Project 2: Cryptanalysis of integer based homomorphic cryptosystems

Computer Algebra for Cryptography (B-KUL-H0E74A)

Deadline: 4pm (Leuven time) on 26/05/2023

Homomorphic encryption is one of the hottest topics in cryptography since it allows to manipulate encrypted data without decrypting it first. The prototypical use case for this technology is where a user encrypts his data and stores the encrypted data in the cloud, whilst still allowing the server to perform computations on this encrypted data. To illustrate this with a simple example: assume I encrypt the marks for each student, and store these on a server, I would still be able to ask the server to compute **an encryption** of the average mark, and send it back to me. The server has no knowledge on the individual marks, it only knows it has to compute the average of all the encrypted marks.

The goal of this project is to design, implement and test attacks on integer based homomorphic cryptosystems (one existing system and one totally new). The first such system [2] was proposed by van Dijk, Gentry, Halevi and Vaikuntanathan in 2009 and will be referred to as the DGHV cryptosystem.

1 Submission

The result of this project consists of two parts: one .m file containing all the Magma code you wrote during the project and one .pdf file containing your report. Note that the report should be **max 5 pages (not counting references)**. Reports written in LaTeX are preferred, but this is not mandatory. Please follow the numbering of the tasks in this document, e.g. Task 1.a and so on.

Make sure that **both** files clearly contain your **name and student number**.

The two files should be emailed to `computer.algebra@esat.kuleuven.be` (as attachment) and the title of your email should read "Submission project 2". Your email should be received before **4pm (Leuven time) on 26/05/2023**.

2 The DGHV cryptosystem

In this section we will present a symmetric key version of the DGHV cryptosystem. Note that the paper [2] presents a slightly more complex public key system, where the public key essentially corresponds to many ciphertexts encrypting the zero message.

The system relies on the following parameters (η, ρ, γ) which are all bit-lengths of the following variables:

- η: the secret key is an **odd integer** p, of bit-length η, i.e.
$$2^{\eta-1} < p < 2^{\eta}$$
- ρ: the noise terms r will be integers in the interval $\mathbb{Z} \cap]-2^{\rho}, 2^{\rho}[$. Note that ρ will be quite a bit smaller than η, and thus, the noise terms r will be much smaller than p.
- γ: the multipliers q will be integers in the interval $\mathbb{Z} \cap [0, 2^{\gamma}/p[$

The symmetric key homomorphic DGHV cryptosystem to encrypt single bit messages $m \in \{0, 1\}$ can now be formally defined as follows:

- **Key Generation**
  - The secret key is an **odd integer** p, of bit-length η, i.e.
  $$2^{\eta-1} < p < 2^{\eta}$$
  The secret key is kept private by the user.
  - The evaluation key is given by $x_0 = pq_0$ where q_0 is a uniformly random multiplier in $\mathbb{Z} \cap [0, 2^{\gamma}/p[$. The evaluation key can be made public, e.g. it can be given to the server.
- **Encryption** To encrypt a message $m \in \{0, 1\}$, sample a uniformly random noise term $r \in \mathbb{Z} \cap]-2^{\rho}, 2^{\rho}[$ and a uniformly random multiplier $q \in \mathbb{Z} \cap [0, 2^{\gamma}/p[$, and return the ciphertext c
$$c = pq + 2r + m\,.$$
An encryption of m will be denoted as $\mathcal{E}(m)$. An encryption as c above is called "fresh" since no operations have been executed on the ciphertext. In particular, the noise term r will be smaller than 2^{ρ} in absolute value.
- **Decryption** To decrypt a ciphertext c, compute the unique integer $c' \in]-p/2, p/2[$ such that $c' = c \bmod p$, and return the message $m = c' \bmod 2$. The decryption of c will be denoted as $\mathcal{D}(c)$.

- **Addition** Given two ciphertexts $c_1 = \mathcal{E}(m_1)$ and $c_2 = \mathcal{E}(m_2)$, one can compute an encryption of the sum of the two messages by defining

$$c_3 = c_1 + c_2 \bmod x_0$$

- **Multiplication** Given two ciphertexts $c_1 = \mathcal{E}(m_1)$ and $c_2 = \mathcal{E}(m_2)$, one can compute an encryption of the product of the two messages by defining

$$c_3 = c_1 \cdot c_2 \bmod x_0$$

2.1 Basics of the DGHV cryptosystem

Task 1 [3 marks]

The following questions should be addressed in your report.

1.a Generalize the DGHV cryptosystem such that it can encrypt messages in $\mathbb{Z}_N$ instead of just in $\mathbb{Z}_2$, in particular, specify any changes to Key Generation, Encryption and Decryption (Addition and Multiplication will remain the same). Note that the encryption of a message m in $[0, \ldots, N[$ should also consist of a single integer (just like in the original scheme), so splitting m up into its individual bits and encrypting these, is not a valid answer. [If you fail to generalize the system, simply keep on working with $N = 2$]. Also argue why you impose some restrictions on the private key p.

1.b For the system you designed in [1.a], derive a condition on η, ρ, γ, N such that decryption works.

1.c For the system you designed in [1.a] analyse the noise growth after one addition and also after one multiplication. Derive a bound for the maximum number of "fresh" ciphertexts you can multiply together before decryption fails, i.e. what is the maximum k such that

$$\prod_{i=1}^{k} c_i \bmod x_0$$

still decrypts, where each c_i is a fresh ciphertext.

Task 2 [2 marks]

Implement the DGHV cryptosystem as you designed it in [1.a] with the following APIs:

2.a `DGHVKeyGen(eta::RngIntElt, gamma::RngIntElt, N::RngIntElt) -> p::RngIntElt, x0::RngIntElt`, which on input the parameters η, γ, N

generates secret key p and evaluation key x_0. The types of these input / output variables are all integers.

2.b `DGHVEncrypt(m::RngIntElt, N::RngIntElt, p::RngIntElt, rho::RngIntElt, gamma::RngIntElt) -> c::RngIntElt`, which encrypts a message $m \in [0, \dots, N-1]$ (you can throw an error if m is not in that interval) given a secret key p, and the parameters ρ and γ specifying the sizes of the noise terms and multipliers.

2.c `DGHVDecrypt(c::RngIntElt, N::RngIntElt, p::RngIntElt) -> m::RngIntElt` which decrypts a ciphertext c and returns the corresponding message $m \in [0, \dots, N-1]$.

2.d `DGHVAdd(c1::RngIntElt, c2::RngIntElt, x0::RngIntElt) -> c::RngIntElt` which computes a ciphertext c encrypting the sum of m_1 and m_2, the messages encrypted in c_1 and c_2.

2.e `DGHVMult(c1::RngIntElt, c2::RngIntElt, x0::RngIntElt) -> c::RngIntElt` which computes a ciphertext c encrypting the product of m_1 and m_2, the messages encrypted in c_1 and c_2.

2.f `DGHVNoiseTerm(c::RngIntElt, N::RngIntElt, p::RngIntElt) -> r::RngIntElt` which returns the noise term r contained in a ciphertext c. Note that for $N = 2$ the noise term r is defined as

$$c = pq + 2r + m$$

in particular, in this case ($N = 2$) do not return $2r$, but only r. Use this function to plot the growth of the noise when computing products of k ciphertexts as in [2.c] for growing k, and verify that your bound in [2.c] corresponds with your implementation. Choose appropriate η, ρ to illustrate this behaviour.

2.2 Lattice attack

A server that receives many ciphertexts c_i from a particular user (say his/her private key is called p), therefore has many integers satisfying

$$c_i = pq_i + v_i \, ,$$

where $v_i = 2r_i + m_i$ (in the case $N = 2$) are quite small compared to p (since the r_i are much smaller than p). This shows that the c_i are all near-mulitples (almost multiples but not quite exact) of the common secret key p. This problem is called the Approximate Common Divisor problem, since p is the GCD of integers close to the c_i. The paper [1] gives an extensive overview of existing attacks on the ACD problem.

Study Sections 3 and 4 of the paper [1] (you can ignore Section 5 which is a different type of attack) and choose either the algorithm from Section 3 or the algorithm from Section 4 to implement.

Some Magma functions that might be useful are the following:

- `Lattice(B::Mtrx) -> Lat`: construct a lattice from a basis matrix, with basis vectors as rows
- `LLL(L::Lat) -> Lat, AlgMatElt`: perform LLL reduction on lattice L
- `BKZ(L::Lat, k::RngIntElt) -> Lat, AlgMatElt`: perform BKZ with block size k

Task 3 [5 marks]

3.a Explain your choice, i.e. why did you choose Section 3 vs. Section 4? Also explain the core of the algorithm you have chosen, i.e. why does the algorithm work? There is no need to copy the analysis of [1] which derives when the algorithm works; you just have to explain why it works.

3.b Implement your attack with the following API: `LatAttackDGHV(eta::RngIntElt, rho::RngIntElt, cs::SeqEnum(RngIntElt)) -> p::RngIntElt` where the input specifies the parameters η and ρ that were used to encrypt, and a sequence of t valid ciphertexts, and returns a candidate for the private key p. How do you verify that you have found the correct p?

3.c If you are not given η and ρ, can you still manage to mount the above attack?

3.d Run your attack for the following parameter sets

η	ρ	γ	t
100	20	2000	10, 20, 30, 50, 100, 500
100	80	1000	10, 20, 30, 50, 100, 500

and determine the minimal number of ciphertexts you require before the attack succeeds. What would you do if you have access to many more ciphertexts than the minimal number? Does the choice of lattice reduction algorithm LLL vs. BKZ influence this number? Explain the influence of the different parameters on the running time of the attack.

3 A new variant of the DGHV cryptosystem

A very recent proposal called System X (you will not find any literature on this, so don't waste time looking for it) for another integer FHE scheme works as follows:

- **Key Generation** The private key consists of 3 primes p_1, p_2, p_3 where p_1 and p_2 have bit-length λ and p_3 has bit-length $\alpha\lambda$ for some integer $\alpha \geq 2$ which will determine how many additions / multiplications can be done before decryption fails. The evaluation key in this case is $x_0 = p_1 p_2 p_3$. The prime p_1 defines the message space, in particular, a message can be any element in $[0, \ldots, p_1[$.

- **Encryption** To encrypt a message $m \in [0, \ldots, p_1[$, sample a uniformly random element $r \in [0, \ldots, p_2[$ and encode the message as

$$e = CRT([m, r], [p_1, p_2]) \in [0, \ldots, N-1] \quad \text{where} \quad N = p_1 p_2 \,.$$

Using a 2-D lattice reduction (see slide 21 of the lecture on lattices), find small x, y (e.g. with $|x|, |y| \leq 2N^{1/2}$) such that

$$e = \frac{x}{y} \bmod N \,.$$

The encryption $\mathcal{E}(m)$ of the message m then finally is given by

$$c = \frac{x}{y} \bmod p_3 \in [0, \ldots, p_3[\,.$$

- **Decryption** To decrypt a ciphertext c, first compute $c' = c \bmod p_3$, express c' as a fraction of small integers x, y, i.e. find x, y such that $c' = \frac{x}{y} \bmod p_3$ and finally return the message $m = \frac{x}{y} \bmod p_1$.

- **Addition** Given two ciphertexts $c_1 = \mathcal{E}(m_1)$ and $c_2 = \mathcal{E}(m_2)$, one can compute an encryption of the sum of the two messages by defining

$$c_3 = c_1 + c_2 \bmod x_0$$

- **Multiplication** Given two ciphertexts $c_1 = \mathcal{E}(m_1)$ and $c_2 = \mathcal{E}(m_2)$, one can compute an encryption of the product of the two messages by defining

$$c_3 = c_1 \cdot c_2 \bmod x_0$$

3.1 Basics of the System X cryptosystem

Task 4 [2 marks]

4.a Explain why decryption works and also derive a necessary condition on λ and α for decryption to work.

4.b Given parameters λ and α, determine the maximum number of **additions** of fresh ciphertexts you can do before decryption fails. Similarly, derive a bound on the maximum number of **multiplications** you can do before decryption fails. Compare the homomorphic capabilities of this variant with the DGHV scheme, i.e. which scheme do you prefer and why?

Task 5 [2 marks]

Implement the System X cryptosystem with the following APIs:

5.a `SystemXKeyGen(lambda::RngIntElt, alpha::RngIntElt) -> p1::RngIntElt, p2::RngIntElt, p3::RngIntElt, x0::RngIntElt`, which on input the parameters λ, α generates the secret key primes p_1, p_2, p_3 of bit-lengths specified above.

5.b `SystemXEncrypt(m::RngIntElt, p1::RngIntElt, p2::RngIntElt, p3::RngIntElt) -> c::RngIntElt`, which encrypts a message $m \in [0, \ldots, p_1 - 1]$ (you can throw an error if m is not in that interval) given a secret key p_1, p_2, p_3.

5.c `SystemXDecrypt(c::RngIntElt, p1::RngIntElt, p3::RngIntElt) -> m::RngIntElt` which decrypts a ciphertext c and returns the corresponding message $m \in [0, \ldots, p_1 - 1]$.

5.d `SystemXAdd(c1::RngIntElt, c2::RngIntElt, x0::RngIntElt) -> c::RngIntElt` which computes a ciphertext c encrypting the sum of m_1 and m_2, the messages encrypted in c_1 and c_2.

5.e `SystemXMult(c1::RngIntElt, c2::RngIntElt, x0::RngIntElt) -> c::RngIntElt` which computes a ciphertext c encrypting the product of m_1 and m_2, the messages encrypted in c_1 and c_2.

Using your implementation above, verify that the bounds you derived in [4.b] are indeed correct.

3.2 Cryptanalysis of the System X cryptosystem

Task 6 [6 marks]

6.a Show that if you would know p_3 and one plaintext / ciphertext pair, i.e. a message m and corresponding ciphertext $c = \mathcal{E}(m)$, then you can easily recover the other parts of the secret key, namely p_1 and p_2. This shows it really is sufficient to find p_3 to break System X completely (having a known message / ciphertext pair is a very common assumption).

6.b Can you rewrite the encryption equation $c = \frac{x}{y} \bmod p_3$ such that you recover a problem that resembles ACD?

6.c Do the lattice attacks from Section 3 or 4 in the paper [1] still apply to the problem you found in [6.b]? What goes wrong?

6.d Design and implement an attack on the System X cryptosystem that recovers p_3 given a list of t ciphertexts c_i. What is the complexity of your attack as a function of λ, α and t? The API of the attack is `AttackSystemX(lambda::RngIntElt, alpha::RngIntElt, cs::SeqEnum(RngIntElt),` `x0::RngIntElt) -> p3::RngIntElt` where the input specifies the parameters λ and α that were used in key generation, and a sequence of t ciphertexts, and the evaluation key and returns a candidate for the private key p_3. [Note that this really is an open ended question, you can use lattices, but don't have to. As an attacker, you are totally free to choose which techniques you use.] Illustrate your attack for $\lambda = 256$ and $\alpha = 2, 3, 4, 5, 6$ by determining the minimal number of ciphertexts required and tabulating the running time.

4 An active attack on DGHV [Bonus marks]

One of the main problems of basic cryptosystems such as the DGHV cryptosystem as described in Section 2, is that (as it is formulated now) there are no restrictions on what is seen as a "valid" ciphertext, i.e. a ciphertext that was really obtained as an encryption of a message.

Such cryptosystems can be attacked by what is called an active attack: in this case, an attacker can ask a user to decrypt anything that looks like a ciphertext and receive what the user thinks is the decryption of this ciphertext. The attacker therefore exploits the user as a "Decryption Oracle". Although it might look strange that you give an attacker the power to ask a user to decrypt anything the attacker wants, this functionality is often available in practice in authentication protocols, i.e. in a protocol where the user has to prove he knows the secret key.

Looking at the decryption equation of the DGHV cryptosytem, it is clear that $\mathcal{D}(c)$ is a deterministic function of the input c, but also the secret key p, so it might leak information about p. An active attack exploits this flaw by **repeatedly and adaptively** submitting "ciphertexts" to the user to decrypt and using the output to derive information on p.

Task 7 [2 marks]

7.a Design an active attack against the basic DGHV cryptosystem described in Section 2 (assuming $N = 2$) and analyze how many calls to the decryption oracle your attack requires as a function of η, ρ, γ.

7.b Implement your attack using the following API `ActAttackDGHV(p::RngIntElt) -> n::RngIntElt` that takes input the private key p, and returns the number of calls to the decryption oracle that the attack required. Note that you might find it strange that you pass the private key p as input to the attack (which is the quantity you are trying to recover), but this is only so that you can mimic the decryption oracle (i.e. call the function `DGHVDecrypt(c, 2, p)`). The **only place** where you should use p is in the decryption oracle, i.e. in the call to the following function:

```
function DecryptionOracle(c,p)
  return DGHVDecrypt(c, 2, p);
end function;
```

As you can see, this is just a wrapper around your own decryption function. Illustrate your attack by running it on the following parameter sets:

η	ρ	γ
100	20	1000, 2000, 3000
200	40	1000, 2000, 3000
1000	100	1000, 2000, 3000

7.c Is it possible to change the decryption function such that the active attack is no longer possible? How would you do this?

References

[1] Steven D. Galbraith, Shishay W. Gebregiyorgis, and Sean Murphy. Algorithms for the approximate common divisor problem. Cryptology ePrint Archive, Report 2016/215, 2016. `https://eprint.iacr.org/2016/215`.

[2] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2009/616, 2009. `https://eprint.iacr.org/2009/616`.