IBGA Algorithm

Mohamed Emary Mohamed Abdelfattah Abdelfattah Zakaria

Shrouk Elsayed Dalia Abdallah Sara Reda

December 19, 2023

1 IBGA Algorithm Steps

1.1 Encryption Process

1. Specify the degree of the polynomial d that will be used as a shared key between the two parties. It should be an odd degree till 13th order
2. Generate $d + 1$ random data points.
3. Get the polynomial P_1 that passes through these points using **Lagrange Interpolation**.
4. Convert the text you want to encrypt into a decimal number t using **UTF-8** encoding.
5. Subtract t from the polynomial constant term to get the new polynomial. So for example if the polynomial is $P_1(x) = x^2 + 3x + 5$ and $t = 10$ then the new polynomial will be $P_2(x) = x^2 + 3x + 5 - 10 = x^2 + 3x - 5$.
6. **Find the root** of P_2 using any of the root finding methods like Bisecting Method, Newton Method, Secant Method, or Hybrid Method and save it into r.
7. The root r will be the encrypted message that will be sent to the receiver.

1.1.1 N_{th} Root & Initial Guesses

If you noticed in step 6 we used the root finding method to find the root of the polynomial. But to apply any of these methods we need to know some initial guesses. So how can we find these initial guesses?

1. Specify the degree of the polynomial.
2. Based on the degree of the polynomial you specified, you will need to find the N_{th} root of the constant term of the polynomial.

$$\text{nth\_root} = \begin{cases} \frac{\text{degree}}{2} & \text{if } 1 \leq \text{degree} \leq 10 \\ \frac{\text{degree}}{3} & \text{if } 10 \leq \text{degree} \leq 15 \\ \text{degree} & \text{otherwise} \end{cases} \tag{1}$$

3. Then you get the N$_{\text{th}}$ root of the constant term of the polynomial. So for example if the polynomial is $P_2(x) = x^5 + 3x - 5$ then the N$_{\text{th}}$ root will be $\frac{5}{2} = 2.5$ since the degree of the polynomial is 5 so we have to divide it by 2. Now we get the N$_{\text{th}}$ root of the constant term of that polynomial which is $\sqrt[2.5]{-5}$.
4. Now when we get the result of that root let it be $\phi = \sqrt[2.5]{-5}$, we will use it to find the initial guesses for the root finding method that we will use and it will be as follows:
    1. With **Bisection Method** we need two initial points as it is a bracketing method so we will use: $[\text{round}(-\phi), \text{round}(\phi)]$. For example if $\phi = 2.5$ then the initial points will be $[-3, 3]$.
    2. With **Newton Method** we need one initial point as it's an open method so we will use: $\text{round}(\phi)$. For example if $\phi = 2.5$ then the initial point will be 3.
    3. With **Secant Method** we need two initial points as it is a bracketing method too so we will use: $[\lfloor\phi\rfloor, \lceil\phi\rceil]$. For example if $\phi = 2.5$ then the initial points will be $[2, 3]$.

$$\text{Initial guesses} = \begin{cases} [\text{round}(-\phi), \text{round}(\phi)] & \text{for Bisection Method} \\ \text{round}(\phi) & \text{for Newton Method} \\ [\lfloor\phi\rfloor, \lceil\phi\rceil] & \text{for Secant Method} \end{cases} \tag{2}$$

1.2 Decryption Process

1. You as a receiver should have the same polynomial P_1 that was used to encrypt the message (our algorithm is a symmetric algorithm so both parties should have the same polynomial). You will also get the encrypted message r which is the root of the polynomial P_2.
2. Now when you substitute r in the polynomial P_1 you will get the value of t which is the decimal number of the text that was encrypted.
3. Convert t back to the text using **UTF-8** decoding to get the original text back.

2 UTF-8 & ASCII Encoding

UTF-8 is a variable-length character encoding standard used for electronic communication. Defined by the Unicode Standard, the name is derived from Unicode Transformation Format – 8-bit. *Variable-length* character encoding means that different characters can take up a different number of bytes. In the context of UTF-8, it means that a single character can be represented using anywhere from 1 to 4 bytes.

For example, standard ASCII characters (like ‘a’, ‘b’, ‘1’, etc.) only require 1 byte in UTF-8. However, other characters, such as many emoji, accented letters, and characters from non-Latin scripts, require more bytes. This flexibility allows UTF-8 to efficiently represent the wide range of characters defined in the Unicode standard, while maintaining backward compatibility with ASCII.

In UTF-8 encoding, a character is not always 1 byte. It can be anywhere from 1 to 4 bytes depending on the specific character. This is what allows UTF-8 to represent a wide range of characters from various languages and symbol sets, including those outside the ASCII range.

ASCII is limited compared to UTF-8 ASCII character encoding uses 7 bits for each character. This means that it can only represent 128 characters ($2^7 = 128$) These characters include the English alphabet (in both upper and lower cases), digits, punctuation marks, and control characters. The first 128 characters in Unicode are the same as ASCII, so UTF-8 is backwards compatible with ASCII. This means that any ASCII text is also valid UTF-8 text.

However, ASCII is often stored in 8-bit bytes for convenience, with the extra bit typically used as a parity bit for error checking in some systems, or left as 0 in systems that don't require error checking. This 8-bit version is often referred to as "extended ASCII" or "ASCII-8", but it's not a standard and the extra bit's usage can vary between systems.

3 Some Notes

The changes we will make to the algorithm will include replacing both bisection and newton methods with the hybrid method to achieve a faster performance.

We may also (still not sure) do the following:

1. Use another algorithm faster than Lagrange Interpolation to find the polynomial. Newton Interpolation is a good candidate.
2. Use a changing number of rotations agreed upon by both parties instead of just 1 rotation. The number of rotations will not exceed the degree of the polynomial as we rotate in a circular way so it's just useless to rotate more than the degree of the polynomial, one way to ensure this is to use mod % operation.

4 Steps Flowcharts

4.1 Encryption Process

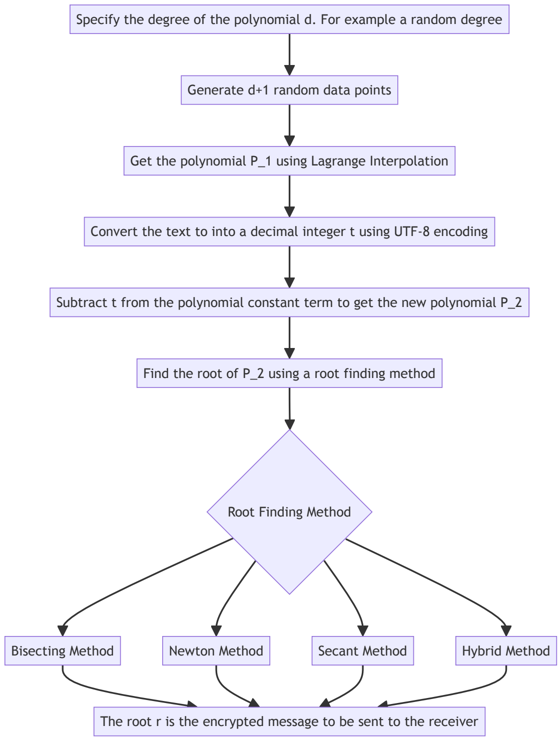
Specify the degree of the polynomial d. For example a random degree
Generate d+1 random data points
Get the polynomial P_1 using Lagrange Interpolation
Convert the text to into a decimal integer t using UTF-8 encoding
Subtract t from the polynomial constant term to get the new polynomial P_2
Find the root of P_2 using a root finding method
Root Finding Method
Bisecting Method
Newton Method
Secant Method
Hybrid Method
The root r is the encrypted message to be sent to the receiver

Figure 1: Encryption Steps

4.2 Decryption Process

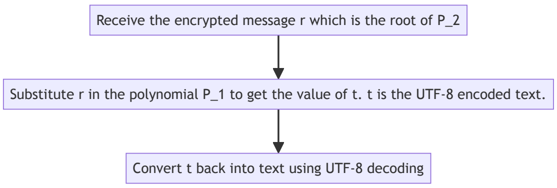
Receive the encrypted message r which is the root of P_2
Substitute r in the polynomial P_1 to get the value of t. t is the UTF-8 encoded text.
Convert t back into text using UTF-8 decoding

Figure 2: Decryption Steps