

Building a Fully Homomorphic Encryption Scheme in Python

Nolan Hedglin ^{*1}, Kade Phillips ⁺¹, and Andrew Reilley ^{‡1}

¹Department of Electrical Engineering and Computer Science, MIT

May 16, 2019

Executive Summary

The goal of this final project for MIT's 6.857 Computer and Network Security class was to implement a quantum-resistant homomorphic encryption scheme that can eventually be used to encrypt data for blind quantum computation. Using Python, we were able to develop a Gentry-Sahai-Waters homomorphic scheme that supported integer addition. We also review the role that classical homomorphic encryption will play in securing data sent to a quantum computer.

1 Background and Motivation

1.1 Quantum Computation

Quantum computers have been promised to provide computational speedups in factoring, linear algebra, particle simulations, and many other fields of computation [9]. Algorithms for these problems have been designed and are waiting for hardware capable of executing them on reasonably large inputs. As of 2019, most of the largest quantum computers built by companies like Intel, IBM, and Google have under 100 qubits and are thus not useful for solving many problems that are difficult to solve on classical computers. A bigger problem with these early quantum computers is that they have strict isolation requirements that must be met for them to function properly. Their temperatures and power supplies must be strictly controlled for their computations to be completed successfully, and current quantum computers have significant noise even given these conditions. Based on the last few years of progress, it seems that new developments in quantum computation in the next few years will increase the number of qubits that machines can utilize, but that the requirements of maintaining a quantum computer will not decrease significantly. This means that quantum computers will be able to solve more problems and thus be more appealing to users, but that they will not be significantly more accessible to the vast majority of users. For this reason, companies like IBM are building internet connected quantum computers that users can outsource their computations to without needing to maintain the machine themselves. This promises to improve access to quantum computation significantly, but it raises issues with the security of the user's data. An encryption scheme that allows the user to send their

*nhedglin@mit.edu

+kade@mit.edu

‡areilley@mit.edu

data to a third party without disclosing anything while still allowing the third party to perform computations on it is necessary for this outsourcing to be done securely.

1.2 Securing distributed computing using partially homomorphic encryption

One of the issues with outsourcing computation is that the third party must have access to the data in a form that allows them to perform computations on it. For some applications, it is acceptable to send unencrypted data to the third party, but for many cases this is not a passable solution as the data cannot be disclosed. In these cases, a homomorphic encryption scheme must be employed so the data can be encrypted before leaving the original user's computer and not decrypted until the results are returned. Homomorphic encryption refers to the use of schemes that are intentionally malleable that allow operations on the cipher text to have some predictable effect on the message that it encodes. Though malleability is considered to be an undesirable property when not specifically required, it is key to the implementation of a homomorphic encryption scheme. Some encryption schemes not explicitly designed for homomorphic applications exhibit malleability and can thus be used for this purpose. Typically these schemes are only homomorphic under a specific operation that results from their construction, so they are not useful for general blind computation. These schemes are referred to as partially homomorphic or somewhat homomorphic, depending on if they support one or two operations on cipher texts. The limited number of operations that these schemes support limit their applications, and many of them (such as RSA) are not quantum resistant and thus cannot solve the problem of secure outsourced quantum computation.

2 Learning with errors

In order to solve the issue of quantum resistance and to enable a scheme to use a less limited set of operations, a new cryptographic primitive must be utilized. Learning with errors is a computational problem that many homomorphic schemes have been built around, and it is also considered to be a problem that a quantum computer cannot solve in exponentially faster time than a classical one. For these reasons, it is appropriate for our needs. The problem is to determine a linear function given many samples of its value at many inputs, when some of these samples may have small errors. In the context of cryptography, the problem is often given in the form of a secret vector s which is used to generate a set of random points along its length. A random error is added to each point and the adversary must recover s given these points. This problem reduces to the worst case lattice assumption. For this project, we elected to implement an encryption scheme based on this problem called GSW.

2.1 Gentry-Sahai-Waters Encryption (2013)

In 2013, GSW encryption was proposed as a very promising method for performing homomorphic encryption in the classical setting because of its simplicity [7]. GSW applies the difficulty of learning with errors to create a fully homomorphic encryption scheme. There are three commonly referred to generations of fully homomorphic encryption, and this scheme comes from the third. The innovation that started the first generation was the introduction of bootstrapping by Craig Gentry, which uses an encrypted version of the secret key to allow adversaries to safely

decrypt ciphertexts and repack them to reduce combat the growth of error that occurs during homomorphic operations. This bootstrapping procedure allows an unlimited number of operations on ciphertexts, which represented a significant improvement over the previous bounded homomorphic encryption schemes that existed before. The second generation solved the problem of ciphertext multiplication (implemented as an outer product) leading to the size of the ciphertext growing exponentially by applying key switching. Key switching uses a few terms of the secret key s encrypted with a different key to allow the third party to relinearize the size of the ciphertext. Finally, the third generation of FHE, including GSW, does away with this key switching concept by relying on the difficulty of a slightly different problem, the approximate eigenvector problem, removing the need for multiple secret keys. This is the version of FHE that we implemented, and more details about it follow.

2.2 Challenges working with fully homomorphic encryption

The question we initially sought to answer was this: how can we develop a protocol that protects user data from un-trusted quantum computers while preserving their utility? Unfortunately, this question was too optimistic and it implied a lot about the resources we had at our disposal. Current publicly available quantum computers have a high qubit error rate do not scale beyond the tens of qubits. Couple with the fact that homomorphic encryption is computationally intensive, this makes fitting and testing a homomorphic implementation onto an existing quantum machine nearly impossible. This became clear to us as we worked though our FHE implementation, but we decided that it was still an interesting direction for research and that we could still do valuable work by providing a clean, easy to extend implementation of GSW.

3 The algorithm

The GSW scheme we implement supports integer operations and closely mimics that outlined in Michele Minelli's Ph.D dissertation [12].

3.1 $\text{KeyGen}(1^k) \rightarrow (\text{sk}, \text{pk})$

Choose a security parameter k and randomly select a Sophie-Germain prime q represented as k bits. Then let $l = \lceil \log q \rceil$ and $m = nl$.

Randomly sample a secret $\mathbf{s} \leftarrow \mathbb{Z}_q^{n-1}$ and a matrix $\mathbf{A} \leftarrow \mathbb{Z}_q^{(n-1) \times m}$.

Sample an error vector $\mathbf{e} \leftarrow \chi^m$, where χ is an integer-normal distribution modulo q .

Construct the following $n \times m$ generator matrix:

$$\mathbf{G}^{n \times m} = \begin{bmatrix} 1 & 2 & \dots & 2^{\ell-1} & 0 & 0 & \dots & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 2 & \dots & 2^{\ell-1} & \dots & 0 & \dots & 0 & 0 \\ \vdots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & \dots & 1 & 2 & \dots & 2^{\ell-1} \end{bmatrix} \quad (1)$$

Let the secret key be $\text{sk} = \mathbf{t} := (\mathbf{s} \| 1)$ and the public key be $\text{pk} = \begin{pmatrix} -\mathbf{A} \\ \mathbf{s}^\top \mathbf{A} + \mathbf{e}^\top \end{pmatrix}$.

3.2 $\text{Enc}(\text{pk}, \mu \in \mathcal{M}) \rightarrow \mathbf{C}$

Under this notation, μ is the integer we want to encrypt and it is selected from the message space \mathcal{M} . Sample a random matrix $\mathbf{R} \leftarrow \{0, 1\}^{m \times m}$ and compute the ciphertext as follows:

$$\mathbf{C} = \begin{pmatrix} -\mathbf{A} \\ \mathbf{s}^\top \mathbf{A} + \mathbf{e}^\top \end{pmatrix} \mathbf{R} + \mu \mathbf{G} \in \mathbb{Z}_q^{n \times m} \quad (2)$$

3.3 $\text{Dec}(\text{sk}, \mathbf{C}) \rightarrow \mu$

In order to decrypt the message, we multiply the secret key by the ciphertext:

$$\text{sk}^\top \mathbf{C} = \mathbf{t}^\top \left(\begin{pmatrix} -\mathbf{A} \\ \mathbf{s}^\top \mathbf{A} + \mathbf{e}^\top \end{pmatrix} \mathbf{R} + \mu \mathbf{G} \right) = \mathbf{e}^\top \mathbf{R} + \mu \mathbf{t}^\top \mathbf{G} \quad (3)$$

The value we are interested in decrypting is μ . Since $\mathbf{e}^\top \mathbf{R}$ is small, we focus our attention on $\mu \mathbf{t}^\top \mathbf{G}$. To determine the value of μ that most closely fits the vector $\mu \mathbf{t}^\top \mathbf{G}$, we do the following:

Compute the ratio $\mathbf{r} = \frac{\text{sk}^\top \mathbf{C}}{\mathbf{t}^\top \mathbf{G}}$, which will be an array. Ideally every index in this new array should contain the value μ , but in practice this is not the case. To isolate the correct value μ , we take each unique value in \mathbf{r} and multiply it by $\mathbf{t}^\top \mathbf{G}$. We then compute the distance $d = \|\mathbf{r} \mathbf{t}^\top \mathbf{G} - \text{sk}^\top \mathbf{C}\|$ for every unique value in \mathbf{r} . The correct μ will have the smallest distance d , so this is the value we decrypt to.

3.4 Security

This scheme is semantically secure. From the Leftover Hash Lemma and decisional LWE assumption, we know that the product $(\text{pk} \mathbf{R})$ must be truly random for a uniform pk and randomly selected \mathbf{R} . Thus, the ciphertext for a given input μ hides μ information theoretically.

3.5 Addition and Multiplication

Addition is defined as $\mathbf{C}^+ := \mathbf{C}_1 + \mathbf{C}_2$. Expanded, this becomes:

$$\mathbf{t}^\top \mathbf{C}^+ = \mathbf{t}^\top (\mathbf{C}_1 + \mathbf{C}_2) = (\mathbf{e}_1^\top + \mathbf{e}_2^\top) + (\mu_1 + \mu_2) \mathbf{t}^\top \mathbf{G} \quad (4)$$

Since the error terms are negligible, this will give us the output $(\mu_1 + \mu_2) \mathbf{t}^\top \mathbf{G}$.

We can then define multiplication as $\mathbf{C}^\times = \mathbf{C}_1 \cdot \mathbf{G}^{-1}(\mathbf{C}_2)$. This will be equal to:

$$\mathbf{C}_1 \cdot \mathbf{G}^{-1}(\mathbf{C}_2) = \begin{pmatrix} -\mathbf{A}_1 \mathbf{G}^{-1}(\mathbf{C}_2) \\ \mathbf{s}^\top \mathbf{A}_1 \mathbf{G}^{-1}(\mathbf{C}_2) + \mathbf{e}_1^\top \mathbf{G}^{-1}(\mathbf{C}_2) \end{pmatrix} + \mu_1 \begin{pmatrix} -\mathbf{A}_2 \\ \mathbf{s}^\top \mathbf{A}_2 + \mathbf{e}_2^\top \end{pmatrix} + \mu_1 \mu_2 \mathbf{G} \quad (5)$$

The first two terms will be negligible, leaving us with $\mu_1\mu_2\mathbf{G}$.

4 Key generation

Keys are created by first picking the modulus q , which is k bits long (k is the security parameter). The modulus q is a Sophie Germain prime generated using a guess-and-check method with the Fermat primality test. From there, key generation proceeds as described in Section 3. When $k \leq 28$, our software uses native machine words as the datatype for all vector and matrix operations, which dramatically speeds up computation. When k is larger than 28, we start to see overflow while performing multiplications, and so our software automatically switches to python's arbitrary precision integers.

Code listings for the key generation function are attached at the end of the paper.

5 Encryption and decryption

Encryption and decryption are done exactly as described in Section 3, and the code listing for these function are attached at the end of the paper. Note the method for finding μ , which was missing from the sources we studied and required a novel solution.

6 Correctness

For $k = 16$ and above, where the magnitude of elements in the error vector \mathbf{e} is sufficiently small relative to q , encryption and decryption succeeded 100%, according to 1000+ iteration testbenches, and the additive homomorphic property was also verified (although with a smaller number of iterations).

7 Initial results

Our implementation of GSW is capable of key generation, encryption, and decryption and has been tested with security parameters of up to 48. Larger security parameters may work, but performance drops drastically as this value is increased and memory requirements grow significantly. A significant drop in performance is observed for security parameters of 29 and above, because at this level some intermediate values involved in the calculation overflow the largest integer type available in our linear algebra package, forcing it to use very slow arbitrary precision integers. Though performance suffers significantly, the results of our implementation are still correct above this threshold so we consider it acceptable. At a security parameter of 16, it is able to perform 100 encrypt/decrypt cycles in approximately 0.155 seconds. This value grows to around 1.55 seconds for a security parameter of 24. For a security parameter of 29, the largest parameter that can be used before slow arbitrary precision calculations must be utilized, the test takes 5.1 seconds. The performance difference between fixed- and arbitrarily-precision integers is highlighted by the 100 seconds that the test takes for a security parameter of 30. By a security parameter of 48 the

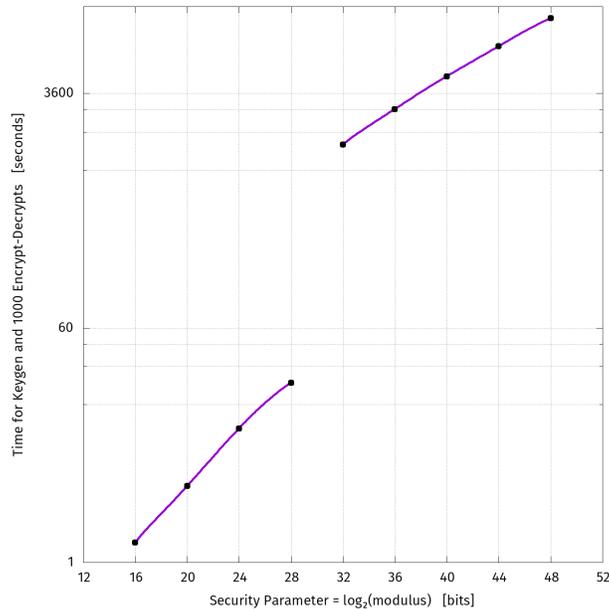


Figure 1: Computation time versus security parameter k . With a larger security parameter, there is an exponential increase in time required to perform the key generation, encryption, and decryption.

test takes 24 minutes to complete, which highlights the poor asymptotic performance of the algorithm. Based on these results, it appears that our solution will not scale to a security parameter that provides any protection from dedicated adversaries. This is acceptable because our goal was to develop a functional, easily extensible version of GSW for experimentation. If the goal was to use it for cryptography for a production system it would be worth looking into other ways to represent the large numbers involved when sizable security parameters are chosen.

8 Toward a fully homomorphic quantum scheme

The search for protocols that can secure cloud quantum computing can be broadly categorized into two tracks: 1) blind computing, which leverages the “no-cloning” property of quantum information, and 2) quantum fully homomorphic encryption (QFHE), which seeks to reconfigure existing FHE schemes for a quantum computer.

8.1 Blind quantum computing

When this problem was first discussed in 2009, early literature on this subject revolved around Alice having a quantum device of her own that she could use to prepare qubits to be sent to the server [2]. Alice could leverage the inherent physical properties of quantum information to secure her information. The property that is of interest to security is the “no-cloning” theorem, which states that quantum information cannot be duplicated. If Eve were to intercept Alice’s quantum message intended for Bob, then both Alice and Bob would know immediately that the

line has been compromised. In addition to security against man-in-the-middle attacks, the no-cloning theorem has also led to several other quantum cryptographic applications, such as key distribution, digital signatures, and secret sharing [1, 6, 8, 10].

The limitation of blind quantum computing is that quantum information must reliably transmitted across between Alice and Bob. Without delving too much into the specifics of quantum network design, this is an issue because a quantum network perform significantly worse in its throughput and range than its classical counterpart. Furthermore, these proposals are limited in that Alice still has quantum capabilities. In effect, blind quantum computing is designed for two purposes: 1) an already quantum-capable user wants to access a more powerful device, and 2) two quantum computers can communicate and perform operations without trusting one another.

8.2 Quantum fully homomorphic encryption

Instead of relying on the no-cloning theorem for securing data against an untrusted quantum computer, we can build protocols around the fortunate coincidence that existing FHE schemes are already quantum resistant. In 2015, Broadbent and Jeffery proved that any existing FHE scheme can be reconfigured to support operations over a quantum computer using the notion of key encapsulation [3]. The data of interest would be encrypted using a quantum one time pad and then the pad itself would be encrypted using FHE. Thus, any operation performed on the data would be preserved on the one time pad. The drawback to their scheme is that, even though it supports the Clifford gate set (i.e. Hadamard, $\pi/8$, controlled NOT, and any combination of those three), it only a limited set of non-Clifford gates (such as Toffoli) and it is done at the expense of the ciphertext dimensions expanding polynomially. Dulek et al. were able to improve upon the work of Broadbent and Jeffery by introducing a one-time use quantum gadget matrix to be used for each non-Clifford gate [5]. By doing so, they transferred the polynomial expansion from the ciphertext to the secret key. One important characteristic of both of these proposals is that the key generation process is quantum.

In 2017, Urmila Mahadev proposed the first homomorphic protocol to use fully classical key generation, which allows the duplication of keys and an improvement in the depth of operation possible [11]. The protocol still uses key encapsulation, but now requires that the randomness of the ciphertext can be recovered after applying the secret key. Mahadev accomplishes this by changing the secret key from an array to a trapdoor that maps to the public key's corresponding lattice. Another interesting property of the protocol is that the correctness of the homomorphic function (i.e. how closely it matches that same operation in plaintext) is only approximate. Mahadev proves that this approximate distance to the correct output is proportional to the per gate error rate of the quantum computer. Therefore, with a negligible gate error rate, the distance between the homomorphic operation and its plaintext counterpart will be negligible. The protocol also requires an additional security assumption that is not present in the other protocols: *circular security*. Circular security is the assumption that the secret key used for decryption will remain secret even after undergoing other operations such as encryption. Fortunately, circular security is already needed in order to multikey GSW, so there is precedence for this assumption holding [4].

9 Future Work

The first item that should be addressed in the future is adding a function to multiply two ciphertexts together. From there we can be a universal gate set to perform any algorithm we want. If we want to reconfigure the scheme we built for interfacing with quantum computers, then we will also need to address ways to reduce the magnitude of the computations being done while maintaining the same level of security. This will allow us to solve more complex problems and explore methods of working with noisy intermediate-scale quantum machines.

References

- [1] BENNETT, C. H. Quantum cryptography using any two nonorthogonal states. *Physical Review Letters* 68, 21 (1992), 3121–3124.
- [2] BROADBENT, A., FITZSIMONS, J., AND KASHEFI, E. Universal Blind Quantum Computation. Tech. rep., 2009.
- [3] BROADBENT, A., AND JEFFERY, S. Quantum homomorphic encryption for circuits of low t-gate complexity. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2015), vol. 9216, pp. 609–629.
- [4] CLEAR, M., AND MCGOLDRICK, C. Multi-identity and multi-key leveled FHE from learning with errors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2015), vol. 9216, pp. 630–656.
- [5] DULEK, Y., SCHAFFNER, C., AND SPEELMAN, F. Quantum homomorphic encryption for polynomial-sized circuits. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2016), vol. 9816, pp. 3–32.
- [6] EKERT, A. K. Quantum Cryptography Based on Bell’s Theorem. *Physical Review* (1991), 661–663.
- [7] GENTRY, C., SAHAI, A., AND WATERS, B. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013), vol. 8042 LNCS, pp. 75–92.
- [8] GOTTESMAN, D., AND CHUANG, I. L. Quantum Digital Signatures. Tech. rep., 2001.
- [9] GRUMBLING, E., HOROWITZ, M., SCIENCE, C., BOARD, T., COMMUNITY, I., BOARD, S., AND SCIENCES, P. Quantum Computing: Progress and Prospects. Tech. rep., The National Academies of Science, Engineering, Medicine, Washington DC, 2018.
- [10] HILLERY, M., BUŽEK, V., AND BERTHIAUME, A. Quantum secret sharing. *Physical Review A - Atomic, Molecular, and Optical Physics* 59, 3 (1999), 1829–1834.
- [11] MAHADEV, U. Classical homomorphic encryption for quantum circuits. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS* (2018), vol. 2018-Octob, pp. 332–338.
- [12] MINELLI, M. Fully Homomorphic Encryption for Machine Learning. Tech. rep.

```

from time import time
from math import floor
from random import randint
import numpy as np

start = None
def stat(msg):
    global start
    now = time()
    if start is None:
        start = now
    print("\x1B[2m%10.4f %s\x1B[0m" % (now-start, msg))

def powmod(a, b, m):
    """ Returns the power a**b % m """
    # a^(2b) = (a^b)^2
    # a^(2b+1) = a * (a^b)^2
    if b==0:
        return 1
    return ((a if b%2==1 else 1) * powmod(a, b//2, m)**2) % m

def is_prime(p):
    """ Returns whether p is probably prime """
    for null in range(16):
        a = randint(1,p-1)
        if powmod(a,p-1,p) != 1:
            return False
    return True

def gen_prime(b):
    """ Returns a prime p with b bits """
    p = randint(2**(b-1), 2**b)
    while not is_prime(p):
        p = randint(2**(b-1), 2**b)
    return p

def generateSophieGermainPrime(k):
    """ Return a Sophie Germain prime p with k bits """
    p = gen_prime(k-1)
    sp = 2*p + 1
    while not is_prime(sp):
        p = gen_prime(k-1)
        sp = 2*p + 1
    return p

def generateSafePrime(k):
    """ Return a safe prime p with k bits """
    p = gen_prime(k-1)
    sp = 2*p + 1
    while not is_prime(sp):
        p = gen_prime(k-1)
        sp = 2*p + 1
    return sp

def text2array(txt):
    ary = []
    for row in txt.split('\n'):
        if row.strip() != '':
            row = row.replace('[', '').replace(']', '').strip()
            ary.append([int(x) for x in row.split()])
    return np.array(ary, dtype=np.int64)

```

```

from util import *
from math import ceil, log2
import numpy as np
import random

class GSWKeys:
    def __init__(self, k, q, t, e, A, B, datatype):
        self.n = k
        self.q = q
        self.l = ceil(log2(q))
        self.m = self.n * self.l
        self.SK = t
        self.e = e
        self.A = A
        self.PK = B
        self.datatype = datatype

def keygen(k):
    if k > 29:
        datatype = 'object'
    else:
        datatype = np.int64
    # pick a random Sophie Germain prime [q] in the range 2...2**k
    # and get its bit length [l]
    stat("Generating modulus")
    q = generateSophieGermainPrime(k)
    l = ceil(log2(q))
    print(" "*12 + "q = %d" % q)
    #
    # the gadget matrix [G] is an n*m matrix (n rows, m = n*l columns)
    #
    # the secret vector [s] is an (n-1)-dimensional vector,
    # the secret key [t] is -s||1, an n-dimensional vector
    #
    # the error vector [e] is an m-dimensional vector
    #
    # the matrix [A] is an (n-1)*m matrix (n-1 rows, m = n*l columns)
    #
    # the public key [B] is ( A )
    # ( sA+e )
    #
    stat("Generating secret key")
    n = k
    m = n*l
    s = np.random.randint(q, size=n-1, dtype=np.int64).astype(datatype)
    t = np.append(s, 1)
    stat("Generating error vector")
    e = np.rint(np.random.normal(scale=1.0, size=m)).astype(np.int).astype(datatype)
    stat("Generating random matrix")
    A = np.random.randint(q, size=(n-1, m), dtype=np.int64).astype(datatype)
    stat("Generating public key")
    B = np.vstack((-A, np.dot(s, A) + e)) % q

    check = np.dot(t, B) % q
    okay = np.all(check == (e % q))
    if okay:
        stat("Keygen check passed") # t·B = e
    else:
        stat("\x1B[31;1mKeygen check failed\x1B[0m") # t·B ≠ e

    return GSWKeys(k, q, t, e, A, B, datatype)

```

enc.py

```
from util import *
import numpy as np
from scipy.linalg import block_diag

def buildGadget(l, n):
    # the secret vector [s] is an (n-1)-dimensional vector,
    # the secret key [t] is -s||1, an n-dimensional vector
    #
    # the error vector [e] is an m-dimensional vector
    #
    # the matrix [A] is an (n-1)*m matrix (n-1 rows, m = n*l columns)
    #
    # the public key [B] is ( A ) which is an n*m matrix
    # ( sA+e )
    #
    g = 2*np.arange(l)
    return block_diag(*[g for null in range(n)])

def encrypt(keys, message):
    stat("Encrypting message")
    #
    # the gadget matrix [G] is an n*m matrix (n rows, m = n*l columns)
    #
    # the matrix R is an m*m matrix (n*l rows, n*l columns)
    #
    # the ciphertext is (n*m)*(m*m) => an n*m matrix
    #
    R = np.random.randint(2, size=(keys.m, keys.m), dtype=np.int64).astype(keys.datatype)
    G = buildGadget(keys.l, keys.n)
    return (np.dot(keys.PK, R) + message*G) % keys.q
```

dec.py

```
from util import *
from enc import buildGadget
import numpy as np
from scipy.stats import mode

def decrypt(keys, ciphertext):
    stat("Decrypting message")
    msg = np.dot(keys.SK, ciphertext) % keys.q
    g = buildGadget(keys.l, keys.n)
    sg = np.dot(keys.SK, g) % keys.q
    div = np rint((msg / sg).astype(np.float)).astype(np.int64)
    modes = np.unique(div, return_counts=True)
    modes = sorted(zip(modes[0], modes[1]), key = lambda t: -t[1])
    best_num = 0
    best_dist = float('inf')
    for mu, count in modes:
        dist = (msg - mu*sg) % keys.q
        dist = np.minimum(dist, keys.q - dist)
        # dist = np.linalg.norm(dist)
        dist = np.dot(dist, dist)
        if dist < best_dist:
            best_num = mu
            best_dist = dist
    return best_num
```

```
import numpy as np
from util import *
from keygen import keygen
from enc import encrypt
from dec import decrypt

keys = keygen(24)

for a,b in [(1,1), (17,19), (34,62)]:
    ca = encrypt(keys, a)
    cb = encrypt(keys, b)
    a_b = a + b
    ca_cb = (ca + cb) % keys.q
    d_ca_cb = decrypt(keys, ca_cb)

    print(" "*12 + "Expected %d" % a_b)
    print(" "*12 + "Received %d" % d_ca_cb)
    if a_b == d_ca_cb:
        print(" "*12 + "\x1B[32;1mPassed\x1B[0m")
    else:
        print(" "*12 + "\x1B[31;1mFailed\x1B[0m")

ca = encrypt(keys, a)
cb = encrypt(keys, b)
a_b = a + a + a + b + b + b
ca_cb = (ca + ca + ca + cb + cb + cb) % keys.q
d_ca_cb = decrypt(keys, ca_cb)

print(" "*12 + "Expected %d" % a_b)
print(" "*12 + "Received %d" % d_ca_cb)
if a_b == d_ca_cb:
    print(" "*12 + "\x1B[32;1mPassed\x1B[0m")
else:
    print(" "*12 + "\x1B[31;1mFailed\x1B[0m")
```

```
import numpy as np
np.set_printoptions(edgeitems=6, linewidth=200)

from util import text2array
from keygen import keygen
from enc import encrypt
from dec import decrypt

keys = keygen(20)

a = int(input('\n A = '))
b = int(input(' B = '))
print()

ca = encrypt(keys, a)
cb = encrypt(keys, b)

print('\nCiphertext of A \x1B[38;5;203m→ /home/kade/cA\x1B[0m\n\x1B[38;5;33m')
print(ca)
print('\x1B[0m\nCiphertext of B \x1B[38;5;203m→ /home/kade/cB\x1B[0m\n\x1B[38;5;33m')
print(cb)

np.set_printoptions(threshold=np.inf, linewidth=np.inf)
fhA = open('/home/kade/cA', 'w')
fhB = open('/home/kade/cB', 'w')
print(ca, file=fhA)
print(cb, file=fhB)
fhA.close()
fhB.close()

while True:
    fn = input('\x1B[0m\nCiphertext of f(A,B) \x1B[38;5;203m← ')
    fh = open(fn, 'r')
    ciphertext = fh.read()
    fh.close()

    cf = text2array(ciphertext) % keys.q

    print('\x1B[0m')
    f = decrypt(keys, cf)

    print('\nDecrypted message = \x1B[32;1m%d\x1B[0m' % f)
```

```
import numpy as np
np.set_printoptions(threshold=np.inf, linewidth=np.inf)

from util import text2array

fh = open('/home/kade/cA', 'r')
ca = text2array(fh.read())
fh.close()

print('Loaded [ca] from /home/kade/cA')

fh = open('/home/kade/cB', 'r')
cb = text2array(fh.read())
fh.close()

print('Loaded [cb] from /home/kade/cB')

def write2file(c):
    fh = open('/home/kade/fAB', 'w')
    print(c, file=fh)
    fh.close()
    print('Wrote matrix to /home/kade/fAB')

# This file should be run interactively with [python -i demoServer.py]
# after ciphertexts are generated with [demoClient.py]
```