Fully Homomorphic Encryption Over The Integers: Implementation and Benchmarking

Thursday 8$^{\text{th}}$ June, 2023 - 12:49

Georgi Tyufekchiev
University of Luxembourg
Email: georgi.tyufekchiev.001@student.uni.lu

This report has been produced under the supervision of:
Marius Lombard-Platet
University of Luxembourg
Email: marius.lombard-platet@uni.lu

***Abstract*—This project focuses on Fully Homomorphic Encryption (FHE) and its implementation using the DGHV protocol. To improve its efficiency we incorporate public key compression and modulo switching in the implementation. Two lattice attacks were employed to evaluate the security of the implemented protocol. Additionally, benchmarking was conducted to assess the performance of both the DGHV protocol and the lattice attacks. The findings from this project shed light on the protocol's computational efficiency.**

1. Introduction

Fully Homomorphic Encryption (FHE) is a powerful cryptographic technique that allows computation on encrypted data without the need to decrypt it. In other words, it enables data to be processed in its encrypted form, preserving the privacy and confidentiality of the data. This revolutionary cryptographic technique has opened up new horizons for secure data processing and has been of great interest to the research community and industry.Historically, encryption was used primarily to protect data in transit or at rest. However, with the advent of cloud computing and the growth of the internet, it became apparent that data was vulnerable to attacks even while it was being processed. For instance, a cloud provider that processes data on behalf of clients must be trusted with the data's confidentiality and integrity. Unfortunately, this trust cannot be guaranteed as no system is fully secure. FHE solves this problem by enabling data to be processed in its encrypted form, which significantly reduces the risk of data breaches. This is achieved by transforming plaintext data into ciphertext, which can be manipulated through mathematical operations, without revealing the underlying data. The result of the computation is then encrypted, and the output is decrypted only when it is needed. Therefore, the decrypted data remains hidden from the computation party, and the data remains secure. The application of FHE is vast, and it has several use cases. For instance, it can be used in the healthcare sector, where privacy regulations dictate that patients' sensitive medical data must be protected. In this case, FHE can be used to process medical data without compromising its confidentiality. It can also be used in the finance sector to perform secure financial transactions, online voting systems to ensure the integrity of the voting process, and machine learning to enable privacy-preserving data analysis. FHE is a complex cryptographic technique that requires advanced mathematics to understand. The first FHE scheme was proposed in 2009 by Craig Gentry, but it was inefficient and impractical. However, over the years, several FHE schemes have been proposed, and they have become more efficient and practical. The most popular FHE schemes in use today are based on the Learning with Errors (LWE) problem and Ring Learning with Errors (RLWE) problem.

In this project, we will focus on implementing the DGHV scheme along with some optimisation techniques. The protocol will be benchmarked in terms of computational time and memory usage. Furthermore, we will look into existing attacks against the scheme to test the security of the protocol.

2. Fully Homomorphic Encryption

The idea of fully homomorphic encryption has existed for many decades. It was only in 2009 when the first scheme was created by Gentry [5]. This initial scheme is said to be "somewhat homomorphic" since each ciphertext contains some amount of noise, which increases with every operation, making it unrecoverable after a certain threshold has been passed. To solve this problem, Gentry came up with the idea of "bootstrapping". The bootstrapping technique in Fully Homomorphic Encryption (FHE) involves a homomorphic evaluation of the decryption procedure. It utilizes an encrypted secret key and a ciphertext to produce an ”equivalent” ciphertext that can be subjected to further computations. This encrypted secret key is often referred to as a bootstrapping or refreshing key. By leveraging this bootstrapping key, FHE enables the generation of ciphertexts that maintain the same underlying plaintext while reducing noise, thereby facilitating continued computation on encrypted data.. However, since the scheme is heavy in computations, it is not practical to use it in most systems. Since then many improvements have been proposed, which depend on the LWE or RLWE problems, with some of them being [2], which does not include bootstrapping, and [3] based on approximate numbers.

This project is mainly focused on fully homomorphic encryption over the integers [4], which includes optimisation techniques like modulo switching and public key compression.

3. Preliminaries

First, we briefly look at the main scientific concepts and technical tool needed for the project.

3.1. Scientific

- Public key compression is a method to reduce the size of the public key so that it requires less memory for storage.
- Modulo switching - the basic idea is to multiply the ciphertext by a term so that the noise is reduced. The advantage is that the bit length of the modulus is linear in regards to the depth instead of exponential.
- LLL [7] is a polynomial-time algorithm, which is used for lattice reductions.

3.2. Technical

C++ is a general-purpose programming language. It was initially created as an extension to the C language, but has since added many features such as OOP. The current version is C++ 20, in which the project is implemented.

3.3. Notation

The parameters of the FHE are denoted by Greek letters (e.g. η, γ, ρ) with λ being the security parameter. All logarithms are in base 2, unless stated otherwise. For $z \in \mathbb{R}$ we denote $\lceil z \rceil, \lfloor z \rfloor, \lceil z \rfloor$ as rounding z up, down, or to the nearest integer. The notation $[z]_p$ denotes the number z modulo some other number p.

4. Pre-requisites

The main pre-requisites for the project is to have basic understanding of the C++ programming language.

5. Scientific Research

5.1. DGHV Scheme

The DGHV scheme [12] is a second generation FHE. It is simpler than Gentry's original idea in the sense that DGHV is over the integers instead of ideal lattices. The scheme defined by the authors is as follows

Parameters
The following 4 parameters are used

- η is the bit-length of the secret key
- γ is the bit-length of the integers in the public key.
- ρ is the bit-length of the noise.
- τ is the number of integers in the public key.

The constraints they put on the parameters are

- $\eta \geq \rho.\Theta(\lambda \log^2 \lambda)$
- $\gamma = \omega(\eta^2 \log \lambda)$
- $\rho = \omega(\log \lambda)$
- $\tau \geq \gamma + \omega(\log \lambda)$

Construction
$KeyGen(\lambda)$. Generate a random prime number of size η bits. For $0 \leq i \leq \tau$ sample $x_i \leftarrow \mathbb{D}_{\gamma,\rho}(p)$, where

$$\mathbb{D}_{\gamma,\rho}(p) = \{q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho) : x = p{*}q{+}r\}$$

Restart unless x_0 is odd and $[x_0]_p$ is even. The public key is $pk = (x_0, x_1, \ldots, x_\tau)$ and the secret key is $sk = p$.

$Encrypt(\mathbf{pk}, m \in \{0,1\}$. Choose a random subset $S \subseteq \{1, 2, \ldots, \tau\}$ and a random integer r in $(-2^\rho, 2^\rho)$ and output the ciphertext

$$c = \left[m + 2r + 2\sum_{i \in S} x_i \right]_{x_0} \quad (1)$$

$Evaluate(\mathbf{pk}, C, c_1, \ldots, c_t)$. Given a circuit C with t input bits and t ciphertext, apply addition and multiplication gates of C to the ciphertexts c_i.

$Decrypt(sk, c)$. Output $m \leftarrow [c]_p \mod 2$

The scheme is somewhat homomorphic since it has limited number of homomorphic operations on ciphertexts

5.2. Public Key Compression

One of the disadvantages of the traditional scheme is the amount of memory it takes to store the public key, which can be in the range of couple gigabytes. To solve this problem, it is possible to compress the public key as described in [4]. The process is as follows

- Use a pseudo random number generator (PRNG) with public seed to generate χ_i for $1 \leq i \leq \tau$, where each χ_i is of size γ bits.

- Compute the small corrections $\delta_i = \chi_i - 2r \mod p$. Only the small δ_i are stored in the public key together with the seed.
- To restore the x_i, the public seed can be used to generate the same χ_i and compute $x_i = \chi_i - \delta_i$

The memory required by the new public key is τ numbers of η bits (e.g.$\eta = 2700$) instead of γ bits (e.g. $\gamma = 10^7$), reducing the size from gigabytes to just a few megabytes (roughly 4MB for the concrete example given by the authors).

5.3. Modulo Switching

The basic idea of modulo switching is to reduce the noise of the ciphertext by switching to a lower modulus after one homomorphic operation. First, why is this needed? Assume that the noise is bounded by some B and one multiplication level roughly increases the noise to B^2, after one more level the noise is B^4 or B^{2^L} for L levels of multiplications. To be able to correctly decrypt the ciphertext we need a modulus p of size bigger than $\log B^{2^L} = \Theta(2^L)$ - exponential scaling.

Now, assume we have public keys $p_L = B^{L+1}$, $p_{L-1} = B^L$,...,$p_0 = B$ and therefore $\frac{p_{i-1}}{p_i} = \frac{1}{B}$. After one level of multiplication the noise is B^2, but then we switch to lower modulo by multiplying with $\frac{p_{L-1}}{p_L}$ reducing the noise back to B. After L levels of multiplication the ciphertext has noise lower than B and can be decrypted. Because of this, the modulus p needs to be of size $\Theta(L)$ - linear scaling instead of exponential. Of course, it is not as simple as multiplying by $\frac{p_{L-1}}{p_L}$ because the ciphertext will be invalid. The process of switching moduli for DGHV can be seen again in [4].

For modulo switching we need two additional protocols. The $SwitchKeyGen$, which essentially outputs 2 vectors. The first one is **y** - a vector of Θ random numbers modulo $2^{\eta'+1}$, where $\eta' > \eta$. This vector will be used to mask the ciphertext. The second vector is σ - a vector encryption of a secret vector s' under sk'. Those two vectors are used during $SwitchKey$ to produce the new ciphertext under a new modulo. One thing to notice is that, when we have addition it is not necessary to switch the modulo, since addition increases the noise much slower than multiplication. The procedure, in simple terms, can be described as follows

- Run $KeyGen(1^\lambda)$ and also provide the number of levels L. Let μ be a parameter specified later. Generate a ladder of L moduli from $\eta_L = (L+1)\mu$ down to $\eta_1 = 2\mu$. For each public key pk_i and corresponding secret key sk_i, run

  $$\tau_{pk_i \to pk_{i-1}} \leftarrow SwitchKeyGen(pk_i, sk_i, pk_{i-1}, sk_{i-1})$$

  The final public key is

  $$pk = (pk_L, \tau_{pk_L \to pk_{L-1}}, ..., \tau_{pk_2 \to pk_1})$$

  and the secret key is $sk = (p_1, ..., p_L)$
- the Encryption works as in the original scheme, with the difference that the ciphertext is encrypted with one of the keys. For decryption, if we assume that the ciphertext is under modulo p_j, do $m \leftarrow [c]_{p_j} \mod 2$.
- For addition and multiplication, if both of the ciphertexts are under different modulo, run $SwitchKey$ to make them under the same modulo. After the operation, again run $SwitchKey$ to switch to a lower modulo, unless both ciphertexts are under pk_1, in that case simply do the operations as in the original scheme.

5.4. DGHV Benchmarking

Before we proceed wit the concrete benchmarkings, all C++ code is compiled using the flags $-O2$ and $-march = native$, meaning the compiler will use 2nd degree optimisation on the code and will further optimise it for the hardware of the current device we use. The first benchmark is performed only on the public compression, without modulo switching. The parameters we used are

TABLE 1: Parameters used to test different security levels of DGHV with public compression

Security Level	ρ	η	$\gamma \times 10^{-6}$	τ	pk size
Toy	27	1026	0.15	158	38 KB
Small	41	1558	0.83	572	214 KB
Medium	56	2128	4.2	2110	1084 KB

All security levels were tested on Intel i5-9300H processors at 3-GHz on Ubuntu. The results from testing are summarised in the table below.

TABLE 2: Timings of the C++ 20 code

Security Level	KeyGen	Encrypt	Decrypt
Toy	53 ms	2 ms	0 ms
Small	470 ms	38 ms	0 ms
Medium	6694 ms	830 ms	1 ms

From table we are able to see that the public key size is in the range of kilobytes. Now, let us look at what it would like if the key was not compressed. We used the Valgrind tool [8] to perform memory analysis of the program during execution. Only the heap has been analysed, thus we do not include any stack or mmap allocations. The profiling is done on the Toy and Medium instance. From the 2 graphs, found in the Appendinx section, we can see peak of 3 MiB for Toy and 1.1 GB for Medium. On the second graph, we also show that the GMPU library [6] is used to allocate memory for the majority of objects. The peaks occur during the generation of the χ_i, and after we remove them, the memory consumption drops significantly. Both graphs show that public compression is indeed a useful technique to use since we reduce the size from 1.1GiB to only 1084 KB.

Next, we benchmark the DGHV scheme with both modulo switching and public key compression. We used only 5 levels for each instance due to using an off-the-shelf computer. In Tables 3 and 4 we summarise the parameters used during the tests and the timings of the C++ implementation. The performance seems to scale exponentially, which is noticeable when looking at the Medium and Large instance.

TABLE 3: Parameters used to test different security levels of DGHV scheme with modulo switching and public key compression.

Security Level	ρ	μ	$\gamma \times 10^-6$	Θ	pk size
Toy	14	56	0.061	195	58 KB
Small	20	65	0.27	735	192 KB
Medium	26	73	1.02	2925	651 KB
Large	34	82	2.2	5700	1405 KB

The 3rd and 4th graph, found in the Appendix section, again show the memory consumption of the program for the Toy and Small instance. In both graphs, we are able to see 5 peaks at the beginning, which represent the 5 public keys being generated and the immediate drop after the keys has been compressed. As the program continues its execution we can see a second peak. This occurs when we add 2 ciphertexts and we switch from one modulo to another, which requires to generate a few vectors of size $(\eta' + 1) \times \Theta$. Graph 5 shows the memory consumption of the Large instance. The key generation peaks are harder to distinguish and the peak for modulo switching again seems to grow exponentially compared to the previous instances.

5.5. Lattice Attacks and Results

The first attack [1] is on the encrypted bits, and the goal is to recover the plaintexts using a lattice attack. The encryption of the bit is computed using the formula $c = (m + N \times X_1) \bmod X_0$, where X_0, X_1 are the public key and N is randomly generated. Their algorithm consists of the following process.

- Create a lattice of size $t \times (t+1)$, which consists of the ciphertexts and the identity matrix.
- Reduce the matrix using LLL, and the resulting basis is used to create a second lattice of size $t \times (2t-3)$
- Reducing the lattice again with LLL, which gives us the plaintexts in the first vector of the new lattice.

We created the algorithm in both SageMath and C++. The benchmarking was only done with C++ since the Python language is much slower due to many factors. The tests were ran on Intel i5-9300H at 3-GHz running on Ubuntu. The following parameters were used, where n used to set ρ and $d = 2n$, which is the bit-length of N.

Level of security	n	η	ρ	d	γ
Toy	32	1024	32	64	32768
Small	64	4096	64	128	262144
Medium	80	6400	80	160	512000

For each level of security, we ran it 10 times for each $t = 5, 10, 15, 20$, where t is the amount of ciphertexts. From each t, we took the best result. The first table and chart show the results from the Toy parameters.

TABLE 5: Toy Results. Time is in milliseconds

t	n	LLL_1	LLL_2	Total
5	32	22ms	<1ms	40ms
10	32	156ms	<1ms	161ms
15	32	454ms	1ms	463ms
20	32	996ms	4ms	1013ms

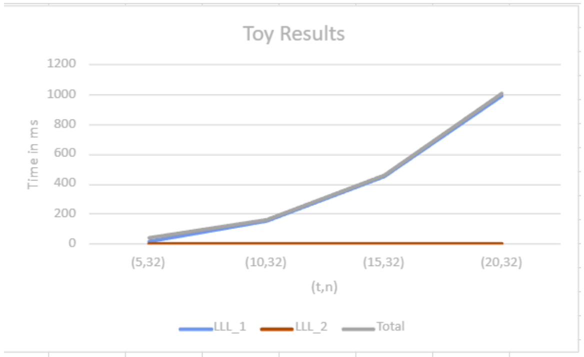

Figure 1: Toy Parameters Graph

The table and chart below show the results for the Small parameters

TABLE 6: Small Results. Time is in seconds (except for LLL_2)

t	n	LLL_1	LLL_2	Total
5	64	1,41s	<1ms	4,21s
10	64	8,14s	1ms	10s
15	64	26,76s	2ms	30,19s
20	64	56,25	8ms	57,67s

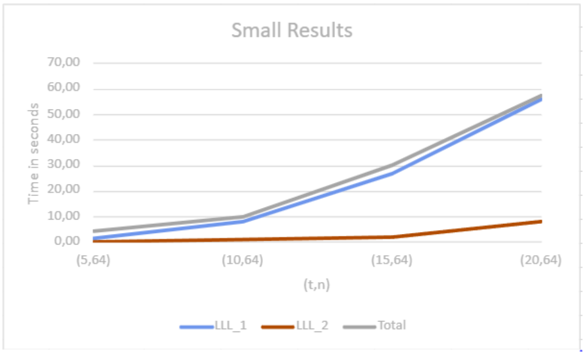

Figure 2: Small Parameters Graph

The table and chart below show the results for the Medium parameters

TABLE 4: Timings for the C++ 20 code

Security Level	KeyGen	SwitchKeyGen	Encrypt	SwitchKey	Decrypt
Toy	16 ms	0 s	2 ms	2 ms	0 ms
Small	58 ms	0 s	11 ms	36 ms	0 ms
Medium	219 ms	0 s	32 ms	481 ms	0 ms
Large	533 ms	15 s	63 ms	2179 ms	0 ms

TABLE 7: Medium Results. Time is in seconds (except for LLL_2)

t	n	LLL_1	LLL_2	Total
5	80	4,3s	<1ms	29,8s
10	80	33,4s	<1ms	40s
15	80	103,3s	3ms	113,3
20	80	218,7s	11ms	227,9s

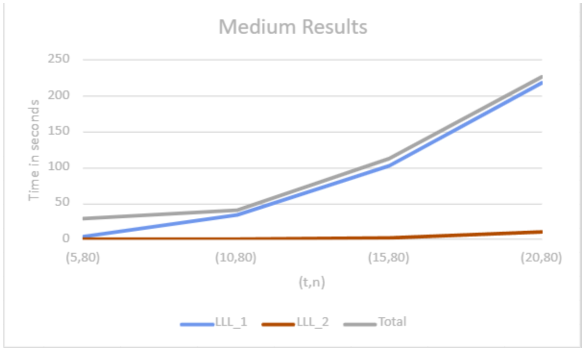

Figure 3: Medium Parameters Graph

From the result, we can conclude that the first LLL reduction is the bottleneck of the algorithm. All of the graphs look to be similar with an approximate complexity of $O(n^3)$ (except for the second LLL reduction, which seems to remain constant). For all LLL reduction we used the C++ library FPLLL [10] and used their default options for the LLL algorithm. Currently they do not support multicore processing for that particular algorithm. Nevertheless the performance is quite good considering we use an off-the-shelf computer.

Moving on to the second lattice attack [9], they use a similar method to the first attack. The ciphertexts are of the form $c = pq_i + r_i$ for $i = 1, \dots, m$ and the goal is to solve the equation $a_1c_1 + \dots + a_mc_m = a_1r_1 + \dots + a_mr_m$, where $r_i \neq 0$ and $\|a_i| < 2^\alpha$ and if $\alpha < \frac{1}{m}\log(c_m) + log(\frac{\sqrt{m}}{m+1}) - \rho$, the p can be found in polynomial time. The algorithm works as follows. First, create the lattice with the basis matrix and apply the LLL algorithm to it.

$$M' = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & c_1 \\ 0 & 1 & 0 & \dots & 0 & c_2 \\ 0 & 0 & 1 & \dots & 0 & c_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & c_{m-1} \\ 0 & 0 & 0 & \dots & 0 & c_m \end{bmatrix} \tag{2}$$

The resulting basis is used to create a second lattice with basis matrix

$$M' = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & Ca_1 \\ 0 & 1 & 0 & \dots & 0 & Ca_2 \\ 0 & 0 & 1 & \dots & 0 & Ca_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & Ca_m \\ 0 & 0 & 0 & \dots & 0 & C\left(c_1a_1 + \dots + c_ma_m\right) \end{bmatrix} \tag{3}$$

After reducing the second lattice, the solution is found in the vector, which has a 0 as the last entry. The C parameter must be chosen carefully in order to find that particular vector, which has the solution. Unfortunately, finding such C proved to be difficult when we increase the amount of ciphertexts used. The reason is that there were many vectors, which have a 0 as the last entry and thus the solution was hidden somewhere in the matrix. Still, we use the following parameters for our benchmark

TABLE 8: Parameters used for the lattice attack.

η	ρ	(γ_1, n)	(γ_2, n)	(γ_3, n)
50	$\sqrt{\eta}$	(79,2)	(123,3)	(210,5)
100	$\sqrt{\eta}$	(117,2)	(200,3)	(328,5)
200	$\sqrt{\eta}$	(253,2)	(300,3)	(443,5)

The results were not very consistent using the same gammas, so we had to pick those, which gave good success rate. Even so, the results are quite disappointing, especially if we tried to use more ciphertexts. As for the LLL reductions, they finished in under a millisecond for all tests.

TABLE 9: Success rate and total time for lattice attack

η	Time γ_1	Success %	Time γ_2	Success %	Time γ_3,	Success %
50	44 ms	60%	62 ms	55%	57 ms	55%
100	56 ms	60%	61 ms	66%	82 ms	60%
200	65 ms	71%	80 ms	54%	125 ms	60%

5.6. Assessment

The goal of the scientific deliverable was too look at the DGHV scheme and the different methods to optimise it as well as test its security with by using lattice attacks. From the benchamrking, we can conclude that public key compression and modulo switching indeed are efficient methods to use when implementing such scheme. The lattice attacks show that DGHV parameters have to be carefully chosen to make them computationally infeasible.

6. DGHV Implementation

6.1. Requirements

In this section we describe the functional and non-functional requirements the system will have. For the non-functional requirements we will use the international standard **ISO/IEC 25051:2014**

6.1.1. Functional Requirements

6.1.1.1. **FR#1**
The program is able to generate a public and private key. In the case of modulo switching it is able to generate multiple key pairs at once.
6.1.1.2. **FR#2**
The program is able to encrypt and decrypt bits correctly.
6.1.1.3. **FR#3**
The program is able to homomorphically add and multiply bits correctly.

6.1.2. Non-Functional Requirements

6.1.2.1. **NFR#1**
Functional completeness, meaning all the protocols must be implemented for system to function as intended.

6.1.2.2. **NFR#2**
Functional correctness, meaning all functions must perform accurate calculations and output the correct result of the concrete operation.

6.1.2.3. **NFR#3**
Time behaviour, the processing times when the protocol is performing its functions, must be fast.

6.1.2.4. **NFR#4**
Resource utilisation, meaning the memory consumption must remain relatively low.

6.1.2.5. **NFR#5**
Modularity, meaning the system is composed of discrete components, facilitating changes and code maintenance.

6.1.2.6. **NFR#6**
Reusability, meaning a function can be used to build different components of the system.

6.1.2.7. **NFR#7**
Modifiability, meaning components can be changed without introducing defects or degrading the system.

6.2. Development Process and Design

The development of the technical deliverable is split into 3 phases - Public Key Compression, Modulo Switching, Lattice Attacks.

6.2.1. Public Key Compression

First, we implement the DGHV scheme with public key compression. Since during modulo switching, we need to generate multiple key pairs, it made sense to use object oriented programming, and create a class called ”PkGenerator” to generate the keys. To store numbers, we decided to use the vector data structure in C++ as it has relatively fast access time. It is also possible to use a deque, which is able to insert and access elements in $O(1)$, faster than a vector, but also requires more memory. All vectors are with known size, thus we can reserve space in advance, which improves the efficiency of the vector (essentially having the same complexity as a deque) because there are no re-allocations, which are quite expensive. Most of the numbers are large, especially when using the recommended parameter sizes, so we are going to use the GMP library to store the integers and perform operations on them. The class consists of the following functions

- KeyGen, which generates the public/private key and also compresses the public key.
- Encrypt, which takes as input the public key and the plaintext to encrypt. The output is the encrypted plaintext.
- Decrypt, which takes as input the private key and a ciphertext. The output is the original plaintext.
- Addition function, which takes as input two ciphertexts and the public key. The output is the result of the addition, which is still encrypted.
- Multiplication function, which takes as input two ciphertexts and the public key. The output is the result of the multiplication, which is still encrypted.

6.2.2. Modulo Switching

The next step is to implement modulo switching. As we decided to use OOP, we will create a separate class called ”SwitchKey” for the modulo switching. After we generate a ladder of decreasing moduli, we can pass the corresponding parameters to the ”PkGenerator” and store each instance in ”SwitchKey”, creating a composition relationship between the two classes. Both classes require a random number generator RNG, which generates the majority of numbers we need. For this reason we use the Mersenne Twister algorithm. It is important to remember that the Marsenne Twister is not secure when it comes to cryptographic applications. Here we use it only for demonstration purposes, otherwise we must use a cryptographically secure RNG.

6.2.3. Lattice Attacks

Both lattice attacks were first created with SageMath [11], Python based language, due to its ”user friendliness”. After we made sure the attacks work correctly, we proceeded to implement them in C++. For the LLL reduction we decided to use the FPLLL library instead of NTL.

6.3. Production

6.3.1. PkGenerator Class

First, we present the ”PkGenerator” class. The first step is the generate a prime number. Since the prime is quite large and we also need to for computations, we are going to store it in a ”mpz_class” type. It is also possible to store it in ”mpz_t” type, but the difference in speed is negligible in our case and the class type is a nice wrapper for the ”mpz_t” type. The GMP library does not have a specific function to generate a prime, we can first generate a number of certain bit length and then use the GMP function to find the next prime from the generated number. They use a probabilistic algorithm to find the prime, which has a very low chance of producing a composite number.

```
mpz_urandomb(prime.get_mpz_t(), state, eta);
mpz_nextprime(prime.get_mpz_t(),
     prime.get_mpz_t());
```

Listing 1: Prime generation

Next, we need to generate the χ_i and r_i. Again for χ_i we need an MPZ type because the size can be for example 1 million bits or more. The RNG is created with the GMP library by using their default algorithm, currently Mersenne Twister, and seeding with the current UNIX time. It is also a good idea to make the seed of a constant type to make sure it does not change when we need to recover the χ_i.

```
const time_t rand_seed {time(NULL)};
rand(gmp_randinit_default)
rand.seed(rand_seed);
chi[i] = rand.get_z_bits(gamma);
```

Listing 2: Generation of chi_i

Of course, it is not necessary to store the χ_i in a vector as we do not need it any more after computing the δ_i. We do it only to show the memory consumption if the public key is not compressed and then the immediate drop after we delete the χ_i. Moving on to the encryption function, we can see it is quite simple. We need to generate a random r and compute the ciphertext, which is then bounded my x_0. The sum of the public key is in a separate function. In the ”computePKsum” function, we can notice that the RNG is created again then the χ_i is directly generated using the ”rand.get_z_bits(γ)”. The value variable $\epsilon \in \{0, 1\}$ to create some randomness in the sum when we encrypt.

```
void computePKsum(){
        mpz_class bound {2};
        mpz_class x;
        mpz_class epsilon;
        gmp_randinit_default(state);
        rand.seed(rand_seed);
        for(int i = 0; i < tau; i++){
            epsilon = rand.get_z_range(bound);
            x = rand.get_z_bits(gamma) − delta[i];
            pk_sum += x * epsilon;
        }
    }

mpz_class encrypt(mpz_class m){
        mpz_class r;
        mpz_class ciphertext_mod;
        mpz_class ciphertext;
        r = rand.get_z_bits(noise);
        computePKsum();
        ciphertext = 2*pk_sum + 2 * r + m;
        mpz_mod(ciphertext_mod.get_mpz_t(),
             ciphertext.get_mpz_t(),
             x_zero.get_mpz_t());
        return ciphertext;
    }
```

Listing 3: Encryption function

The decryption function is also very simple. Here, we can use ”mpz_t” to store the modulus 2. We cannot use a int type for the modulus since the GMP function for computing operations some modulo, requires a ”mpz_t” type number. Is is important not to forget the initialisation of the ”mpz_t” type with the correct GMP function, otherwise the default value is 0.

```
mpz_class decrypt(mpz_class c){
        mpz_class plaintext;
        mpz_class tmp;
        mpz_t modulus;
        mpz_init_set_ui(modulus,2);
        mpz_mod(tmp.get_mpz_t(), c.get_mpz_t(),
             prime.get_mpz_t());
        mpz_mod(plaintext.get_mpz_t(),
             tmp.get_mpz_t(), modulus);
        return plaintext;

    }
```

Listing 4: Decryption function

6.3.2. SwitchKey Class

To start the implementation of the modulo switching, first we need to generate a ladder of decreasing moduli. This time they fit in an integer type, so we can initialize a vector of unsigned integers.

```
void decreasingModuli(){
        eta_ladder[0] = (levels + 1) * mu;
        eta_ladder[eta_ladder.size() − 1] = 2 * mu;
        for (int i = levels−1; i > 1; i−−) {
            unsigned int eta = (i + 1) * mu;
            eta_ladder[eta_ladder.size() − i] = eta;
        }
    }
```

Listing 5: Generate moduli

Now we are ready to create the different key pairs. This is done by creating pointer objects of type PkGenerator. It is crucial not to forget and delete those pointers at the end of the program execution. This can be done by implementing the class destructor.

```
void generatePK(){
        for(int i = 0; i < eta_ladder.size(); i++){
```

```
        PKgenerator* pk = new
              PKgenerator(eta_ladder[i],tau,rho,gamma);
        pk_list.push_back(pk);
    }
  }
  ~SwitchKey() {
    for (auto pk : pk_list) {
        delete pk;
    }
    ...
  }
```

Listing 6: Generate PK ladder

Now we proceed to the generator of the **y** vector, consisting of random number with κ bits of precision. To create a float number, we generate the integer and floating part seperately and then combine them in one. To store floating number we use the GMP type "mpf_class".

```
int genYvector(mpz_class eta_prime){
        gmp_randstate_t state;
        gmp_randclass rand(gmp_randinit_default);
        rand.seed(time(NULL));
        const unsigned int decimal_min = 1;
        const unsigned int decimal_max = (2 <<
              (eta_prime.get_ui() + 1)) - 1;

        for (int i = 0; i < theta; i++) {
            mpz_class decimal_part =
                rand.get_z_range(decimal_max -
                decimal_min + 1) + decimal_min;
            mpf_class f = rand.get_f(kappa) +
                mpf_class(decimal_part);
            y[i]=f;
        }
        gmp_randclear(state);
        return 0;
    }
```

Listing 7: Generating y vector

Next, we need to generate the **s**, **s'** and σ vectors, with the last one being the encryption of **s'**. Only part of the code will be shown. First, we tried to use a vectors of mpz_class to store all generated numbers. After, we tested with large parameters, we found they filled the RAM memory to the max (i.e. 16 GB). This was due to the fact that we need to allocate vectors of size $(\eta' + 1) \times \Theta$ combined with the fact that mpz_class is 16 bytes alone. Then we realised we can store the number in an unsigned long, which is only 8 bytes, and this quickly solved the problem. Since we moved to integer type, creating a RNG can be done with the standard library, which has the Mersenne Twister.

```
 const unsigned int range = (eta_prime.get_ui() +
      1) * theta;
q.reserve(range);
std::vector<unsigned long int> r(range,0);
std::random_device dev;
std::mt19937 rng(dev());
std::uniform_int_distribution<std::mt19937::result_type>
      dist(1,q0.get_ui());
for(int i=0; i<range;i++){
        q.push_back(dist(rng));
    }

 std::uniform_int_distribution<std::mt19937::result_type>
      distr(1,rho);

for(int i=0;i<range;i++){
        r[i] = distr(rng);
    }
```

Listing 8: Vector generation

If we want to switch the modulus after an operation, the following process will happen. First, we expand the ciphertext with the **y** vector. The resulting elements are decomposed into bit representation. Then we just need to compute the dot prodcut of the bit vector with the δ vector and add the original ciphertext modulo 2.

```
  mpz_class switchKey(mpz_class c){
        std::vector<unsigned long int>
              expand_c(theta,0);
        std::vector<bool> bits;
        mpz_class modulus = mpz_class(1) <<
              (eta_prime.get_ui() + 1);
        mpz_class c_mod;
        mpz_mod(c_mod.get_mpz_t(),c.get_mpz_t(),
        mpz_class(2).get_mpz_t());
        mpz_class mod_res;
        for(int i = 0; i < theta;i++){
            mpz_class product = c *
                  mpz_class(y[i].get_ui());
            mpz_mod(mod_res.get_mpz_t(),product.get_mpz_t(),
            modulus.get_mpz_t());
            expand_c[i] = mod_res.get_ui();
        }
        bits.reserve(eta_prime.get_ui() + 1 * theta);
        for(auto elem : expand_c){
            for (int i = eta_prime.get_ui(); i >= 0;
                  --i) {

                bool bit = (elem >> i) & 1;
                bits.push_back(bit);
            }
        }
        mpz_class c_prim = 0;
        for(int i = 0;i<q.size();i++){
            c_prim += delta[i] * bits[i];
        }
        c_prim *= 2;
        c_prim += c_mod;
        return c_prim;
    }
```

Listing 9: Switching modulo

The encryption and decryption function remain unchanged. Next, we show the lattice attacks.

6.3.3. Lattice Attacks

First, we cover the attack by [9]. We use the FPLLL library to constructs the lattices using the ZZ_mat of mpz_t numbers. Again, the mpz_t type has to be properly initialised using the GMP library function. To create the initial lattice we use the following function

```
fplll::ZZ_mat<mpz_t> create_lattice(const
      std::vector<mpz_class>& ciphertexts) {
    int n = ciphertexts.size();
    fplll::ZZ_mat<mpz_t> L(n, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                mpz_t one;
                mpz_init_set_ui(one, 1);
                L[i][j] = one;
            } else {
                L[i][j] = mpz_t{0};
            }
        }
        mpz_t c;
        mpz_init(c);
        mpz_set(c,ciphertexts[i].get_mpz_t());
        L[i][n-1] = c;
    }
    return L;
}
```

Listing 10: First lattice

The identity matrix is created and the last entry is filled with the ciphertexts. The vector of ciphertexts is passed by reference in order not to make copies, which can be expensive. The second lattice is very similar, with the difference that last entry is taken from the reduction of the previous matrix and multiplied by a constant C. Only the last entries are shown in the code below

```
if (i == n) {
            mpz_class sum{0};
            for (int j = 0; j < n; j++) {
                mpz_addmul(sum.get_mpz_t(),
                     c[j].get_mpz_t(),
                     mpz_class(ai[j].get_si()).get_mpz_t());
            }
            sum *= C;
            mpz_t res;
            mpz_init(res);
            mpz_set(res,sum.get_mpz_t());
            L[i][n] = res;
        } else {
            mpz_class res = C *
                 mpz_class(ai[i].get_si());
            mpz_t x;
            mpz_init(x);
            mpz_set(x,res.get_mpz_t());
            L[i][n] = x;
        }
```

Listing 11: Second lattice

To reduce both lattices, we can use the "fplll::lll_reduction()" with the default parameters. To obtain the prime number from the second reduced lattice, we need to compute the gcd, which is available in GMP

```
for(int j=0;j<rows_p;j++){
            if(L_prime[j][cols_p-1].get_si() == 0){
                mpz_class n = c[0] -
                     (abs(L_prime[j][0].get_si()));
                mpz_class m = c[1] -
                     (abs(L_prime[j][1].get_si()));

                mpz_gcd(gcd.get_mpz_t(),n.get_mpz_t()
                ,m.get_mpz_t());
                return gcd;
            }
        }
```

Listing 12: Recovering prime

The second lattice attack [1] looks similar in terms of code. The initial lattice is created with the first entry being the ciphertext multiplied by a large constant and the rest is the identity matrix (omitted in the code below)

```
ZZ_mat<mpz_t> create_lattice(int t,
      vector<mpz_class>& c) {
    ZZ_mat<mpz_t> M(t, t+1);
    mpz_t one;
    mpz_init_set_ui(one, 1);
    for (int i=0; i<t; i++) {
        mpz_t ci;
        mpz_init(ci);
        mpz_set(ci,c[i].get_mpz_t());
        mpz_mul(ci,ci,lambd.get_mpz_t());
        M(i, 0) = ci;
    }
```

Listing 13: Recovering prime

For the second lattice, when we fill it with values from the first lattice, we must not forget to skip the initial column of the first lattice, since we do not need it.

```
ZZ_mat<mpz_t> create_lattice_prime(const
      ZZ_mat<mpz_t>& A, int t) {
    int rows = t;
    int cols = t - 3 + t;
    ZZ_mat<mpz_t> M(rows, cols);
    mpz_t one;
    mpz_init_set_ui(one, 1);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < t - 3; j++) {
            mpz_class val;
            A[j][i+1].get_mpz(val.get_mpz_t());
            mpz_t tmp;
            mpz_init(tmp);
            mpz_set(tmp,val.get_mpz_t());
            mpz_mul(tmp,tmp,lambd.get_mpz_t());
            M[i][j] = tmp;
        }
    }
```

Listing 14: Recovering prime

The plaintexts are found in the first vector of the reduced lattice, so we can simply check they match with the original plaintexts

```
for (int i = cols; i < L_prime.get_cols(); i++) {
        L_prime[0][i].get_mpz(val.get_mpz_t());
        bool res = val.get_ui() == m[i - cols];
    }
```

Listing 15: Recovering prime

Both programs were simple in terms of implementation, and even simpler in SageMath. If speed is not a concern, then SageMath is the better option.

6.4. Assessment

The goal of technical part was to implement the DGHV scheme with public key compression and modulo switching and test its security with lattice attacks. We were able to accomplish all those tasks using the C++ language, with the attacks even implemented with SageMath. The code for the DGHV scheme can definitely be improved and written in a better way by following more strictly C++ standards. Nevertheless, all the requirements have been completed.

Acknowledgment

I, Georgi Tyufekchiev, would like to thank my tutor Dr. Marius Lombard-Platet for supporting me during the project.

7. Conclusion

In this project, we looked at why fully homomorphic encryption is important. Furthermore, we went into more details about the DGHV scheme, a second generation FHE, and how some aspects of it can be improved using public key compression and modulo switching. After we benchmarked our implementation in C++, we saw that those methods indeed improve the efficiency of the DGHV scheme. To test the security of the protocol, we looked at two different lattice attacks, one of which recovers the prime and second one the original plaintexts. From them, we can conclude that parameters for DGHV must be carefully chosen to make the attacks computationally infeasible. Overall, we were able to complete all goals of the project.

8. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:

  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Jingguo Bi, Jiayang Liu, and Xiaoyun Wang. "Cryptanalysis of a Homomorphic Encryption Scheme Over Integers". In: *Information Security and Cryptology - 12th International Conference, Inscrypt 2016, Beijing, China, November 4-6, 2016, Revised Selected Papers*. Ed. by Kefei Chen, Dongdai Lin, and Moti Yung. Vol. 10143. Lecture Notes in Computer Science. Springer, 2016, pp. 243–252. DOI: 10.1007/978-3-319-54705-3_15. URL: https://doi.org/10.1007/978-3-319-54705-3%5C_15.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. Ed. by Shafi Goldwasser. ACM, 2012, pp. 309–325. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262.

[3] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *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*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Springer, 2017, pp. 409–437. DOI: 10.1007/978-3-319-70694-8_15. URL: https://doi.org/10.1007/978-3-319-70694-8%5C_15.

[4] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. “Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers”. In: *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 446–464. DOI: 10.1007/978-3-642-29011-4_27. URL: https://doi.org/10.1007/978-3-642-29011-4%5C_27.

[5] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*. Ed. by Michael Mitzenmacher. ACM, 2009, pp. 169–178. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[6] GNU Project. *GMPLib*. Version 6.2.1. 1991. URL: https://gmplib.org/.

[7] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. “Factoring polynomials with rational coefficients”. In: *Mathematische annalen* 261.ARTICLE (1982), pp. 515–534.

[8] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746. URL: https://doi.org/10.1145/1250734.1250746.

[9] Abderrahmane Nitaj and Tajjeeddine Rachidi. “Lattice Attacks on the DGHV Homomorphic Encryption Scheme”. In: *IACR Cryptol. ePrint Arch.* (2015), p. 1145. URL: http://eprint.iacr.org/2015/1145.

[10] The FPLLL development team. “fplll, a lattice reduction library, Version: 5.4.4”. Available at https://github.com/fplll/fplll. 2023. URL: https://github.com/fplll/fplll.

[11] The Sage Developers. *SageMath, the Sage Mathematics Software System*. Version 9.5. DOI 10.5281/zenodo.6259615. 2022. URL: https://www.sagemath.org.

[12] Marten Van Dijk et al. “Fully homomorphic encryption over the integers”. In: *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer. 2010, pp. 24–43.

9. Appendix

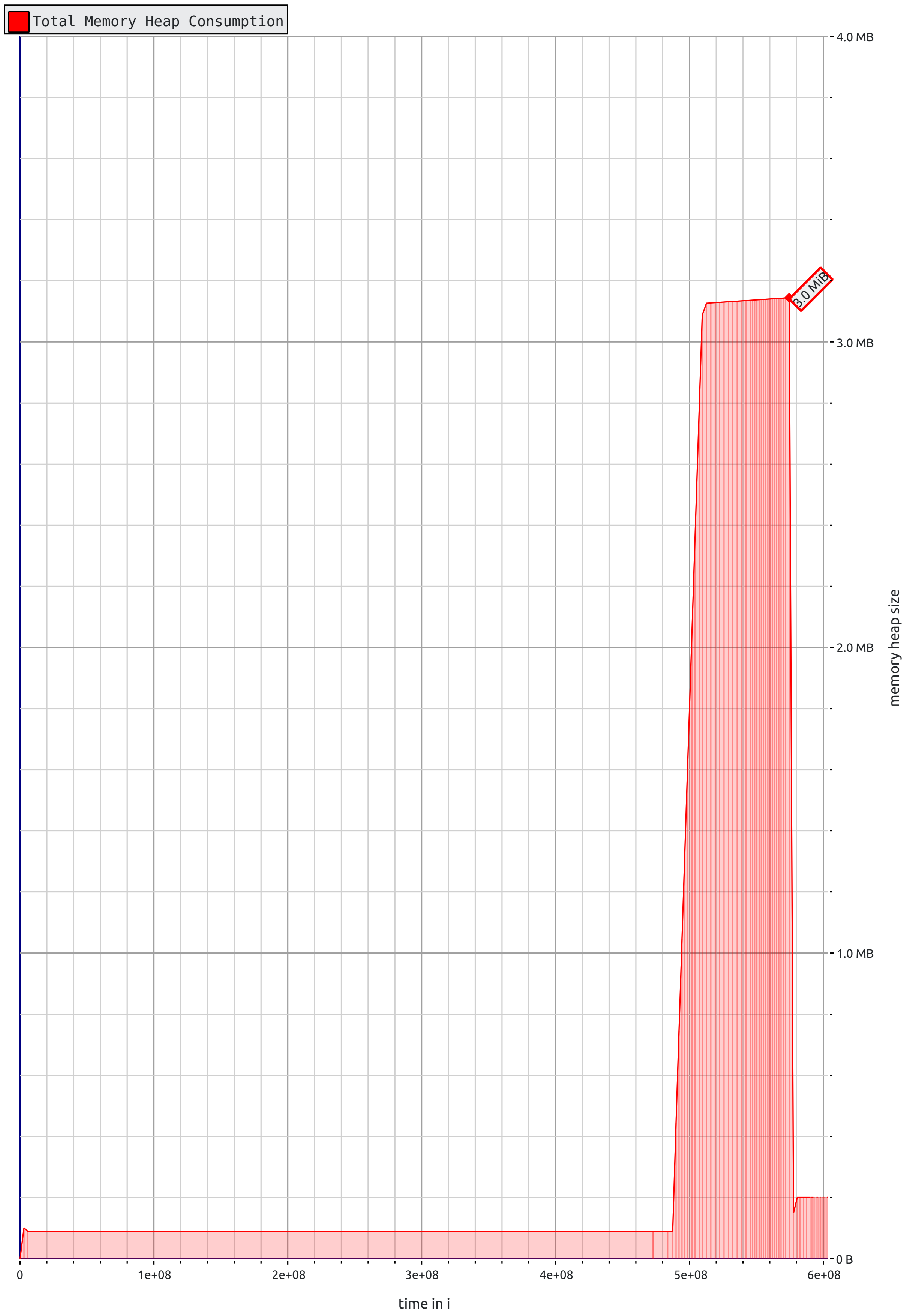

Figure 4: Public Key Compression - Toy parameters

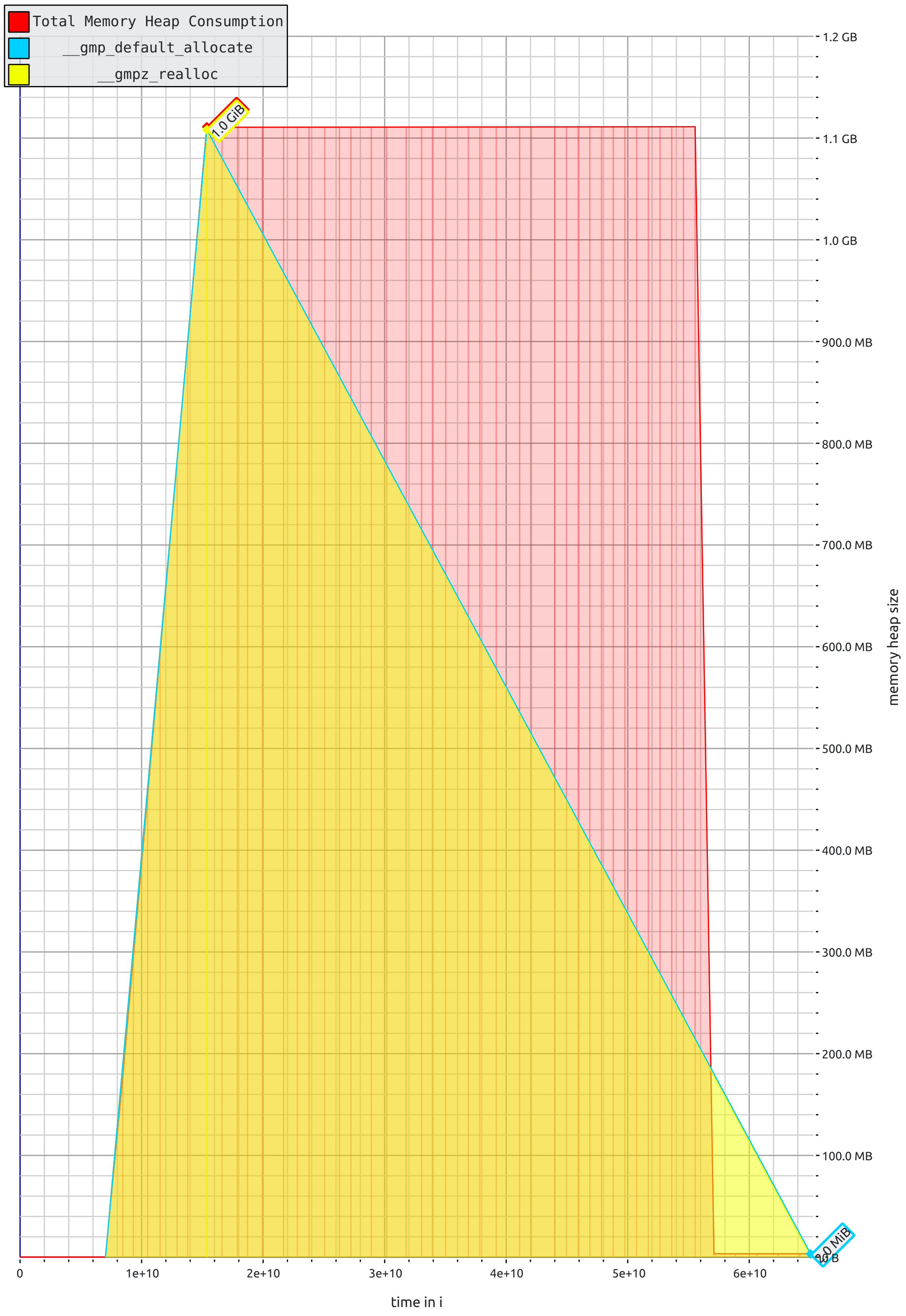

Figure 5: Public Key Compression - Medium parameters

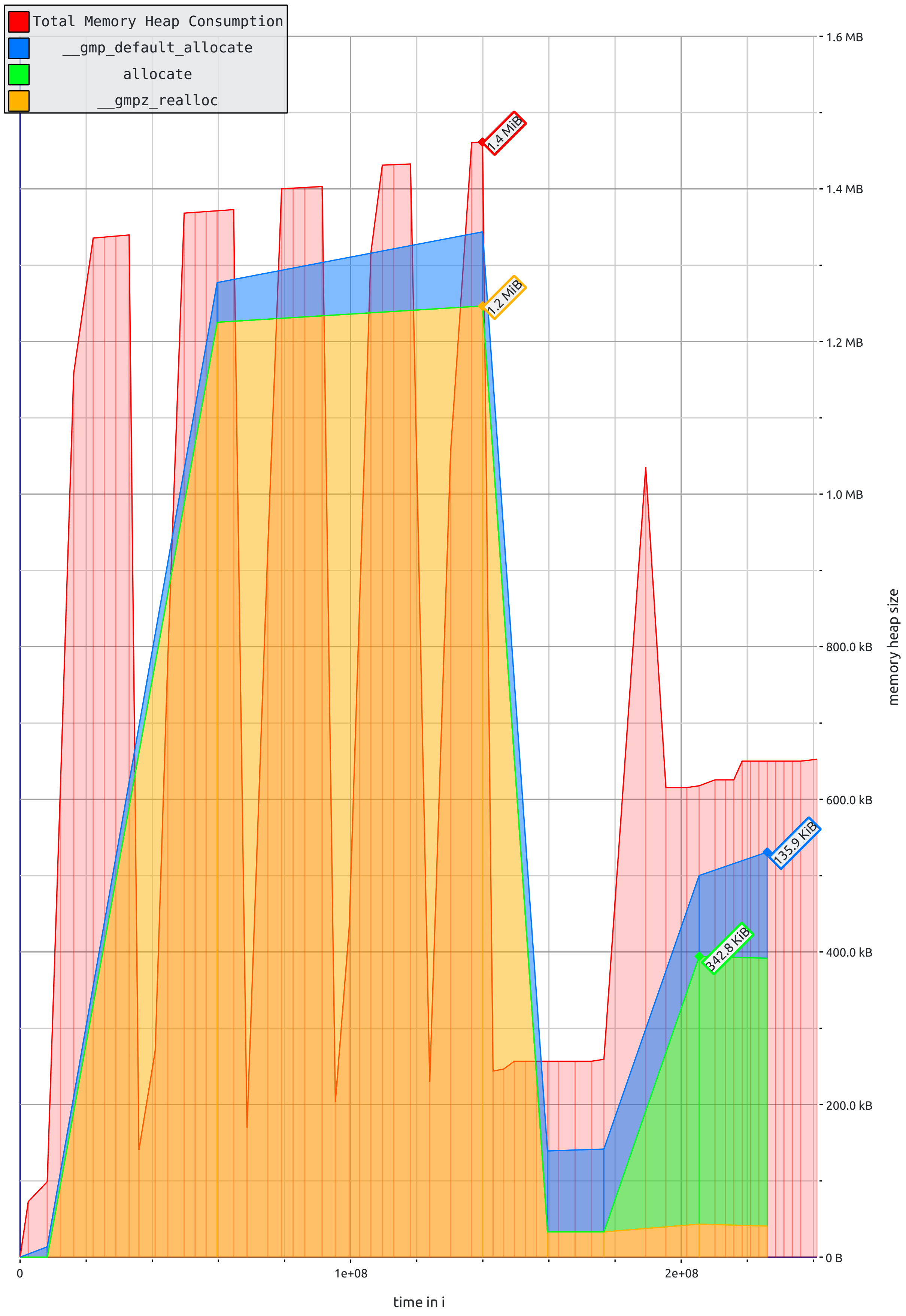

Figure 6: Key Compression, Modulo Switching - Toy parameters

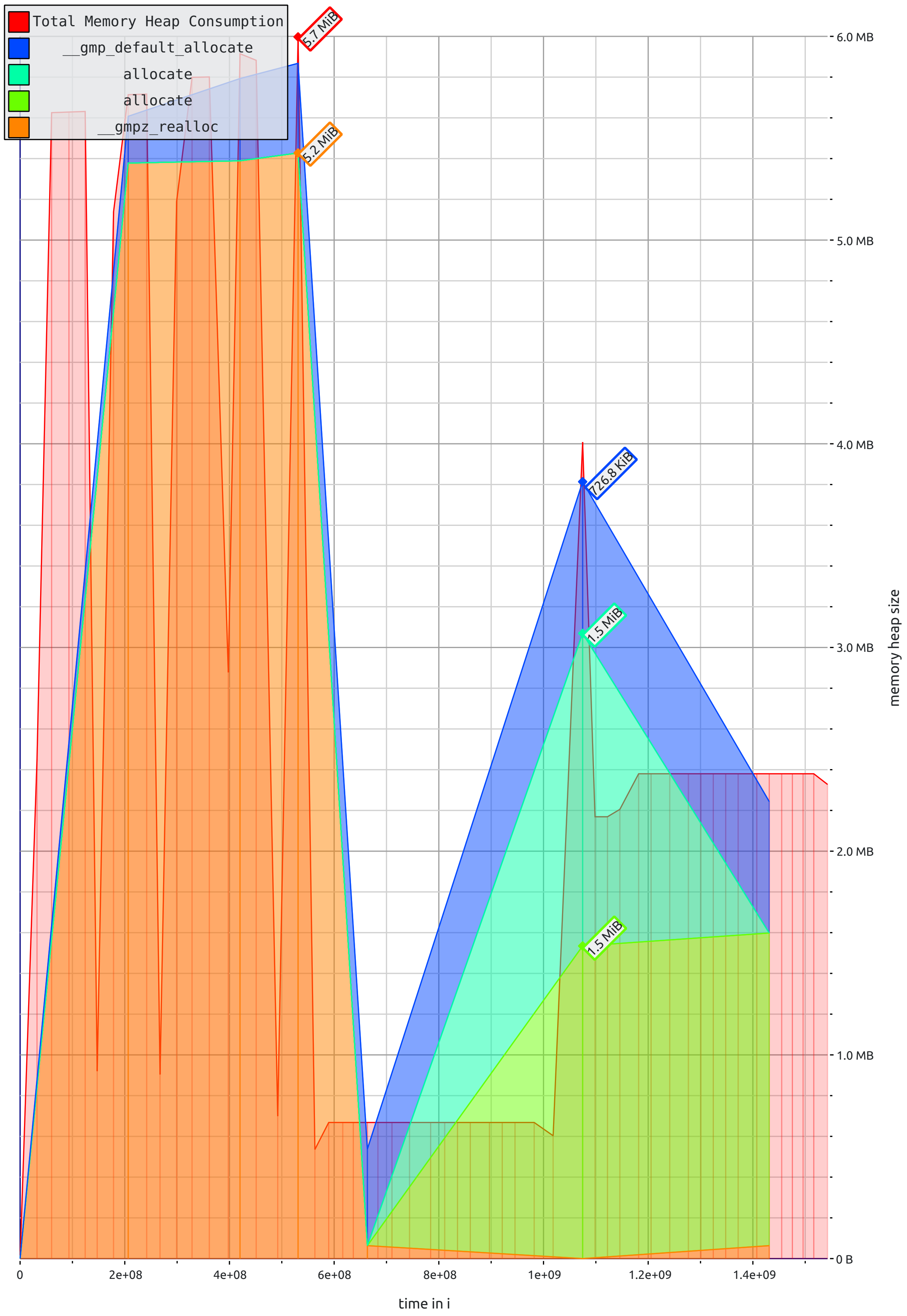

Figure 7: Key Compression, Modulo Switching - Medium parameters

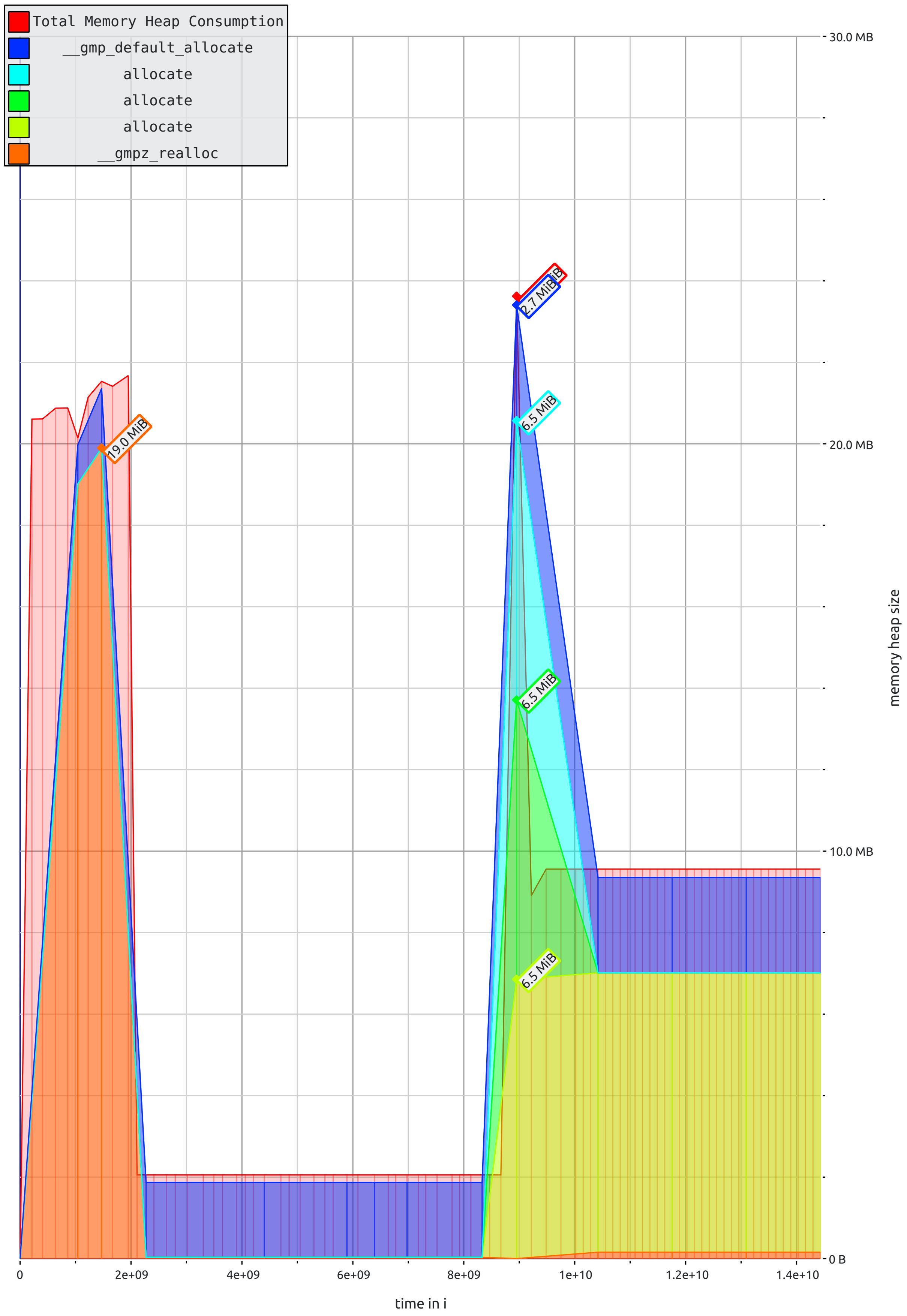

Figure 8: Key Compression, Modulo Switching - Large parameters