks only a negligible amount of information, as claimed in [23], about either a_i or x_i to the Server B and nothing else other than the ordering of the x_i's. We note here that the x_i values may never be uniquely recovered in (1) with probability $= 1$ since $p(x + c)$ is also an equivalent polynomial satisfying the equation for $c \in \mathbb{Z}$ and there may be many values of c such that $0 \leq x_i - c < 2^\beta$. Hence the inputs and the polynomial may only be recovered up to an affine scaling. Other non-linear transformations may also result in an equivalent solution. For instance, $p(\sqrt{x})$ can be a potential solution when all the

x_i are perfect squares and $p(x)$ contains only even powers. But these possibilities will likely be significantly small when $n \gg d$, and the coefficients a_i and/or the inputs x_i come from a random source with sufficiently high entropy, which indeed is the scenario in [23]. Please refer to Section 4.10 where we describe the uniqueness of the obtained solution.

3.6 Our Contribution

We give an algorithm (see Algorithm 1 on pp. 36) to the above defined polynomial recovery problem where the goal is to recover the integer inputs (up to an affine scaling) by observing only the outputs of the monotonic polynomial with positive integer coefficients, assuming the number of evaluation points is much greater in number compared to the degree of the polynomial. Sections 4.1 to 4.7 explain the core idea of our proposed algorithm. Our algorithm has a heuristic expected time complexity that is exponential in the size of the inputs (β) and the degree (d) of the chosen polynomial, but *polynomially* dependent on the size of the coefficients (α). As derived in Equation 11 on pp. 11, the heuristic expected time complexity of our algorithm is $\tilde{O}(d^2\beta 2^\beta(\alpha + d\beta) + nd^3(\alpha + d\beta)^d)$. We would like to note that in many real world scenarios the inputs are/can be encoded as integers of 16- or 32-bits length and our method runs efficiently for inputs of such size. Note also that in SHE applications d is required to be not large as well. Since there can be many solutions to the above polynomial reconstruction problem, hence we will output one solution that satisfies all the output points (there is also a possibility to enumerate all the solutions). But as discussed above, when the number of output values is far bigger than the input degree, and the coefficients and/or inputs are sufficiently random, then the number of equivalent solutions will likely be small. Indeed, we show in Section 4.10 that when the polynomial coefficients are chosen uniform randomly, the original polynomial along with the inputs are recovered by our algorithm with a high probability. Our algorithm extends to recovering any integer polynomial (not necessarily a monotonic integer polynomial) and any input range (not necessarily $[0, 2^\beta - 1]$), as long as the range of the coefficients and inputs are known apriori.

The above results invalidate the security claim in [23, Theorem 4.2] regarding the leakage profile for the Server B. In particular, the Server B will be able to learn the square of the Euclidean distances between the query point and the data set points. It may not be able tell the exact distance to a given point due to random re-ordering but will be able to know all such values. Such an information can potentially help the adversary to narrow down further if it has access to extra information about the underlying data set or query point. For the concrete parameters suggested in [23, Section 4.2], i.e., $\beta = 16$ and $d = 9$, we can recover the inputs for $\alpha = 16$ in a few seconds (see

Table 3 on pp. 29). We then extend it further up to $d = 96$, $\alpha = 128$, $\beta = 48$ and show that we can recover the polynomial and the inputs efficiently (see Table 1 on pp. 28). Lastly, we investigate in Section 6 another variant of the protocol of [23] where the (homomorphically) transformed polynomial outputs are perturbed by a noise yet maintaining the monotonicity. In this case, our previously mentioned attack will not work. But we show, in a straightforward way, that it is still possible to recover the ratios of the inputs.

A preliminary version of our work has appeared as part of 17th IMA International Conference of Cryptography and Coding (IMACC'17), 2019 [31]. We have made a number of improvements to our algorithm (and the material in this paper has more than 50% new material compared to the earlier version). We have also run our experiments with much larger bounds. We list hereunder some of the important changes and enhancements that are present as part of this submission:

1. The heuristic expected running time of our earlier polynomial recovery algorithm (found in the conference proceedings) was $\tilde{\mathcal{O}}(n2^{\beta}(\alpha + d\beta)^{d})$, where n is the number of polynomial integer outputs, the parameters d and α denote the degree and the size of coefficients of the original polynomial, and β denotes the size of the original inputs. This earlier algorithm used a brute force approach to iterate over the input space to try and find a suitable polynomial in the second step.
   In the current version, we have improved the algorithm significantly and the heuristic expected running time of the polynomial recovery algorithm is now $\tilde{\mathcal{O}}(d^2\beta 2^{\beta}(\alpha + d\beta) + nd^3(\alpha + d\beta)^{d})$. This is made possible by avoiding the brute force search mentioned above.
2. In addition to running tests with parameters that were claimed secure in the paper by Kesarwani et al. mentioned above, we show that our algorithm can recover the polynomial and the inputs for larger bounds on the size of inputs, size of coefficients and the degree of the polynomial. In the earlier version, input values were in the range $[0, 2^{24} - 1]$, polynomials up to degree 9 and coefficient space was $[0, 2^{32} - 1]$. In this version, the range of the parameters have been greatly increased; range of input values is now $[0, 2^{64} - 1]$, polynomials up to degree 96 and the polynomial coefficient space is now $[0, 2^{128} - 1]$.
3. When the coefficients are chosen uniformly random, we analyse and show that the recovered polynomial and the input values are, with a very high probability, the same as what was chosen initially.

The earlier version of our polynomial recovery algorithm has also found applications in the privacy-preserving Ride-Hailing Service (RHS) scenario [27] to recover driver locations that are homomorphically transformed by a Service Provider as a means to protect drivers' privacy.

4 Recovering integer inputs given only integer polynomial outputs

We give here a high level overview of the steps we follow to recover all x_i. The key idea is to dramatically reduce the search space of x_i by using the fact that we are looking for only the non-negative integer roots of the polynomial, not just non-negative real numbers.

1. The n number of y_i integer polynomial outputs are input to the algorithm.
2. Obtain integer divisors of $y_i - y_1$ that are less than 2^β with $1 < i \leq n$. One of these divisors will be $x_i - x_1$, which is to be determined.
3. Using the divisors, isolate the possible values of $\delta_i = x_i - x_1$, which constitute the guess for differences of the x inputs.
4. Using x_1 as the unknown parameter, construct a "candidate" degree-d Lagrange polynomial $L(x)$ in x using inputs $(x_1, x_2 = x_1 + \delta_1, \ldots, x_{d+1} = x_1 + \delta_{d+1})$ and outputs $(y_1, y_2, \ldots, y_{d+1})$ respectively, as,
$$L(x) = L_0(x_1) + L_1(x_1) \cdot x + L_2(x_1) \cdot x^2 + \ldots + L_d(x_1) \cdot x^d \qquad (2)$$
whose coefficients $L_i(\cdot)$ are in turn polynomials in the unknown x_1.
5. It is known that the coefficients of $p(x)$ are non-negative integers less than 2^α, and so are that of $L(x)$. Find one or more candidate γ for x_1 such that $0 \leq L_i(x_1 = \gamma) \leq 2^\alpha - 1 \; \forall \, i = 0, \ldots, d$.
6. When one such γ is obtained, use the differences δ_i to find all other inputs x_i. The coefficients of $p(x)$ that correspond to γ are immediately obtained as $L_i(x_1 = \gamma)$.
7. Using $p(x)$ thus obtained, use the remaining $(n - d - 1)$ outputs to verify the correctness by computing the roots of the polynomial and checking if they are integers that lie in the range $[0, 2^\beta - 1]$. If these verifications are successful, algorithm outputs the satisfying polynomial.

Remarks:
1. The polynomial $p(x)$ output by the algorithm is just one possible solution set satisfying the given y_i values, since for example $(x_i + c)$ with $c \in \mathbb{Z}$ gives the same y_i outputs for the polynomial $p(x - c)$.
2. In Section 4.10, we give an analysis on how many such values of c can exist when coefficients of $p(x)$ are chosen at random.
3. While our method works for arbitrary values of α, β and d, we demonstrate the practicality of our solution by running experiments with α up to 128, β up to 64, d up to 96, as described in detail in Section 5.
4. While n is typically bigger, $n = 2d$ was sufficient for polynomial recovery in our experiments.

The procedure is given as Algorithm 1 in Appendix A and each of the steps are described in detail in the following sections.

4.1 Divisors from y-differences

Summary of this step: Given a non-decreasing order of n outputs $y_i = p(x_i)$ with y_1 being the smallest, the compute y-differences $y_{i,j} = y_i - y_j$, where $i > j$, followed by finding divisors of $y_{i,j}$, of which one divisor is equal to $x_i - x_j$. Differences of y values are stored in Matrix Y and the corresponding divisors are stored in Matrix D at the end of this step.

Refer to the procedure `GuessTheDifference` from Algorithm 1.

Lemma 1. *Let $p(x) \in \mathbb{N}[x]$ be a degree-d polynomial, $y_i = p(x_i)$, $y_j = p(x_j)$, $x_i \neq x_j$, then $(x_i - x_j)|(y_i - y_j)$.*

Proof. Let $y_i = a_d {x_i}^d + \cdots + a_1 x_i + a_0$ and $y_j = a_d {x_j}^d + \cdots + a_1 x_j + a_0$, where $a_i, 0 \leq i \leq d$, are the coefficients of $p(x)$.
Then $y_i - y_j = a_d({x_i}^d - {x_j}^d) + \cdots + a_1(x_i - x_j) \Rightarrow (x_i - x_j)|(y_i - y_j)$. □

Lemma 2. *Let $p(x) \in \mathbb{N}[x]$ be a degree-d polynomial, $y_i = p(x_i)$, $y_j = p(x_j)$ and $x_i < 2^\beta \, \forall i$, then $0 < (x_i - x_j) < 2^\beta$.*

Proof. By our choice of ordering the polynomial outputs, $y_i > y_j$ and $p(x)$ is monotonic as all its coefficients are non-negative integers, $\Rightarrow x_i - x_j > 0$ and each x_i is upper bounded by 2^β. □

Let $y_{i,j} = y_i - y_j$. From Lemmas 1 and 2, $(x_i - x_j) < 2^\beta$ is a divisor of $y_{i,j}$ and as part of this step, we find all positive divisors of $y_{i,j} < 2^\beta$. For $\beta \leq 28$, we use the sieve method to obtain factors by dividing $y_{i,j}$ by successive primes and forming positive divisors $< 2^\beta$. When $\beta > 28$ and since we know that there will exist a small factor (as in our case, since the range of x_i are known and the factors are differences of x_i) we use elliptic-curve factorization method (ECM) [35] to find the required factors.

Time Complexity for finding factors. In the expected case, ECM is a sub-exponential algorithm with complexity $\mathcal{O}(e^{\sqrt{(\log p \ \log\log p)(1+\mathcal{O}(1))}})$ to find a single factor, where p is the smallest factor. We first use the sieve method to obtain all small factors $\leq 2^{28}$ and then invoke SageMath `ECM.find_factor()` on Q, where Q is the quotient obtained by dividing $y_{i,j}$ by all the small factors and their multiplicity, to obtain the next possible factor f. If $f < 2^\beta$ we continue the process of invoking `ECM.find_factor()` on Q/f as long as we obtain a factor $< 2^\beta$ or we hit a

timeout. The timeout value is empirically set such that the above function is able to find "small" factors ($< 2^\beta$) well within the timeout.
Remark. There is a possibility of losing factors due to our timeout being insufficient. Section 5 has results on loss of factors for different timeout values. But we would like to stress that we must enumerate all the factors of the differences that are less than 2^β if we want to eliminate any risk of missing any satisfying polynomial.

Let $D_{i,j}$ be the set of all divisors of $y_{i,j}$. This set constitutes the guesses for the differences of the (to be determined) values x_i. It turns out that for many values of $y_{i,j}$ there may be too many divisors that are $< 2^\beta$, so we need to sample larger number of output values (i.e., larger n) and carefully pick d number of $y_{i,j}$'s such that the number of elements in each of $D_{i,j}$ is a small positive number (say, $\leq \psi$), whereby the search space for the guesses becomes feasible to enumerate. We also store the corresponding $y_{i,j}$ values in Matrix Y that are to be used later in the Lagrange interpolation step .
Remark. In our experiments as well as in our analysis, we consider $\psi = \mathcal{O}(\alpha + d\beta)$, since the number of divisors of an integer N is bounded by $N^{\mathcal{O}\left(\frac{1}{\log\log N}\right)}$ and on the average case it is $\log N$ [40].

$$Y = \begin{pmatrix} 0 & 0 & 0 & \dots \\ y_{2,1} & 0 & 0 & \dots \\ y_{3,1} & y_{3,2} & 0 & \dots \\ \vdots & & & \\ y_{d+1,1} & y_{d+1,2} & \dots & y_{d+1,d} \end{pmatrix}$$

The output of this step is Matrix D of divisor sets :

$$D = \begin{pmatrix} 0 & 0 & 0 & \dots \\ D_{2,1} & 0 & 0 & \dots \\ D_{3,1} & D_{3,2} & 0 & \dots \\ \vdots & & & \\ D_{d+1,1} & D_{d+1,2} & \dots & D_{d+1,d} \end{pmatrix}$$

Time Complexity. The worst-case time complexity of this step (in terms of bit operations) to find the factors less than 2^β is

$$\tilde{\mathcal{O}}(d^2(\alpha + d\beta) \cdot \beta \cdot 2^\beta) \tag{3}$$

since the size of each $Y_{i,j}$ is $\mathcal{O}(\alpha + d\beta)$ and there are $\mathcal{O}(d^2)$ elements in $D_{i,j}$.

4.2 Consistency Check on the Guessed Differences

Summary of this step: The key idea in this step is to eliminate as many divisors as possible before performing Lagrange Interpolation in the following step.

Refer to the procedure `CheckConsistencyInColumn` from Algorithm 1 in Appendix A.

Definition 6. Consistent Divisor: *Given the matrix of divisors D and $f \in D_{i,k}$, if $\exists\, g \in D_{j,k}$ such that $(f - g) \in D_{i,j}$ $\forall\, j$, $1 < j < i$ and $\forall\, k$, $1 < k < i-1$, then f and g are termed as Consistent Divisors of their respective sets.*

Remark. If $x_i - x_k$ is a divisor of $y_i - y_k$, and $x_j - x_k$ is a divisor of $y_j - y_k$, then $x_i - x_j$ must be a divisor of $y_i - y_j$.

The output of the previous step is the matrix D. Element $D_{i,j} \in D$, is a list of divisors of $y_{i,j}$, and we know that only one of them is (but as yet unknown) $x_i - x_j$. We create sets of Consistent Divisors for each $D_{i,j}$ $(2 \leq i \leq d+1)$ and $(j < i)$. All elements that are not part of the consistent set are set equal to 0.

For each non-zero element in $D_{2,1}$, pick its corresponding "consistent" element δ_{i-1} from $D_{i,1}$, $2 < i \leq d+1$ to form a tuple of $d+1$ elements $(\delta_0 = 0, \delta_1, \ldots, \delta_d)$. Store each tuple in an array of tuples C. The output of this step is Array C and $|C|$.
Remark. In the beginning of Section 4 we use the notation δ_i to mean the $x_i - x_1$ differences, but in this section we use the same notation to mean "consistent" elements. This is because they represent the same element. In other words, a "consistent" element is one that is a divisor of the y output difference as well as (yet to be identified) x input difference.

Time Complexity. The worst-case time complexity of this step is

$$\tilde{\mathcal{O}}(d^3\psi^2\beta) \tag{4}$$

as $\mathcal{O}(d^2)$ elements each having $\mathcal{O}(\psi)$ factors of size $\mathcal{O}(\beta)$ bits are compared in $\mathcal{O}(d)$ columns.

4.3 Recursive Consistency Check

Summary of this step: In Section 4.2, we obtain the consistency set by examining the divisors of differences of the polynomial outputs and using that we obtain a set of tuples that constitute the sets of guessed differences of inputs. In this step, the differences are obtained in a different manner as described below. These two sets of guessed differences are compared to enable us to converge faster on the set of x differences.
Remark. In the worst-case we need to run this step for ψ^d number of times, so we enable this procedure only for experiments with small polynomial degree. In practice, this step quickly reduces the number of inconsistent tuples in Array C, thus reducing the overall running time of the algorithm.

Refer to the procedure `RecursiveConsistencyCheck` from Algorithm 1 in Appendix A.

This step takes the tuple Array C as input. The d elements of the each tuple of Array C, for example $C[i][1]$ to $C[i][d]$ of the i^{th} tuple are factors of y_i output differences, as obtained the previous step. Using the d elements of the first tuple, we can rewrite the first column of Matrix Y as:

$$\begin{aligned}
y_{2,1} &= y_2 - y_1 = (x_2 - x_1)[\,\cdot\,]_{2,1} \\
y_{3,1} &= y_3 - y_1 = (x_3 - x_1)[\,\cdot\,]_{3,1} \\
&\vdots \\
y_{d+1,1} &= y_{d+1} - y_1 = (x_{d+1} - x_1)[\,\cdot\,]_{d+1,1}
\end{aligned}$$

We start with Column $m = 1$ of the Matrix Y, and rename y_i outputs as $z_{i,0}$ (for ease of notation) for $1 \leq i \leq d+1$. In other words, y_1 is renamed $z_{1,0}$, y_2 is renamed $z_{2,0}$ and so on. In Step 1 of this procedure, we compute quotients $[\,\cdot\,]_{2,1} = \frac{z_{2,0}-z_{1,0}}{x_2-x_1}$, $[\,\cdot\,]_{3,1} = \frac{z_{3,0}-z_{1,0}}{x_3-x_1}$, and so on. Generalizing this, in step m, we compute the quotients $[\,\cdot\,]_{i,m}$, for $(m+1) \leq i \leq (d+1)$, as

$$z_{i,m} = \frac{z_{i,m-1} - z_{m,m-1}}{x_i - x_m}.$$

Next, we find the differences $z_{i,m} - z_{m+1,m}$ for $(m+1) < i \leq (d+1)$. We then find divisors ($< 2^\beta$) of each of these differences. We call this divisor set as $DN_{i,m+1}$. In Matrix D, we set $D_{i,m+1} = DN_{i,m+1} \cap D_{i,m+1}$. We then build the consistent set for Column m similar to the way described in Section 4.2. Finally, we pick the non-zero element in Column m in every row to form a tuple τ of $(d-m)$ number of elements, which form the guessed difference divisors for the next recursive step. The recursion terminates in $d+1$ steps as given in Lemma 3. In each step, the computed τ is used as input to the subsequent call.

Lemma 3. *Given $p(x) \in \mathbb{N}[x]$, a degree-d polynomial, the procedure Recursive Consistency check terminates recursion for a single tuple of guessed differences in at most d steps.*

Proof. In step 1 of the procedure, the y_i outputs are that of $p(x)$, a degree-d polynomial. By taking the difference of the y_i outputs, the coefficient a_0 is eliminated. The input tuple of guessed differences are factored out from the y_i differences leaving integer quotients. In step 2, by taking differences of quotients, the next coefficient, namely, a_1 is eliminated and the process is repeated in each subsequent step. Thus in step i the coefficient a_{i-1} is eliminated. Finally, in step d :

1. the coefficient a_{d-1} is eliminated

2. the guessed differences at this step are factored out leaving the quotient $[\cdot]_{d+1,d} = a_d$, the coefficient of x^d of $p(x)$
3. the algorithm terminates for the input tuple of guessed differences and returns a_d.

During any recursive step, if a non-integer quotient is obtained or if no non-zero element remains in the consistent set, then the recursion is terminated returning 0. □

As shown in Lemma 3, for a single tuple of guessed differences, this step terminates in at most d steps. If no a_d is returned, then the tuple in Array C is discarded and the procedure is repeated using the next tuple in Array C.
When recursion completes successfully, we get

$$z_{d+1,d} = \frac{z_{d+1,d-1} - z_{d,d-1}}{x_{d+1} - x_d}.$$

We obtain $z_{d+1,d} = a_d$ as given in Lemma 3. We note that a tuple in Array C having values with successive differences matching that of the original x_i values will successfully go through all the $d+1$ steps until an integer a_d is obtained.

Time Complexity. In this step, the factors are obtained for $\mathcal{O}(d^2)$ elements followed by running consistency check on $d-1$ columns. For one tuple, the worst-case time complexity of this step based on (3) and (4) is

$$\tilde{\mathcal{O}}(d^2(\alpha + d\beta) \cdot \beta \cdot 2^\beta + (d^3\psi^2\beta)) \tag{5}$$

4.4 Choosing Optimal Divisors Sets

Summary of this step: In Section 4.2, we enumerate over ψ^d many tuples of divisors/guesses. We provide here a way to choose the divisors sets such that the enumeration complexity is as small as possible.

Definition 7. Output Difference Graph: *A complete undirected graph $G = (V, E)$ with $|V| = n$, $E \subseteq V \times V$, where the number of elements in $D_{i,j}$ is the edge cost between nodes V_i and V_j.*

We refer to Equation (1) giving the polynomial outputs. As explained in Section 4.1, we compute the differences of polynomial outputs and form the Matrix D of divisors sets (less than 2^β) of the output differences. We need to find a set of d many $D_{i,j}$ elements of the Matrix D such that $\prod D_{i,j}$ is minimum. In other words, the product of the number of guesses is minimum. Graph G is the Output Difference Graph of Matrix D. Now finding the minimum $\prod D_{i,j}$ is akin to finding the d-Minimum Spanning Tree (i.e., a minimum weight tree with d edges only) in G, where the weight of the tree is

represented by the product of the weights. The requirement that the subgraph is a tree comes from the linear independence requirement of the corresponding set of equations. Essentially, we are transforming the problem of finding the small search space of divisors to the problem of finding a d-minimum spanning tree having the least cost (in terms of divisor product) across all divisors sets of Matrix D yet satisfying the linear independence condition. It is shown in [34] that the d-MST problem is NP-hard for points in the Euclidean plane. The same paper provides an approximation algorithm to find d-MST [34, Theorem 1.2] and is reproduced here for reference.
"**Theorem 1.2.** There is a polynomial-time algorithm that, given an undirected graph G on n nodes with non-negative weights on its edges, and a positive integer $k \leq n$, constructs a tree spanning at least k nodes of weight at most $2\sqrt{k}$ times that of a minimum-weight tree spanning any k nodes."
Note that this approximation algorithm also works for multiplication of edge weights (weights greater than 1) since by extraction of logarithms this can be trivially turned into addition of edge weights. Using this algorithm, we can carefully select $d < n$ nodes having close to the minimum enumeration complexity in order to make our search space feasible to guess the differences $(x_i - x_j)$ with $x_i \geq x_j$. From the d-MST so obtained, we can now go on to find the set of divisors $(x_i - x_j)$ such that they are consistent as explained in Section 4.2 and continue with finding the polynomial coefficients using Lagrange interpolation as described in Algorithm 1. However, we did not implement this optimisation in our code as the concrete running time was already small enough. But for larger instances this optimisation will be useful.

4.5 Lagrange interpolation to recover all coefficients

Summary of this step: Use Lagrange interpolation to construct a degree-d polynomial using the guessed difference set.

Refer to the procedure `ApplyLagrangeInterpolation` from Algorithm 1 in Appendix A.

Each tuple $(\delta_0 = 0, \delta_1, \ldots, \delta_d)$ in the Array C obtained as described in Section 4.2 presents possible candidates for the differences $(x_i - x_1)$, $1 \leq i \leq d+1$. Optionally, we could use the method described in Section 4.3 to narrow down the possible candidates to only those tuples for which an integer a_d coefficient is obtained.

Since δ_i represents differences with respect to some unknown value x_1, we treat x_1 as an unknown parameter and add it to each of the d differences to get $(x_1, x_2, \ldots, x_{d+1}) = (x_1, x_1 + \delta_1, \ldots, x_1 + \delta_d)$. The Lagrange polynomial of degree-d is obtained using these values for inputs x, expressed in terms of x_1 and the corresponding output values come from the y set as given in Section 4.1. We now have a degree-d polynomial L in x with parameter x_1 as shown in Equation (6).

$$L(x) = \sum_{k=1}^{d+1} \frac{\prod_{j=1, j\neq k}^{j=d+1} (x - x_j)}{\prod_{j=1, j\neq k}^{j=d+1} (x_k - x_j)} \cdot y_k \tag{6}$$

Lemma 4. *Given a degree-d polynomial L(x) with parameter x_1 as shown in Equation 6, can be written as a polynomial in x whose coefficients are polynomials in x_1.*

Proof. (1) The denominator of each of the summands in Equation 6 $\in \mathbb{Z}$ since it is a product of only integers and does not contain the parameter x_1.
(2) The numerator of each of the summands in Equation 6 is a product of d degree-1 integer binomials. We know that the product of d degree-1 integer binomials of the form $(x+p_1)(x+p_2)\dots(x+p_d) = x^d + x^{d-1}\sum p_i + x^{d-2}(\sum p_i p_j) + \dots + \prod p_i$. Now since each of the $p_i = x_1 + \delta_i$, it follows that the coefficient of x^i is a degree-$(d-i)$ polynomial in x_1 and $\prod_d p_i$ is a degree-d polynomial in x_1.
From statements (1) and (2) above, $L(x) = L_0(x_1) + L_1(x_1)x + L_2(x_1)x^2 + \dots + L_d(x_1)x^d$ and each $L_i(x_1) \in \mathbb{Q}[x_1]$. □

From Lemma 4, each L_i is a degree-$(d-i)$ polynomial in x_1, $0 \le i \le d$ and $L_d(x_1) = a_d$ is a constant polynomial. Thus, we have recovered a_d, the coefficient of x^d in $p(x)$.

We know that the coefficients of $p(x)$ are positive integers less than 2^α, and $p(x)$ is evaluated at integers in the range $[0, 2^\beta - 1]$. Our next objective is to output an integer $x_1 \in [0, 2^\beta - 1]$ such that $\forall i = 1, 2, \dots, d$, $1 \le x_i = x_1 + g_i \le 2^\beta - 1$ and $\forall i = 0, \dots, d-1$, $L_i(x_1)$ is an integer with $0 \le L_i(x_1) \le 2^\alpha - 1$ as described in Section 4.6.. If no such x_1 exists, then we repeat this process using the next tuple in C.

Time Complexity. The worst-case time complexity of computing the Lagrange interpolation polynomial for one tuple is

$$\tilde{O}(d^2(\alpha + d\beta)^2). \tag{7}$$

4.6 Obtaining x_1 by Finding Roots of Polynomials $L_i(x_1)$

Summary of this step: Given α, β and polynomials $L_0, L_1, \dots L_{d-1}$ (each $L_i \in \mathbb{Q}[x_1]$ and is of degree d_i, respectively, where $d_i = d - i$), find $x \in \mathbb{Z}$ satisfying the condition: $0 \le x \le 2^\beta - 1$ and $0 \le L_i(x) \le 2^\alpha - 1 \; \forall \, i = 0, \dots, d-1$.

Refer to the procedure `RecoverPossibleXValuesAndPoly` from Algorithm 1 in Appendix A.

It is easy to see that the following algorithm always outputs one such x if it exists (if no integer satisfies the required condition, then this algorithm returns `failure`).

1. For each $i = 0, \dots d-1$, find the ordered set $S_i = \{[a_k, b_k] : a_k, b_k \in \mathbb{Z}, 0 \leq a_k \leq b_k \leq 2^\beta - 1\}$ of disjoint intervals on the real line such that *every* integer m satisfying $0 \leq m \leq 2^\beta - 1$ and $0 \leq L_i(m) \leq 2^\alpha - 1$ is present in some interval $[a_k, b_k] \in S_i$.
2. If there exists an integer x in the range $[0, 2^\beta - 1]$ that is present in some interval from *each* S_i output it, else return `failure`.

Step 1. Finding the ordered set S_i of intervals for each L_i:

Compute the set $R_{i,0}$ of all the (approximations to the) real roots of $L_i(x) = 0$ that lie in the range $[0, 2^\beta - 1]$. Similarly let $R_{i,\alpha}$ contain all the real roots of $L_i(x) = 2^\alpha - 1$ in the range $[0, 2^\beta - 1]$. Let $R_i = R_{i,0} \cup R_{i,\alpha}$ be an ordered set with its elements in increasing order (as they appear on the real line).

Intuition. We first provide an intuition for our algorithm by looking at one particular case. Let a, b (with $a \leq b$) be two consecutive elements of the ordered set $R_i = R_{i,0} \cup R_{i,\alpha}$. For now, assume that $a \in R_{i,0}$ and $b \in R_{i,0}$. Then, the following Lemma easily follows.

Lemma 5. *There are only two possibilities for the values taken by $L_i(x)$ for $x \in (a, b)$: either $0 \leq L_i(x) \leq 2^\alpha - 1 \; \forall \, a < x < b$, or $L_i(x) < 0 \; \forall \, a < x < b$.*

Proof. Note that L_i is continuous everywhere in its domain. Since a, b are consecutive elements in R_i, graphically, the curve L_i does not intersect the lines $y = 0$ and $y = 2^\alpha - 1$ at any point x between a and b. Since $L_i(a) = L_i(b) = 0$, there are only two possibilities: the curve either lies entirely between $y = 0$ and $y = 2^\alpha - 1$, or entirely below $y = 0$. □

To determine which of the aforementioned possibilities occurs, it is enough to compute $L_i(c)$ at any c between a and b. If $0 < L_i(c) < 2^\alpha - 1$, the interval $[\, \lceil a \rceil, \lfloor b \rfloor \,]$ contains *all* integers between a and b satisfying $0 < L_i(\cdot) < 2^\alpha - 1$. Add this interval to S_i. (In our algorithm, we take the midpoint $c = \frac{a+b}{2}$).

Similarly, it is straightforward to analyze other cases – a belongs to $R_{i,0}$ and b belongs to $R_{i,\alpha}$, and so on. When we consider all such consecutive pairs a, b in R_i, we end up with a set of intervals S_i such that every integer m between 0 and $2^\beta - 1$ that satisfies $0 < L_i(m) < 2^\alpha - 1$ is present in some interval of S_i.

Main Idea. Here, we extend the above arguments and describe the algorithm for finding S_i. Let a be the first element in R_i. If $0 \notin R_i$ then for all $0 < x < a$ there are three possibilities: $L_i(x)$ is either entirely above $y = 2^\alpha - 1$ (when $a \in R_{i,\alpha}$), or entirely below $y = 0$ (when $a \in R_{i,0}$), or entirely between $y = 0$ and $y = 2^{\alpha} - 1$. Therefore, add $[0, \lfloor a \rfloor\,]$ to S_i if $0 < L_i(\frac{a}{2}) < 2^{\alpha} - 1$ and $0 \notin R_i$.

Next for every pair of consecutive elements a, b in the (sorted) enumeration of R_i there are four cases based on whether they belong to $R_{i,0}$ or $R_{i,\alpha}$.

– $a \in R_{i,0}$ and $b \in R_{i,0}$: If $0 < L_i(\frac{a+b}{2}) < 2^{\alpha} - 1$, then add $[\,\lceil a \rceil, \lfloor b \rfloor\,]$ to S_i. If $L_i(\frac{a+b}{2}) < 0$ and if a (resp. b) is an integer, we still include them since $L_i(a) = 0$ and $L_i(b) = 0$. In this case, add $[a, a]$ (resp. $[b, b]$) to S_i.
– $a \in R_{i,0}$ and $b \in R_{i,\alpha}$: Now the only possibility is that for all x, $a < x < b \Rightarrow 0 < L_i(x) < 2^{\alpha} - 1$ (that is, L_i is non-decreasing in this region). This is because $L_i(a) = 0, L_i(b) = 2^{\alpha} - 1$ and $L_i(c)$ cannot be 0 or $2^{\alpha} - 1$ for any $a < c < b$. So, add $[\,\lceil a \rceil, \lfloor b \rfloor\,]$ to S_i.
– $a \in R_{i,\alpha}$ and $b \in R_{i,0}$: Similar to the previous case, we once again have $a < x < b \implies 0 < L_i(x) < 2^{\alpha} - 1$ and L_i is non-increasing in this region. Add $[\,\lceil a \rceil, \lfloor b \rfloor\,]$ to S_i.
– $a \in R_{i,\alpha}$ and $b \in R_{i,\alpha}$: For each $a < x < b$, $L_i(x)$ does not intersect $y = 0$ and $y = 2^{\alpha} - 1$. Since $L_i(a) = L_i(b) = 2^{\alpha} - 1$, this means that L_i is either entirely above or below $y = 2^{\alpha} - 1$. So, add $[\,\lceil a \rceil, \lfloor b \rfloor\,]$ to S_i if $0 < L_i(\frac{a+b}{2}) < 2^{\alpha} - 1$, otherwise if a (resp. b) is an integer, then add $[a, a]$ (resp. $[b, b]$) to S_i.

Let b be the last element in R_i. If $2^{\beta} - 1 \notin R_i$, for all $b < x < 2^{\beta} - 1$, there are three possibilities: $L_i(x)$ is either entirely above $y = 2^{\alpha} - 1$ (when $b \in R_{i,\alpha}$), or entirely below $y = 0$ (when $b \in R_{i,0}$), or between $y = 0$ and $y = 2^{\alpha} - 1$. Add $[\,\lceil b \rceil, 2^{\beta} - 1]$ to S_i if $0 < L_i(\frac{b+2^{\beta}-1}{2}) < 2^{\alpha} - 1$ and $2^{\beta} - 1 \notin R_i$.

To ensure S_i consists of disjoint intervals, combine consecutive elements $[a, b], [c, d] \in S_i$ into a single interval $[a, d]$ if $b = c$. Since intervals were added to S_i in a sorted manner, after combining them, the disjoint intervals in S_i are ordered as they would appear in the real line. This property will be useful when the intersection of intervals is computed in the next step.

Time Complexity. For every L_i, the equations $L_i(x) = 0$ and $L_i(x) = 2^{\alpha} - 1$ can each have at most d_i solutions. All real roots for these equations can be computed in time $\tilde{\mathcal{O}}(d_i^3 \tau)$ [25, 36], where τ is the minimum number of bits required to represent all coefficients of L_i. In our case $\tau = \tilde{\mathcal{O}}(\alpha + d\beta)$. The size of R_i is at most $2d_i$ and sorting it takes time $\tilde{\mathcal{O}}(d_i)$. Adding intervals to S_i for every pair of consecutive elements in R_i (and ensuring that they are disjoint) takes $\tilde{\mathcal{O}}(d_i)$. We perform this for all polynomials $L_0, L_1, \ldots, L_{d-1}$ leading to a time complexity of

$$\tilde{\mathcal{O}}(\sum_{i=0}^{d-1} (d_i + d_i^3 \tau)) = \tilde{\mathcal{O}}(d^3(\alpha + d\beta)). \tag{8}$$

Remark. When finding real roots of $L_i(x) = 0$ and $L_i(x) = 2^\alpha - 1$, it is sufficient to compute approximations of these roots up to certain decimal places. For the values of d, α, β that we consider for our experiments in Section 5.1 (refer Table 1), obtaining roots accurately up to 30 decimal places is sufficient for our algorithm to correctly output the required integer.

Step 2. Finding an x Belonging to Some Interval in Each S_i:

When we refer to an interval $I = [a, b]$ (with start point $a = I.start$ and end point $b = I.end$) it is understood that $a \leq b$. A number $x \in I$ if $a \leq x \leq b$. Two intervals $[a, b], [c, d]$ intersect if $max(a, c) \leq min(b, d)$ in which case the interval denoting their intersection is $[max(a, c), min(b, d)]$. Simultaneous intersection of k intervals $I_1, I_2, \ldots I_k$ can be computed iteratively: let X store the required result, which is initially set to I_1. For $i = 2 \ldots k$, set X as the intersection of intervals X and I_i.

Let $S_{i,j}$ be the j^{th} element/interval in the (ordered) set S_i. Our objective is to identify potential intervals in each S_i that contain a common integer $x \in [0, 2^\beta - 1]$. In other words, find an integer $x \in [0, 2^\beta - 1]$ and indices $i_0, i_1, \ldots i_{d-1}$ such that $x \in S_{0,i_0}, x \in S_{1,i_1} \ldots x \in S_{d-1,i_{d-1}}$. A recursive approach can be used: out of $S_0, S_1, \ldots S_{d-1}$, let S_k be the set with the smallest end point in its first interval, in other words, $S_{k,1}.end = \min\limits_{i=0 \ldots d-1} \{S_{i,1}.end\}$. Let I be the interval representing the intersection of $S_{0,1}, S_{1,1} \ldots S_{d-1,1}$. We have two possibilities:

– I is empty: this implies that $S_{k,1}$ will never contribute towards the required common element in future iterations of the recursion since it ends first among all intervals from $S_0 \cup S_1 \cdots \cup S_{d-1}$ on the real line. So remove $S_{k,1}$ from S_k and recurse on the new sets $S_0 \ldots S_{d-1}$.

– I is non-empty: if I contains an integer x in the range $[0, 2^\beta - 1]$ output it since x is common to all $S_{0,1}, S_{1,1}, \ldots S_{d-1,1}$. Otherwise, remove $S_{k,1}$ from S_k as before and recurse.

The size of $S_0 \cup S_1 \cdots \cup S_{d-1}$ decreases by one in each iteration. The recursion terminates by default when any S_i becomes empty, since the required integer must be present in some interval in each S_i. Therefore, once any S_i becomes empty no such x exists and the algorithm returns `failure`.

Time Complexity. The number of intervals in each S_i is $\mathcal{O}(d_i)$ since the size of R_i is bounded by $2d_i$. In the worst case, when no integer satisfies the desired condition, the recursion runs for $|S_0 \cup S_1 \cup \cdots \cup S_{d-1}| = \mathcal{O}(\sum_{i=0}^{d-1} d_i)$ steps. Finding the iterative intersection of $S_{0,1}, S_{1,1} \ldots S_{d-1,1}$ in each step can be done in time $\mathcal{O}(d)$. The worst case time complexity is

$$\mathcal{O}(d \sum_{i=0}^{d-1} d_i) = \tilde{\mathcal{O}}(d^2). \tag{9}$$

4.7 Verification of the Polynomial Against the Remaining Outputs

Once a candidate polynomial is obtained as described in Section 4.6, we use the remaining $(n-d-1)$ outputs to verify the correctness by computing the roots of the polynomial and check if they are integers that lie in the range $[0, 2^{\beta}-1]$. If these verifications are successful, then we say that the algorithm has output a satisfying polynomial.

Time Complexity. We need to compute d roots of $\mathcal{O}(n)$ polynomials where the size of each coefficient is $\mathcal{O}(\alpha + d\beta)$. The worst-case time complexity of this step is

$$\tilde{\mathcal{O}}(nd^3(\alpha + d\beta)) \tag{10}$$

4.8 Correctness of Algorithm 1

Theorem 1. *Given n number of evaluations of a d-degree monotonic integer polynomial $p(x)$, our algorithm will output a polynomial $p'(x) \in \mathbb{N}[x]$ that satisfies the bound requirements on inputs and polynomial coefficients as stated in Problem Statement 1.*

From Lemmas 1 and 2, $(x_i - x_j) < 2^{\beta}$ is a positive divisor of $(y_i - y_j)$. As described in Section 4.1, we obtain all divisors $< 2^{\beta}$ of y-differences of which one of them is a respective x-difference yet to be determined. Next, by forming the consistency-set as described in Section 4.2 we try to eliminate divisors that are "unlikely" to be x-differences. Valid x-differences are guaranteed to be in the consistent set, since given two guessed differences $(x_i - x_1)$ and $(x_j - x_1)$, $(x_i - x_1) - (x_j - x_1) = (x_i - x_j)$ is another guessed difference that should be present in the divisor set of $(y_i - y_j)$. Tuples of guessed differences are formed by picking one consistent element from each row of Column 1 of Matrix D as given in Section 4.1. One of these tuples will be the correct set of x-differences. We then use each tuple to get a candidate polynomial using Lagrange interpolation as described in Section 4.5 and then find x_1 as described in Section 4.6, followed by verification against all outputs. At the end of Algorithm 1, we isolate the set of guessed differences with respect to x_1 and output a polynomial $p'(x)$ such that $p'(x_i) = y_i$ for $1 \leq i \leq n$. Hence as long as we are successful in obtaining all divisors of the y differences, our algorithm will output a polynomial $p'(x)$ satisfying all requirements.

4.9 Consolidated Time Complexity of the Polynomial Reconstruction Algorithm

Theorem 2. *Our algorithm for reconstruction of a degree-d monotonic integer polynomial when given only the outputs of the polynomial evaluated at a sufficient number of inputs, as stated in Problem Statement 1, has a worst-case time complexity of* $\tilde{O}(d^2(\alpha + d\beta) \cdot \beta \cdot 2^\beta + nd^3(\alpha + d\beta)^d)$.

For each of the steps of our polynomial reconstruction algorithm described above, we either provided a worst-case or a (heuristic) expected-case analysis of the running time. However, it looks difficult to do a tight/rigorous analysis of some steps as they deal with, for instance, the distribution of the divisors of the polynomial outputs evaluated at arbitrarily chosen inputs. Hence, we could only provide a heuristic bound on the expected running time.

Using (3) and (4), and considering that (7), (8) (9) and (10) need to be executed ψ^d times in the worst-case, we get the total heuristic expected running time of Algorithm 1 (in terms of bit operations) as

$$\tilde{O}(d^2(\alpha + d\beta) \cdot \beta \cdot 2^\beta + nd^3(\alpha + d\beta)^d). \tag{11}$$

Remark. We do not consider the time complexity of Section 4.3 while computing the consolidated time analysis in (11) since it is not enabled for experiments involving large parameters. (5) in Section 4.3 gives the time complexity for *RecursiveConsistencyCheck* procedure for one tuple. When it is included as part of Algorithm 1, in the worst-case it needs to be executed ψ^d times making the total heuristic expected running time of Algorithm 1 as

$$\tilde{O}(d^2(\alpha + d\beta)^d \cdot 2^\beta). \tag{12}$$

4.10 On the Uniqueness of the Satisfying Polynomial when Coefficients are Uniform Randomly Sampled

We recall that after executing the Algorithm 1, we recover a satisfying polynomial and the corresponding input values x_i that satisfies the given output values y_i. As we have remarked earlier, it may be possible to recover the x_i values only up to a affine scaling. This is because the outputs for $p(x)$ evaluated at x_i are identical to $p(x + c)$ (for $c \in \mathbb{Z}$) evaluated at $x_i - c$, respectively.

We consider here the case when the coefficients a_i ($0 \leq i \leq d$) of $p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$ are picked uniform randomly and independently from the range $[1, 2^\alpha - 1]$. We show below, within certain constraints, that the value of x_1 obtained as described in Section 4.6 along with the guessed differences that results in a polynomial satisfying all conditions are, with a high probability, the original integer inputs that were used to compute the polynomial outputs. Let x_p denote the value of x_1 obtained after the execution of Step 2 of Section 4.6. We examine below the resulting coefficients of the polynomial $p(x)$ evaluated at integer values less than and greater than x_p, namely $(x_p - c)$ for $0 < c < x_p$ and $(x_p + c)$ for $0 < c < 2^\beta - x_p$.

4.10.1 Evaluation at $(x_p - c)$

Theorem 3. *Given* $p(x) \in \mathbb{N}[x]$ *with* $p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$ *and* $Y = \{p(x_p), p(x_p + \delta_1), p(x_p + \delta_2), \ldots\}$. *The probability that at least one coefficient of* $p(x)$ *goes beyond the coefficient range of* $[1, 2^\alpha - 1]$, *in order to obtain* Y *when* $p(x)$ *is respectively evaluated at* $x_p - c, x_p + \delta_1 - c, x_p + \delta_2 - c, \ldots$, *where* $c \in \mathbb{Z}^+$, *is at least* $(1 - \frac{1}{d \cdot c})$.

Proof. The polynomial $p(x)$ evaluated at $(x_p - c)$ is

$$\begin{aligned} p(x_p - c) &= a_d \cdot (x_p - c)^d + a_{d-1} \cdot (x_p - c)^{d-1} + \ldots + a_0 \\ &= a_d \cdot {x_p}^d + (-{}^dC_1 \cdot c \cdot a_d + a_{d-1}) \cdot {x_p}^{d-1} + \ldots + \sum_{i=0}^{d} (-1)^i a_i \cdot c^i \end{aligned}$$

Let T_{d-1} be the coefficient of x^{d-1} in $p(x_p - c)$. $T_{d-1} > 0$ only when $a_{d-1} > d \cdot c \cdot a_d$. We look at the probability of a_{d-1} going beyond the coefficient range $[1, 2^\alpha - 1]$ *when* $a_{d-1} > d \cdot c \cdot a_d$. $P(a_{d-1} \geq 2^\alpha) \geq P(d \cdot c \cdot a_d \geq 2^\alpha) = P(a_d \geq \frac{2^\alpha}{d \cdot c})$. Since the coefficients are picked randomly from an uniform distribution and the coefficient range is $[1, 2^\alpha - 1]$, $P(a_{d-1} \geq 2^\alpha) \geq \frac{2^\alpha - 1 - (2^\alpha / d \cdot c)}{2^\alpha - 2} \geq 1 - \frac{1}{d \cdot c}$. □

From Theorem 3, we see that even for polynomials of small degree, say $d = 10$, in order for T_{d-1} to be positive, $a_{d-1} \geq 2^\alpha$ with a high probability which is a contradiction to our assumption that all the coefficients of $p(x)$ are within the given range for coefficients.

4.10.2 Evaluation at $(x_p + c)$

Theorem 4. *Given* $p(x) \in \mathbb{N}[x]$ *with* $p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$ *and* $Y = \{p(x_p), p(x_p + \delta_1), p(x_p + \delta_2), \ldots\}$. *The probability that at least one coefficient of* $p(x)$ *goes beyond the coefficient range of* $[1, 2^\alpha - 1]$, *in order to obtain* Y *when* $p(x)$ *is respectively evaluated at* $x_p + c, x_p + \delta_1 + c, x_p + \delta_2 + c, \ldots$, *where* $c \in \mathbb{Z}^+$, *is at* *least* $(1 - \frac{1}{d \cdot c})$.

Proof. The polynomial $p(x)$ evaluated at $(x_p + c)$ is

$$\begin{aligned} p(x_p + c) &= a_d \cdot (x_p + c)^d + a_{d-1} \cdot (x_p + c)^{d-1} + \ldots + a_0 \\ &= a_d \cdot {x_p}^d + ({}^dC_1 \cdot c \cdot a_d + a_{d-1}) \cdot {x_p}^{d-1} + \ldots + \sum_0^d a_i c^i \end{aligned}$$

Let T_{d-1} be the coefficient of x^{d-1} in $p(x_p + c)$.
$T_{d-1} = d \cdot c \cdot a_d + a_{d-1} \Rightarrow T_{d-1} \geq d \cdot c \cdot a_d$.
$P(T_{d-1} \geq 2^\alpha) \geq P(d \cdot c \cdot a_d \geq 2^\alpha) = P(a_d \geq \frac{2^\alpha}{d \cdot c})$.
Since the coefficients are picked randomly from an uniform distribution and the coefficient range is $[1, 2^\alpha - 1]$, $P(T_{d-1} \geq 2^\alpha) \geq \frac{2^\alpha - 1 - (2^\alpha / d \cdot c)}{2^\alpha - 2} \geq 1 - \frac{1}{d \cdot c}$. □

From Theorem 4, we find that $P(T_{d-1} \geq 2^\alpha)$ will be high even for small degree polynomials, say $d = 10$.

Remark. In our experiments we have been able to verify that the recovered polynomial matched the one used to obtain the initial set of outputs.

5 Experiments and Results

SageMath 8.6 [38] was used to implement the procedures mentioned in Section 4. Our source code is available at [18]. All our experiments were run on a Lenovo ThinkStation P920 workstation having a 2.2 GHz Intel®Xeon® processor with 20 cores. However, our implementation is not multi-threaded and hence did not explicitly exploit the parallelism available. We classify our experiments into two sets depending on how the input x_i values were sampled - either uniform randomly or come from a real dataset. The coefficients a_i were always uniform randomly chosen from the underlying set. The goal is to recover the inputs and the polynomial. The second set consists of using data available from the UCI Machine learning repository [14] which is a real-world hospital data obtained from a hospital in Caracas, Venezuela. For this case, we have chosen the degree of the polynomial $d = 9$ as in [23].

5.1 Experiments with Random Values

In our experiments with random values, the polynomial coefficients a_i were chosen uniformly random in the range $[1, 2^\alpha - 1]$, with $\alpha = 128$. We have run a number of experiments for different values of β and polynomial degrees. For β up to 28, where we use the sieve method to obtain factors $< 2^\beta$, we have run experiments for polynomials of degrees $d = 12, 24, 48$ and 96. For experiments with degrees $d = 12$ and 24, we also run the procedure *RecursiveConsistencyCheck* described in Section 4.3. For β values of 32, 64, where we first use sieving to find factors $\leq 2^{28}$ followed by the ECM factoring method for larger factors $< 2^\beta$, we have run experiments for polynomials of degrees 12 and 24. We randomly chose x_i values from $[0, 2^\beta - 1]$ and computed $n = 2d$ polynomial outputs. Table 1 gives the time taken for our algorithm (Algorithm 1) to run to completion. In each of the trials, our algorithm was successful in recovering the input polynomial and input values. The time taken for recovery is given in seconds and is averaged over 5 runs in each case. As described in Section 4.1, we use SageMath function `ECM.find_factor()` to find factors. This likely returns the next smallest factor (prime in most cases). We need to extract all factors $< 2^\beta$, hence we call this function iteratively. At each invocation of this function, we start a timer with a timeout, obtained empirically, such that its value is likely sufficient to obtain the biggest prime factor $< 2^\beta$. The iteration ends when we hit the timeout. Table 2 gives timeout values and the percentage of number of times where a suitable polynomial was not recovered, in about 200 trials for different values of β. It is to be noted that there is a direct correlation between failure of polynomial recovery and the timeout occurrence before obtaining all factors $< 2^\beta$ using `ECM.find_factor()`. Based on this, we have set a timeout value of 120 seconds in all our experiments for $\beta \leq 64$.

In the experiments, n many polynomial output values were input to our algorithm. The choice of $n = 2d$ was based on observations from the experiments. For the instances where we used parameters mentioned in [23], we could bound the number of divisors to less than 32 thereby making the search space less than 32^{10}. We then used the divisor set and $(d+1)$ polynomial outputs to compute a possible polynomial using Lagrange interpolation, which we then used to verify successfully against the remaining $(n-d-1)$ output values. We note that our search space is significantly less than the estimate of 2^{160} in [23]. For instances with larger parameters, we could bound the number of divisors to less than 200. In the Procedure `CheckConsistencyInColumn` described in Section 4.2, as soon as any divisor turns out to not belong to the consistent set, it is discarded by setting to 0, thereby reducing the number of divisors to be checked. Also, it looks like many further optimisation could be done to reduce the search space.

α (bits)	β (bits)	Degree	Time to recover polynomial and x_1 (seconds)
128	24	12	10
128	24	24	42
128	24	48	293
128	24	96	2299
128	28	12	129
128	28	24	748
128	28	48	5084
128	28	96	38810
128	32	12	7382
128	32	24	35387
128	64	12	12190
128	64	24	48758

Table 1: Run times for polynomial reconstruction for different polynomial degrees and β values

β (bits)	Timeout (seconds)	Percentage of tests where a suitable polynomial was not recovered
64	5	95%
64	10	85%
64	30	10%
64	60	0
64	120	0
80	60	25%
80	120	10%
80	180	8%
96	60	40%
96	120	37%
96	180	38%

Table 2: Percentage of tests where a suitable polynomial was not recovered in about 200 trials where `ECM.find_factor()` was used to obtain factors. Timeout indicates for how long the `ECM.find_factor()` was allowed to run for each factor before aborting.

5.2 Experiments with Real World Data

We used the cervical cancer (risk factors) data set, same as one of the datasets used by [23], also available from the UCI Machine learning repository [14]. This data set consists of information pertaining to 858 patients, each consisting of 32 attributes comprising of demographic information, habits and historic medical records. The dataset had a few missing values due to privacy concerns and these were set to 0. Values with fractional part were rounded off to the nearest integer. We repeated the experiment with different random degree $d = 9$ polynomials and were able to recover the original polynomial successfully. We also tested with 16, 20, 24 and 32 bit values of α and have tabulated the time taken by SageMath to compute the polynomial in each of the cases. $\beta = 24$ was sufficient to encode this data. Time for execution is given in seconds and is averaged over 5 runs in each case.

α (bits)	β (bits)	Time to recover polynomial and x_1 (seconds)
16	24	3.3
20	24	3.5
24	24	3.6
32	24	3.9

Table 3: Run times for polynomial reconstruction for a real world data with degree 9 polynomial.

Our results invalidate the security claims in [23, Theorem 4.2] regarding the leakage profile for Server B. We use the parameters suggested in [23, Section 4.2], i.e., $d = 9$, α =16 and the (squared plaintext) distances are in the range $[0, 2^{\beta} - 1]$, where, $\beta = 24$. For the parameters mentioned there, with only $n = 20$ output values, we could recover the coefficients of the polynomial in a few seconds as given in Table 3.

Because of the random re-ordering of the distances, Server B will not learn the exact distance of the query point to a specified point (say the i^{th} point in the original order). Nevertheless, in many real world scenarios the data set is publicly available and this, and perhaps other auxilliary information, may potentially be used in combination with our results to leak information about the query point.

6 Attack on the Secure k-NN protocol in the Noisy Setting

In this section, we give another attack on the protocol of [23] if one tries to overcome our attack from Section 4 by perturbing the polynomial outputs by adding noisy error terms. This modified protocol is not mentioned in [23] but we consider it here for completeness.

In the original solution given in [23], in order to hide the Euclidean distance values, Server A chooses a monotonic polynomial and homomorphically evaluates this polynomial on its computed distances and permutes the order before sending them to Server B. Now, instead of sending these (encrypted) polynomial outputs as it is, if they are perturbed with some noise such that the ordering is still maintained, it will make our attack in Section 4 unsuccessful in recovering the polynomial or the inputs, as the attack relies on the exact difference of the polynomial outputs. It is easy to see that the error value can only be as large as the sum of all the coefficients except the constant term. Let $p(x) = a_0 + a_1 \cdot x + \ldots + a_d \cdot x^d$ be the chosen monotonic polynomial, then, $p(0) = a_0$, $p(1) = a_0 + a_1 + \ldots + a_d$ and the maximum value of the added noise needs to be less than $(p(1) - p(0))$ so as to maintain the original ordering of polynomial outputs, meaning the perturbation error may only be chosen from the set $[0, \ldots, (a_1 + \ldots + a_d)]$. This safe choice of the error term is due to the fact that the polynomial output values are encrypted and hence it is not possible for the Server A to inspect the value and accordingly choose the error term. The range of perturbation error terms still depends on the size of the coefficient space that can potentially be very large (unlike the plaintext space as assumed).

Theorem 5. *Given a set of noisy outputs of a degree-d monotonic integer polynomial where the outputs are perturbed with some random noise from the range $[0, \sum_{k=1}^{d} a_k]$, then the Server B will be able to leak ratios of the inputs.*

Proof. Let two of the values that the Server B obtains after decryption be $F(x_i) = p(x_i) + e_i$ and $F(x_j) = p(x_j) + e_j$, where e_i and e_j are the random error terms such that $0 \leq e_i, e_j < \sum_{k=1}^{d} a_k$, $1 \leq a_k < 2^{\alpha}$, and $F(x_i), F(x_j) > 0$.
Consider the ratio $F(x_i)/F(x_j)$ with $0 \leq x_j \leq x_i < 2^{\beta}$:

$$\frac{F(x_i)}{F(x_j)} = \frac{(\sum_{k=0}^{d} a_k) + a_1 \cdot x_i + \ldots + a_d \cdot x_i^d}{(\sum_{k=0}^{d} a_k) + a_1 \cdot x_j + \ldots + a_d \cdot x_j^d}. \tag{13}$$

When x_i and x_j are sufficiently large we obtain that the ratio in (13) is approximately close to $(x_i/x_j)^d$. □

The attack presented in Section 4 will not work in this new setting, where the inputs are perturbed using bounded random noise, because in the attack we rely on the exact differences of the polynomial outputs. In the noisy setting, Theorem 6 shows that it is still possible to leak ratios of the inputs to the Server B by taking the d^{th} root of $(x_i/x_j)^d$, although recovering the exact values (even up to an affine scaling) may be challenging. But still a lot more information about the inputs is leaked than just a single bit. Note also that if the error terms e_k were not significantly smaller than the leading terms (which, fortunately, is *not* the case), then we would not be able to recover the ratios.

7 Conclusion and Future Work

In this paper, we give an attack on the protocol of [23] for securely computing k-NN on SHE encrypted data. This attack is based on our algorithm for integer polynomial reconstruction given only the integer outputs. While, by just using the outputs, it is not possible to accurately determine the coefficients or the inputs, we show that we can feasibly recover the inputs (up to an affine scaling) when the number of outputs is much bigger than the degree of the polynomial. Our experiments were conducted both on uniformly randomly selected values as well as a real-world dataset. Since many of the datasets are available in the public domain it may possible for an adversary to derive more information about the exact values using our method together with some other available metadata.

Our method for polynomial reconstruction runs in exponential time in plaintext space β and in degree d of the chosen polynomial. In many real-world scenarios both these parameters will be small. Future work can look at a lower bound analysis of the time required for this polynomial reconstruction problem. It can also look at relaxing the practical limits on α and β. The requirement of having the integer polynomial to be monotonic was needed by [23] since they needed the polynomial outputs to follow the input ordering to be able to sort the inputs. Relaxing this requirement from the view point of polynomial recovery can be investigated as part of a future work. Finally, an FHE solution that can perform efficient sorting and searching on large datasets would eliminate the need for service providers to be entrusted with decryption keys, thereby providing a more secure cloud computation environment.

Acknowledgements

This work was funded partly by Sonata Software Limited, Bengaluru, India, partly by the INSPIRE Faculty Award (DST, Govt. of India) and the Infosys Foundation Career Development Chair Professorship grant for Srinivas Vivek and partly by the Centre for Internet of Ethical Things (IIIT Bangalore). We thank Debdeep Mukhopadhyay and Sikhar Patranabis for helpful discussions. We also thank V. N. Muralidhara for pointing out the literature on the d-MST problem. We thank the anonymous reviewer for the invaluable comments and suggestions, which helped us improve the manuscript.

References

[1] Y. Aono, T. Hayashi, L. T. Phong, and L. Wang. Scalable and secure logistic regression via homomorphic encryption. In E. Bertino, R. Sandhu, and A. Pretschner, editors, *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 142–144. ACM, 2016.

[2] E. Berlekamp. Algebraic Coding Theory. *McGraw-Hill, New York*, Volume 8, 1968.

[3] D. Bleichenbacher and P. Q. Nguyen. Noisy polynomial interpolation and noisy chinese remaindering. In B. Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 53–69, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[4] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman. Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 563–594, Sofia, Bulgaria, Apr. 26–30, 2015. Springer, Heidelberg, Germany.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[7] G. S. Çetin, Y. Doröz, B. Sunar, and E. Savas. Depth Optimized Efficient Homomorphic Sorting. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America*, volume 9230 of *Lecture Notes in Computer Science*, pages 61–80, Guadalajara, Mexico, Aug. 23–26, 2015. Springer, Heidelberg, Germany.

[8] G. S. Çetin and B. Sunar. Homomorphic rank sort using surrogate polynomials. In T. Lange and O. Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 311–326, Cham, 2019. Springer International Publishing.

[9] A. Chatterjee and I. Sengupta. Searching and sorting of fully homomorphic encrypted data on cloud. *IACR Cryptol. ePrint Arch.*, 2015:981, 2015.

[10] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song. Homomorphic encryption for arithmetic of approximate numbers. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, jan 2020.

[12] S. Choi, G. Ghinita, H.-S. Lim, and E. Bertino. Secure kNN Query Processing in Untrusted Cloud Environments. *IEEE Transactions on Knowledge and Data Engineering*, 26:2818–2831, 2014.

[13] A. Costache, N. P. Smart, S. Vivek, and A. Waller. Fixed-point arithmetic in SHE schemes. In R. Avanzi and H. M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 401–422, St. John's, NL, Canada, Aug. 10–12, 2016. Springer, Heidelberg, Germany.

[14] D. Dua and C. Graff. UCI Machine Learning Repository, 2017.

[15] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-Nearest Neighbor Query over Encrypted Data in Outsourced Environments. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 664–675, 2014.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[17] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Heidelberg, Germany.

[18] GitHub. SAGE code for polynomial recovery. https://github.com/shyamsmurthy/knn_polynomial_recovery, 2019. Last accessed: March 21, 2021.

[19] O. Goldreich, R. Rubinfeld, and M. Sudan. Learning Polynomials with Queries: The Highly Noisy Case. *SIAM Journal on Discrete Mathematics*, 13(4):535–570, 2000.

[20] P. Gopalan, S. Khot, and R. Saket. Hardness of reconstructing multivariate polynomials over finite fields. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pages 349–359, 2007.

[21] V. Guruswami and M. Sudan. Improved Decoding of Reed-Solomon and Algebraic-Geometry Codes. *IEEE Transactions on Information Theory*, 45(6):1757–1767, Sep. 1999.

[22] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *ArXiv*, abs/1711.10677, 2017.

[23] M. Kesarwani, A. Kaul, P. Naldurg, S. Patranabis, G. Singh, S. Mehta, and D. Mukhopadhyay. Efficient Secure k-Nearest Neighbours over Encrypted Data. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 564–575, 2018.

[24] A. Kiayias and M. Yung. Directions in polynomial reconstruction based cryptography. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 87:978–985, 2004.

[25] A. Kobel, F. Rouillier, and M. Sagraloff. Computing real roots of real polynomials ... and now for real! In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 303–310, New York, NY, USA, 2016.

[26] M. Kordos, M. Blachnik, and D. Strzempa. Do we need whatever more than k-nn? In L. Rutkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, and J. M. Zurada, editors, *Artificial Intelligence and Soft Computing*, pages 414–421, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[27] D. Kumaraswamy, S. Murthy, and S. Vivek. Revisiting driver anonymity in oride. In *Selected Areas in Cryptography: 28th International Conference, Virtual Event, September 29 – October 1, 2021, Revised Selected Papers*, page 25–46, Berlin, Heidelberg, 2021. Springer-Verlag.

[28] K. Lewi and D. J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1167–1178, Vienna, Austria, Oct. 24–28, 2016. ACM Press.

[29] B. Li and D. Micciancio. On the security of homomorphic encryption on approximate numbers. In A. Canteaut and F.-X. Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 648–677, Cham, 2021. Springer International Publishing.

[30] Y. Luo, X. Jia, S. Fu, and M. Xu. pRide: Privacy-Preserving Ride Matching Over Road Networks for Online Ride-Hailing Service. *IEEE Trans. Information Forensics and Security*, 14(7):1791–1802, 2019.

[31] S. Murthy and S. Vivek. Cryptanalysis of a protocol for efficient sorting on SHE encrypted data. In M. Albrecht, editor, *Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings*, volume 11929 of *Lecture Notes in Computer Science*, pages 278–294. Springer, 2019.

[32] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In J. S. Vitter, L. L. Larmore, and F. T. Leighton, editors, *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 245–254. ACM, 1999.

[33] U. Pujianto, A. P. Wibawa, M. I. Akbar, et al. K-nearest neighbor (k-nn) based missing data imputation. In *2019 5th International Conference on Science in Information Technology (ICSITech)*, pages 83–88. IEEE, 2019.

[34] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning Trees - Short or Small. *SIAM J. Discrete Math.*, 9(2):178–200, 1996.

[35] Sage 9.1 Reference Manual. The elliptic curve factorization method. https://doc.sagemath.org/html/en/reference/interfaces/sage/interfaces/ecm.html, 2019. Last accessed: June 01, 2020.

[36] M. Sagraloff. When newton meets descartes: A simple and fast algorithm to isolate the real roots of a polynomial. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC '12, pages 297–304, New York, NY, USA, 2012.

[37] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar. Compacting Privacy-Preserving k-Nearest Neighbor Search using Logic Synthesis. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[38] W. Stein et al. *Sage Mathematics Software (Version 8.6)*. The Sage Development Team, 2019. http://www.sagemath.org.

[39] M. Sudan. List decoding: Algorithms and applications. *SIGACT News*, 31(1):16–27, Mar. 2000.

[40] T. Tao. Blog: The divisor bound, 2008. Available at https://terrytao.wordpress.com/2008/09/23/the-divisor-bound/. Last accessed: July 19, 2021.

[41] H. Yu, X. Jia, H. Zhang, X. Yu, and J. Shu. Psride: Privacy-preserving shared ride matching for online ride hailing systems. *IEEE Transactions on Dependable and Secure Computing*, 18(3):1425–1440, 2021.

A Polynomial Reconstruction Algorithm

Algorithm 1: Integer Polynomial Reconstruction From Only the Integer Outputs

```
Procedure Main($\{p(x_1), \ldots, p(x_n)\}$, $\psi$) :
    /* Inputs : $p(x_i)$ are the polynomial outputs in the ascending order, $\psi$ is
       the bound on the number of divisors */
    /* Outputs : All recovered polynomial $p(x)$ and corresponding $x_1$ */
    $D$ = GuessTheDifference($\{p(x_1), \ldots, p(x_n)\}$, $\psi$)
    $D$ = CheckConsistencyInColumn($D$)
    $C$, $tuple\_count$ = FormConsistentTuple(1)
    for $i = 1$ to $tuple\_count$ do
        /* NOTE: Optional loop and function RecursiveConsistencyCheck()
           enabled when executed for small polynomial degree */
        $discardFlag$ = RecursiveConsistencyCheck(1, $C[i]$)
        if $discardFlag$ is TRUE then
            discard $C[i]$;
            $tuple\_count$ $-$= 1
        end
    end
    $\chi[\,]$ := Column 1 of Matrix $Y$
    for $i = 1$ to $tuple\_count$ do
        $L$ = ApplyLagrangeInterpolation($C[i]$, $\chi$)
        $x_1$, $p$ = RecoverPossibleXValuesAndPoly($L$)
        if $x_1$ is not failure then
            Compute roots of $p(x) = y_i$, $(d + 1) < i \leq n$
                    /* For the other $(n - d - 1)$ outputs */
            if No positive integer root $< 2^\beta$ then
                Discard current tuple, move to next one
            end
            Output Polynomial $p$ and $x_1$
        end
    end
    return
```

```
Procedure GuessTheDifference({$p(x_1), \ldots, p(x_n)$}, $\psi$) :
    $count = 0$
    for $i = 2$ to $n$ do
        $y\_val = p(x_i)$
        for $j = 1$ to $(i - 1)$ do
            $y_{i,j} = (y\_val - p(x_j))$ /* $p(x_i) > p(x_j)$ */
            Use sieve method to obtain all the (positive) divisors of $y_{i,j} < 2^\beta$
            $D_{i,j}$ := Set of all divisors of $y_{i,j} < 2^\beta$
            if $|D_{i,j}| > \psi$ then
                /* Discard current ($y_i$, $D_i$) and pick next output */
                $i$ += 1; $j$ = 1;
                $y\_val = p(x_i)$
                continue /* Restart inner loop */
            end
            $count$ += 1
            if $count$ == $d + 1$ then
                return $D$
            end
        end
    end
    return $D$ /* Matrix of divisors set */
```

```
Procedure CheckConsistencyInColumn(Matrix D) :
    for $col = 1$ to $(d-1)$ do
        $top\_row = col + 1$
        for $i = (top\_row + 1)$ to $(d+1)$ do
            $c\_flag = 0$
            for $j = top\_row$ to $(i-1)$ do
                Iterate over all $f \neq 0 \in D_{i,col}$
                    Iterate over all $g \neq 0 \in D_{j,col}$
                        if $(f-g) \in D_{i,col+1}$ then $c\_flag = 1$
                    if $c\_flag == 0$ then Set $f = 0$ in $D_{i,col}$
            end
        end
        /* Check consistency of topmost non-zero element in Column */
        forall $g \neq 0 \in D_{top\_row,col}$ do
            for $i = (top\_row + 1)$ to $(d+1)$ do
                $c\_flag = 0$
                Iterate over all $f \neq 0 \in D_{i,col}$
                    if $(f-g) \in D_{i,col+1}$ then $c\_flag = 1$
                if $c\_flag == 0$ then Set $g = 0$ in $D_{top\_row,col}$
            end
        end
        return $D$ /* Matrix of divisors with consistent elements in each
          column */
    end

Procedure FormConsistentTuple($i$ : Column id of $D$ Matrix) :
    $tuple\_count = 0$
    Initialize Array $C[\,][\,]$ to zero
    /* Examine elements in column $i$ of Matrix $D$ */
    while iterate over each non-zero element $e$ in $D_{i+1,i}$ do
        $tuple\_count += 1$
        Append $e$ to $C[tuple\_count]$
        for $j = (i+1)$ to $(d+1)$ do
            Find the element $f$ consistent with $e$ in $D_{j,i}$
            Append $f$ to $C[tuple\_count]$
        end
    end
    return $C$, $tuple\_count$
```

```
Procedure RecursiveConsistencyCheck(Col_id, Divisor Tuple $\tau$) :
    $ret$ = FALSE
    if $Col\_id <$ degree $d$ then
        $y\_val$ = $Y[Col\_id + 1][Col\_id]$
        for $i$ = ($Col\_id$ + 1) to $d$ do
            $New\_y$ = ($Y[i][Col\_id] - y\_val$) / $\tau[i]$
            $Div\_new$ := Divisors of $New\_y < 2^{\beta}$
            $Div\_int$ := $Div\_new \cap D_{i\,Col\_id}$
            $D_{i\,Col\_id}$ := $Div\_int$
            $Y[i][Col\_id]$ = $New\_y$
            $\tau$, $tuple\_count$ = FormConsistentTuple($i$)
            if $tuple\_count$ == 0 then
                /* Tuple $\tau$ leads to an inconsistent state, can be discarded */
                return TRUE
            end
        end
        $Col\_id$ += 1
        $ret$ = RecursiveConsistencyCheck($Col\_id$, $\tau$) return $ret$
    end
    else
        return $ret$
    end

Procedure ApplyLagrangeInterpolation(CT: tuple of $d + 1$
  differences, $\chi$) :
    Declare $x_1$ as a variable
    for $i = 0$ to $d$ do
        /* Express each difference in terms of $x_1$ */
        $\delta_i$ = $x_1$ + $CT[i + 1] - CT[1]$
    end
    $\delta$ = $[\delta_0, \delta_1, \ldots, \delta_d]$
    Apply Lagrange interpolation on ($\delta$, $\chi$) to get a degree $d$ polynomial $L$ in
      $x$.
    The coefficients $L_0, L_1, \ldots, L_d$ of $L$ are polynomials in $x_1$, such that the
      coefficient of $x$ is a degree $d$ polynomial in $x_1$ and that of $x^d$ is a
      constant integer.
    return $L$
```

```
Procedure RecoverPossibleXValuesAndPoly($L$)
    /* Coefficients $L_0, \ldots, L_d$ of $L$ are polynomials */
    for $i = 0$ to $(d-1)$ do
        $R_{i,0}$ := All real roots of $L_i(x) = 0$ in $[0, 2^\beta - 1]$
        $R_{i,\alpha}$ := All real roots of $L_i(x) = 2^\alpha - 1$ in $[0, 2^\beta - 1]$
        Sort the elements of $R_i = R_{i,0} \cup R_{i,\alpha}$
        $S_i = \phi$; Let $a$ be first element of $R_i$
        if $0 \notin R_i$ and $0 < L_i(a/2) < 2^\alpha - 1$ then
            $S_i = S_i \cup \{[0, \lfloor a \rfloor]\}$
        for every pair of consecutive elements $a, b$ in $R_i$ do
            if $a$ is an integer then
                $S_i = S_i \cup \{[a, a]\}$
            if $a \in R_{i,0}$ and $b \in R_{i,0}$ and $0 < L_i(\frac{a+b}{2}) < 2^\alpha - 1$ then
                $S_i = S_i \cup \{[\lceil a \rceil, \lfloor b \rfloor]\}$
            if $a \in R_{i,0}$ and $b \in R_{i,\alpha}$ then
                $S_i = S_i \cup \{[\lceil a \rceil, \lfloor b \rfloor]\}$
            if $a \in R_{i,\alpha}$ and $b \in R_{i,0}$ then
                $S_i = S_i \cup \{[\lceil a \rceil, \lfloor b \rfloor]\}$
            if $a \in R_{i,\alpha}$ and $b \in R_{i,\alpha}$ and $0 < L_i(\frac{a+b}{2}) < 2^\alpha - 1$ then
                $S_i = S_i \cup \{[\lceil a \rceil, \lfloor b \rfloor]\}$
            if $b$ is an integer then
                $S_i = S_i \cup \{[b, b]\}$
        end
        Let $b$ be the last element of $R_i$.
        if $2^\beta - 1 \notin R_i$ and $0 < L_i(\frac{b+2^\beta-1}{2}) < 2^\alpha - 1$ then
            $S_i = S_i \cup \{[\lceil b \rceil, 2^\beta - 1]\}$
    end
    while every $S_i$ is non-empty do
        Let $S_{i,j}$ denote the $j$-th interval in $S_i$.
        Find $k$ such that $S_k$ has the least end-point in its first interval.
        Compute $I = S_{0,1} \cup S_{1,1} \cup \cdots \cup S_{d-1,1}$
        if $I$ is non-empty and $I$ contains an integer $x$ in $[0, 2^\beta - 1]$ then
            /* Found $x$ */
            Compute polynomial $p$ whose coefficients are $L_0(x), \ldots, L_d(x)$
            return $x, p$
        end
        else
            Remove $S_{k,1}$ from $S_k$.
        end
    end
    return failure
```