Assignment 1 MIFE scheme

Joona Korkeamäki

Introduction

For my implementation of the MIFE scheme, I chose the GoFe DDHMulti scheme, because I determined It most closely follows the ElGamal scheme detailed in the paper. My experiments were conducted on a Windows 10, 64-bit PC with AMD Ryzen 5 5600X 3.70 GHz processor and 32GB of RAM. Only External library I used was GoFe, which had some dependencies on its own. Other libraries were Native to GoLang. The Go version used was 1.21.3.

I tried to follow the experiments detailed in the paper closely and I got some similar results and some that differed. Although the most important result I observed was that using differential privacy on top of encryption did not add virtually any time to the overall processing time of the scheme. However, the results of my experiments differed significantly on the decryption times, the reasons for why I will speculate in the appropriate section.

Tree Generation

Following the paper in question I implemented the binary tree as a slice, which is comparable to a list in Python. The slice contained Node structs that in turn contain the range of values [a,b] that belong in the node, and the occurrences of those values in the dataset. The ranges were generated based on the min and max values of the dataset. The root node having a range of [min,max] and the leaf nodes a range determined by the interval calculated by $\frac{max-min}{n}$, where n is the total number of leaf nodes. My experiments followed the same binary tree sizes as in the paper, so 63,127,255,511,1023 total nodes.

Similarly, to the results in the paper, I observed that the tree generation times were insignificant compared to the total run time of my experiments.

Dataset Size	Number Of Leaves	Average Time
100	32	62.053µs
100	64	62.052µs
100	128	61.052µs
100	256	67.059µs
100	512	80.067µs
100	1024	100.088µs
500	32	94.082µs
500	64	93.079µs
500	128	96.083µs
500	256	107.091µs
500	512	118.104µs
500	1024	149.126µs
1000	32	140.118µs
1000	64	153.133µs
1000	128	147.125µs
1000	256	155.134µs
1000	512	181.156µs
1000	1024	199.168µs
10000	32	1.167004ms
10000	64	1.231058ms
10000	128	1.294115ms
10000	256	1.362171ms
10000	512	1.476271ms
10000	1024	1.565345ms

Figure 1: Time required to generate and populate the tree following Table 1. from the paper. One thing of note here is that my timers seem to go off only on 500µs intervals, although the timers are supposed to be accurate to nanoseconds. It might be OS issue or misuse of the timers on my part. However, I got around the issue by providing averages over 500 rounds

Encryption and Noise

Following the steps described in the paper I then added noise to all the node values in the binary tree, and then generated public/private keypairs for all the nodes, and then encrypted them with their corresponding keys. Adding Laplacian noise to the nodes was even faster than the tree generation. Although I still had the problem with 500µs increments, so most of the time values for adding noise were 0s and sometimes up to ~500µs. I still got quite nice averages over 50 rounds, that might show closer to the actual values. The averages show that the time it takes to add noise is completely insignificant compared to the total processing time. They still range from 0ms-0.07ms when the total time ranges from 77ms-2400ms.

Similarly, to the paper I'm under the impression that GoFe also samples all the subkeys from the same group G, using the same generator g, because the keys are derived from the master key. This would in turn accelerate keygen times on my experiments. My results indeed show that the keygen takes roughly half the time of encryption and they both seem to be O(N) operations. Total processing time seems to be O(N) also.

Number Of Nodes	Tree Generation	Laplacian Noise	Key Generation	Encryption Time	Total Time
63	1.171064ms	0s	24.951616ms	50.443208ms	76.59591ms
127	1.281098ms	10.016µs	50.103156ms	101.68745ms	153.08172ms
255	1.391116ms	20.018µs	100.720846ms	204.341776ms	306.513784ms
511	1.541262ms	10.008µs	201.394138ms	408.720956ms	611.696386ms
1023	1.591324ms	10.01µs	405.81939ms	819.324932ms	1.226825736s
2047	1.751554ms	50.016µs	809.896966ms	1.640731888s	2.45253051s

figure 2: Average processing time for all setup functions for the 10 000 dataset, similar to table 2 in the paper. While on a different scale, my observations seem to follow the same trend as the ones in the paper.

Functional Decryption Key Generation

Here is where I think my observations differed the most from the paper. I experimented with two kinds of queries, both of which were of the form “How many values lie in the interval I [a,b]”, similar to that of in the paper. The first query was a speedy one, where I would search for the single node in that encapsulates the interval I in its entirety, that is furthest away from the root. Then I would only need to decrypt that value.

The other type of query was an accurate one, where I would search for all the leaf nodes that encapsulate the interval I, and only the leaves. this would often result in a query where I would need to sum more than half of the leaf nodes. At first, I separated the time to make the query, generate the functional decryption key and the decryption time, but the query and key generation were so fast compared to the decryption I decided to just lump them together.

The values a and b for the query were chosen at random between min and max of the dataset. This resulted in various size intervals that probably skewed the results a bit, since I only ran the decryption 50 times to get the averages.

The sum of these encrypted values would be computed as inner product of the wanted values and a y vector consisting only of values 1, so that the inner product calculation would result to a summation of the values.

The times on the fast query approach seem too fast to calculate consistently, which is no wonder since only one value is decrypted, which kind of defeats the whole purpose of a multi input functional encryption scheme.

The times for the accurate query however seem slower that the ones depicted on the paper. This raises some questions to me because my results have generally been faster up to this point. roughly half of the values in the tree are used for functional decryption key generation due to the random sampling of the queries, but still my times are much slower than even the 0.119ms time to generate a key as the sum of 1000 private keys mentioned in the paper.

I tried another approach for the decryption function, where I would create a functional decryption key that calculates the inner product of all ciphers in the tree, but this time the y vector would only have 1’s, on the indexes of ciphers I wanted to include to achieve the same result as my first approach. During testing this didn’t seem to make the process any faster, so I opted for the original approach.

I can come up with two reasons as to why my results are slower. First one is that I have done something inefficiently or wrong, resulting in slower process. I tested the correctness of the

decryption results, so I couldn't have done anything too wrong. I also have very little experience with Go as this is my second time using, tutorial 7 being the first, so it is possible I have caused the inefficiency with bad code. The second reason is that GoFe's implementation is simply slower or less efficient than the one implemented in the paper. And the true reason could be a bit of both or neither.

Number Of Nodes	Query Time	KeyGen Fast	Decryption Fast	KeyGen Accurate	Decryption Accurate
63	0s	0s	25.852296ms	18.642µs	237.896034ms
127	24.6µs	0s	17.271336ms	4.776µs	477.815354ms
255	20.018µs	0s	22.759582ms	50.036µs	1.03604156s
511	20.012µs	0s	22.270588ms	120.116µs	2.004503512s
1023	70.066µs	0s	18.455942ms	1.070916ms	3.879018004s
2047	650.592µs	10.01µs	29.30523ms	1.6714ms	8.909226716s

figure 3. 50 round averages for Query, functional decryption key generation and decryption times for both types of queries used.

Conclusion

I think I succeeded in reproducing the results depicted in the paper for the most part. The most important observation that was reproduced was that the time strain to achieve differential privacy on this type of multi-input functional encryption scheme is negligible.