Lecture Notes for Cryptographic Computing
4. Oblivious Transfer (Passive Security)

Lecturers: Claudio Orlandi, Peter Scholl, Aarhus University

October 23, 2023

1 Oblivious Transfer

When talking about Oblivious Transfer (or OT for short) we will refer (unless specified otherwise) to the main variant of OT, namely *1-out-of-2* OT (or $\binom{2}{1}$-OT) which is described by the following functionality:

- Alice inputs a choice bit $b \in \{0,1\}$;
- Bob inputs two messages m_0, m_1;
- Alice learns $z = m_b$;

Informally, a secure OT protocol should guarantee that Alice does not learn anything about the unchosen message and that Bob should not learn anything about the input choice c;

1-out-of-n OT The notation $\binom{2}{1}$-OT suggests that you can get other flavours of OT by plugging different numbers. $\binom{n}{1}$-OT is a natural generalisation, defined as you would expect: Alice inputs a choice value between 0 and $n-1$, Bob inputs n messages, Alice learns only one of them and Bob does not learn which one. $\binom{n}{1}$-OT also directly implies simple two-party secure computation protocols for passive adversaries and small functionalities. The protocol goes as follows: Alice uses her input x as the choice value, and Bob uses his input y to constructs the messages $M_0, \ldots, M_{n-1}$ as $M_i = f(i, y)$.

Removing the Dealer from BeDOZa: Our motivation for studying OT is to be able to design secure two-party computation protocols which do not need a trusted dealer. In particular we want to use OT to implement the same functionality offered by the trusted dealer in the BeDOZa protocol (Boolean circuits, passive security). Remember that in that protocol, for each multiplication gate, the dealer samples random bits u_A, v_A, u_B, v_B, w_B, computes $w_A = (u_A \oplus u_B) \cdot (v_A \oplus v_B) \oplus w_B$ and then sends (u_A, v_A, w_A) to Alice and (u_B, v_B, w_B) to Bob. The dealer can be replaced using the following simple protocol using a $\binom{4}{1}$-OT :

- Alice samples random bits u_A, v_A and inputs $i = 2 \cdot u_A + v_A$ to the OT.
- Bob samples random bits u_B, v_B, w_B and inputs to the OT the following four messages:

$$\begin{aligned}
M_0 &= (0 \oplus u_B) \cdot (0 \oplus v_B) \oplus w_B, \\
M_1 &= (0 \oplus u_B) \cdot (1 \oplus v_B) \oplus w_B, \\
M_2 &= (1 \oplus u_B) \cdot (0 \oplus v_B) \oplus w_B, \\
M_3 &= (1 \oplus u_B) \cdot (1 \oplus v_B) \oplus w_B,
\end{aligned}$$

- Alice sets $w_A = M_i$.

It is easy to see that the protocol produces the right distribution and the security follows from the security of the underlying OT protocol.

Exercise 1. (OT for secret-shared AND)

In Week 2, Exercise 5, we have argued that the only necessary ingredient for implementing the BeDOza protocol (with passive security) is a trusted dealer that samples random bits s_A, r_B, s_B, computes $r_A = r_B \oplus s_A \cdot s_B$, then sends (r_A, s_A) to Alice and (r_B, s_B) to Bob. Show that Alice and Bob can replace such a dealer by a single call to 1-out-of-2 OT.
Hint 1:
Hint 2:

Random OT: There is a variant of OT, called random-OT, which is a randomized functionality where the parties have no input. The functionality samples random bits b, s_0, s_1 and outputs $(b, z = s_b)$ to Alice and (s_0, s_1) to Bob. The following exercise shows that, given random-OT, you can get OT on chosen inputs with a very simple and *information-theoretically secure* transformation. This means that any protocol that uses OT can instead *preprocess* random-OT ahead of time, saving expensive computation later.

Exercise 2. (Random OT and $\binom{2}{1}$-OT)

There is a simple protocol with perfect security against active adversaries for $\binom{2}{1}$-OT in the random-OT hybrid model. Can you construct it?
Hint 1:
Hint 2:

Exercise 3. (Bit OT is symmetric)

When the sender's messages are both *bits*, there is a perfect reduction between the functionality random-OT defined before and the symmetric functionality random-TO that outputs (s'_0, s'_1) to Alice and $(b', z' = s'_{b'})$ to Bob. Can you find it?
Hint :

2 OT Protocol With Passive Security

We describe a generic way of constructing an OT protocol from any public-key encryption scheme (PKE) with some special properties: for now let's say that the PKE has *pseudorandom public-keys*. Then the following is an OT protocol with passive security:

Choose: Alice (with choice bit b) generates a public key pk_b together with a secret key sk_b, and samples a random string pk_{1-b}; Alice sends (pk_0, pk_1) to Bob;

Transfer: Bob (with input messages m_0, m_1) creates two ciphertexts (c_0, c_1) where c_i is an encryption of m_i using the public key pk_i, and sends (c_0, c_1) to Alice;

Retrieve: Alice decrypts c_b with sk_b and learns m_b;

The protocol is clearly correct. About security: Intuitively, Bob does not learn the choice bit b since we assumed that "real" public keys are indistinguishable from random strings. At the same time, Alice does not learn the unchosen message since she does not know the secret key corresponding to the random string[1]. Most of the rest of this note is devoted to making this argument formal, and based on a more general requirement for the public keys.

3 Public Key Encryption With Oblivious Key Generation

3.1 Defining Oblivious Key Generation

In the previous section we stated that we can construct an OT protocol from any PKE scheme where public keys are pseudorandom. In fact, a weaker condition is sufficient: it is enough to have a PKE where there is an *alternative way of generating public-keys* such that:

- Public keys generated in this way look like regular public keys, and
- It is not possible to learn the secret key corresponding to such a "fake" public key.[2]

We introduce here the concept of a PKE with *oblivious key generation.* This is a regular (IND-CPA secure) PKE defined by three algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ enhanced by a fourth algorithm called "oblivious generation" or OGen. Assume that the secret keys in the PKE scheme are random bit strings in $\{0,1\}^n$. Then, we require that Gen takes as input a secret key, and outputs a valid public key;[3]. similarly, OGen takes as input some randomness in $\{0,1\}^n$, and outputs a fake key that looks like a real one. The key requirement is that given a key output by OGen, *together with the randomness used as input*, it should not be possible to find a corresponding secret key that allows decryption.

We formalize the above properties by asking that OGen satisfies the following:

1. Let b be a random bit, $pk_0 \leftarrow \mathsf{Gen}(sk)$ be the regular public key generation algorithm and $pk_1 \leftarrow \mathsf{OGen}(r)$ be the oblivious public key generation algorithm, where sk and r are chosen uniformly at random in $\{0,1\}^n$. Then there is no PPT (efficient) algorithm D such that $D(pk_b) = b$ with probability significantly larger than 1/2.
2. There is a (possibly randomized) *preimage sampling algorithm* OGen^{-1}, such that:
   - If $pk = \mathsf{OGen}(r)$ and $r' = \mathsf{OGen}^{-1}(pk)$, then $\mathsf{OGen}(r') = pk$.
   - For random $r \in \{0,1\}^n$, and $pk = \mathsf{OGen}(r)$, $r' = \mathsf{OGen}^{-1}(pk)$, the string r' is uniformly distributed in $\{0,1\}^n$.
3. Standard IND-CPA security for real public keys. (Look back at the definition from the previous course).

[1]Since we only consider passive corruptions we can trust that Alice will willingly *choose not to learn one of the secret keys*

[2] *Why erasing does not work:* It might be tempting to think that every public-key encryption can be turned into one satisfying the above properties: simply let Alice create a regular public key/secret key pair and then erase the secret key. This would clearly lead to a public key that looks like a "regular" one. There are two reasons why this does not work: 1) it is not easy to securely erase data (so even an honest Alice might not be able to completely erase the secret key) and more importantly 2) It is not possible to verify that Alice has erased the secret key: Think of "Alice" as being a process running on your PC. The secret key will at some point be in memory before it is deleted. How can we make sure that this value does not get copied (perhaps by the operating system, or a malicious spy process) somewhere else before being erased? Passively corrupted parties follow the protocol correctly, but their entire state can still be observed by the adversary.

[3]You might be more used to Gen outputting both the public key and the secret key. Note that it is always possible to redefine any public-key encryption scheme to satisfy this syntax.

It might seem strange to require that OGen is invertible (or preimage sampleable), when we actually want to show that it's not possible to recover the secret key. However, this turns out to be exactly the property that is needed. Intuitively, the idea is that if it were possible to find the secret key for a fake public key, given its randomness r, then you could recover the secret key for a *real* public key in *exactly the same way*, precisely because the OGen^{-1} algorithm can also be run on a real public key to obtain its (fake) randomness r'.

Below, we will argue this a bit more formally, and show that the above three properties imply the following:

4(a). There exists no PPT (efficient) algorithm $\mathcal{A}$ that can output $sk \leftarrow \mathcal{A}(r)$ such that $\mathsf{Gen}(sk) = \mathsf{OGen}(r)$.

First of all, note that OGen^{-1} must be able to "explain" real public keys *as if* they had been generated using OGen. This follows from the fact that if it was *not true* that

$$\mathsf{OGen}(\mathsf{OGen}^{-1}(pk)) = pk$$

for a real public key pk, then an adversary could exploit this to distinguish between real and fake public keys, breaking property (1). So it must be the case that

$$\mathsf{OGen}(\mathsf{OGen}^{-1}(\mathsf{Gen}(sk))) = \mathsf{Gen}(sk)$$

At the same time, if $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is a secure encryption scheme then it must be hard to compute secret keys corresponding to public keys generated with Gen, or in other words $pk \leftarrow \mathsf{Gen}(sk)$ must be a *one-way function.* So we reach a contradiction: assume there is an $\mathcal{A}$ which breaks property $4(a)$, then we can invert $pk \leftarrow \mathsf{Gen}(sk)$ by computing

$$sk \leftarrow \mathcal{A}(\mathsf{OGen}^{-1}(pk))$$

In fact for the OT protocol to be secure we need a stronger property. Informally, this property states that the encryption scheme is still IND-CPA secure even if the encryptions are performed using a public key which is output of OGen and the adversary knows the randomness used by OGen to generate that key. The definition is below, and it is possible to prove that this property (similarly to what we have just done for property 4(a), can be proven using properties (1), (2) and (3):

4(b). For all m: Let b be a random bit, $m_0 = m$, m_1 be a uniform random message and $pk \leftarrow \mathsf{OGen}(r)$. Then there exist no PPT (efficient) algorithm $\mathcal{D}$ such that $\mathcal{D}(r, \mathsf{Enc}(pk, m_b)) = b$ with probability significantly larger than 1/2;

❓ Exercise 4.

Prove that 4(b) follows from properties (1), (2) and (3).
Hint :

3.2 Constructing PKE with Oblivious Key Generation

It is a natural question to ask which functions (not only public key generations algorithms) admit "alternative samplers" which are preimage sampleable but indistinguishable from the original output distribution. It turns out that there are good reasons to believe that not all functions admit such an oblivious sampler, and if you want to know why you can have a look at [IKOS11].

At the same time, there are distributions which clearly admit such an alternative sampling method: Think of a pseudorandom generator PRG which expands an n-bit long seed s into a $2n$ bit long string $y = PRG(s)$. A trivial alternative sampler for PRG is to take as input an n-bit string, and simply output $2n$ random bits. This is clearly preimage sampleable, since you can obtain a random preimage by sampling n bits, and the security of the PRG implies that the output of the PRG is indistinguishable from uniformly random $2n$-bit long strings.

A Concrete Example: ElGamal Cryptosystem. The ElGamal cryptosystem allows for an oblivious key generation procedure. Given the description of a group where the DDH assumption is believed to hold (G, g, q) where g is a generator of G and q is the order of G, ElGamal is described by three algorithms:

$\mathsf{Gen}(sk)$**:** On input a secret key $sk = \alpha \in Z_q$ compute $h = g^\alpha$ and output $pk = (g, h)$;

$\mathsf{Enc}_{pk}(m)$**:** on input a message $m \in G$ sample a random $r \in Z_q$, parse $pk = (g, h)$ and output $C = (g^r, m \cdot h^r)$.

$\mathsf{Dec}_{sk}(C)$**:** parse C as (c_1, c_2) and output $m = c_2 \cdot c_1^{-\alpha}$.

We need to construct an oblivious generation algorithm $\mathsf{OGen}(r)$ which outputs $pk = (g, h)$ which is invertible and indistinguishable from Gen. The way this is done depends on the specific group G.

A classic example of a group where the DDH assumption is believed to hold is the multiplicative subgroup of order q of Z_p^*, where p is prime and $p = 2q + 1$. To generate a random element in this group, you can first use the random bit string r to sample a random number s between 1 and p (note that this step is not entirely straightforward — see Exercise 5). Then, output $h = s^2 \mod p$, to ensure that h is a random element of the subgroup of order q, since this group corresponds to all elements of Z_p^* that are squares.

This process is also preimage sampleable, since it is easy to compute square roots modulo a prime, and given a group element s it is possible to sample a random string r which is conditioned to produce the right s using one of the methods from Exercise 5.

Exercise 5.

The OGen algorithm for ElGamal needs to sample a uniform random number between 1 and p. Suppose that your programming language only provides you with random bits. Clearly p is not a power of 2. How do you sample a random number between 1 and p? Let $2^{n-1} < p < 2^n$. There are essentially two ways:

1. Pick a random number r between 1 and 2^n. If $r < p$ output r otherwise repeat.
2. Pick a random number r between 1 and 2^{2n} and output $(r \bmod p)$.

Can you see any advantage/disadvantage of the two methods, in terms of statistical distance between the output of those procedures and a truly uniform random number between 1 and p? What about the running time?

3.3 The OT Protocol

We conclude by formally specifying the OT protocol:

Choose: Alice (with choice bit b):

1. Samples random sk, r;
2. Generates $pk_b \leftarrow \mathsf{Gen}(sk)$, $pk_{1-b} \leftarrow \mathsf{OGen}(r)$;
3. Sends (pk_0, pk_1) to Bob;

Transfer Phase: Bob (with input messages m_0, m_1):

1. Computes $c_0 = \mathsf{Enc}_{pk_0}(m_0; r_0)$ and $c_1 = \mathsf{Enc}_{pk_1}(m_1; r_1)$ using random r_0, r_1;
2. Sends (c_0, c_1) to Alice;

Retrieve Phase: Alice outputs $m_b \leftarrow \mathsf{Dec}(sk, c_b)$;

Exercise 6.

Prove that the above protocol is secure against passive adversaries.
Hint 1:
Hint 2:
Hint 3:

4 Additional Exercises

Mandatory Assignment.

Generalize the above protocol to a $\binom{8}{1}$-OT, and use it to implement a secure two party computation protocol for the *blood type compatibility* function. Use ElGamal encryption as the underlying PKE scheme. Since the goal of the exercise is to better understand the protocol (not to build a full functioning system), feel free to implement all parties on the same machine and without using network communication. For example, you could implement Alice and Bob as two distinct classes in Java and then let them interact in the following way:

```
m1 = Alice.Choose(x);
m2 = Bob.Transfer(y,m2);
z = Alice.Retrieve(m2);
```

OT from RSA: We showed how to realize OT from any cryptosystem with the additional property that public keys can be obliviously sampled. Unfortunately, this is not the case for RSA: RSA public keys are a product of two large primes, and it is not known how to sample from a distribution close enough to this in an invertible way. However, RSA satisfies a different property, namely that given a public key $pk = (n, e)$ possible to generate "ciphertexts" which look like encryptions of random messages, without knowing what the encrypted message is: an RSA encryption of a random plaintext gives a random element in Z_n, and those can be sampled efficiently (see one of the previous exercises for sampling from Z_p). This property is enough to build an OT protocol, in a way similar to the one done before.

Exercise 7. (OT from RSA)

1) Try to formalize the property "it is possible to generate ciphertexts, for which you do not know the plaintexts, which look like encryptions of random messages" by defining an invertible algorithm OEnc following the blueprint of what we have done for OGen. 2) Construct an OT protocol using the RSA encryption scheme (or any other scheme with the same property).
Hint 1:
Hint 2:

Exercise 8. ($\binom{2}{1}$-OT and $\binom{n}{1}$-OT)

Can you construct a protocol for $\binom{n}{1}$-OT using only $\binom{2}{1}$-OT in a black-box way? This means, you should build a protocol for $\binom{n}{1}$-OT which uses (possibly many) "calls" to a $\binom{2}{1}$-OT protocol. That is, you can only use the input/output behavior of the underlying $\binom{2}{1}$-OT, not the protocol itself. Of course you can, since you know that you can implement any function described by a Boolean circuit using the BeDOZa protocol and $\binom{2}{1}$-OT (instead of a dealer).
However there is a "cheaper way" in terms of how many OTs you need to do this. Can you find it?
Hint :

References

[HL10] Carmit Hazay, Yehuda Lindell
Efficient Secure Two-Party Protocols
`http://lib.myilibrary.com.ez.statsbiblioteket.dk:2048/Open.aspx?id=300348`

[IKOS11] Yuval Ishai, Abhishek Kumarasubramanian, Claudio Orlandi, Amit Sahai
On Invertible Sampling and Adaptive Security
ASIACRYPT 2010
Available `http://cs.au.dk/~orlandi/asiacrypt-draft.pdf`

A ElGamal over Finite Fields, Tips and Pitfalls

You should already know how ElGamal over finite fields works from the Cryptography course (see Ivan's book, sections 9.5.1 and 9.5.2). Here you can find a "quick and dirty" recap of some important properties that can help you during the programming hand-in for this week. You only need to read the "How?" parts to know how to make a correct implementation, but I would encourage you to read the "Why?" part as well to refresh your knowledge from Cryptography.

Warning! If you ever want to implement ElGamal in a real world application, remember that in practice there are almost no reasons to implement ElGamal over a finite field, since using elliptic curves gives better performances and security. In a finite field you need to worry about sub-exponential attacks to the discrete logarithm problem like index calculus, whereas when using the right elliptic curves only generic, exponential time attacks are known. This means that in practice today that if you want security comparable to AES-128, you need public keys of around 4000 bits for finite field ElGamal vs. only 256 bits public keys for elliptic curves ElGamal. That being said, learning about elliptic curves for the assignment in this course is a bit of an overkill, so here you are presented with how to implement securely ElGamal over a finite field.

Generate a safe-prime: How? You need to find a large enough safe prime number p. A safe prime number is one of the form $p = 2q + 1$ where q is prime too. You can do this via rejection sampling e.g., pick random prime p and check that $(p - 1)/2$ is prime too. Or you can pick a random prime q and check that $2q + 1$ is prime too. Repeat until you find a safe prime.

Generate a safe-prime. Why? We want to work in a subgroup of prime order q. We want q to be as large as possible. Picking $p = 2q + 1$ gives you the best possible ratio between q (which impacts the security of the system) and p (which impacts the efficiency of the system since public keys and ciphertexts elements will be numbers of the same size as p). You could also pick any other prime p of the form $p = kq + 1$ for a large enough prime q but that wouldn't be as good a choice.

Generate a DDH-safe group. How? You need to find a generator g of order q. This means you need to find a number in $\mathbb{Z}_p^* = \{1, \dots, p-1\}$ with the property that q is the smallest integer such that $g^q = 1 \bmod p$.

You can do this in several ways:

1. Pick an arbitrary element $g \in \mathbb{Z}_p^*$, with $g \neq 1$. Output g only if g has the right order by checking that $g^q \bmod p = 1$. If not, pick a new g (either a new random one or with a counter). This is "rejection sampling" and works since you will only output an element with the right order.
2. Pick an arbitrary element $x \in \mathbb{Z}_p^*$, with $x \neq 1$ and $x \neq -1$. Compute $g = x^2 \bmod p$. Output g. This works since when you raise to the power of 2 you are "killing off" the 2 component of x. More explanation below.

Generate a DDH-safe group. Why? Let's look at the structure of the group $\mathbb{Z}_p^*$. It has order $p-1 = 2q$ meaning that for all $x \in \mathbb{Z}_p^*$ it holds that $x^{2q} = 1 \bmod p$. Every element x of $\mathbb{Z}_p^*$ also has its own order. The difference between the *order of the group* $\mathbb{Z}_p^*$ and the *order of a group element* x is that the order of the group is the smalles integer o such that *for any elements of the group*, $x^o \bmod p = 1$. The order of a single group element x is the smallest integer such that *for that particular* x, $x^o \bmod p$. We know that the order of a group element must divide the order of the group. So since the order of the group $\mathbb{Z}_p^*$ is $2q$ every group element has order either $1, 2, q$ or $2q = p-1$:

- Elements of order 1: there is only one, namely 1 e.g., $1^1 = 1 \bmod p$.
- Elements of order 2: since we work modulo a prime we are in a field and therefore 1 has a single square root e.g., -1 (which is the shortest way of writing $p-1$, and is the same element modulo p) is the only number such that $(-1)^2 = 1 \bmod p$.
- Elements of order q: you have learned how to sample an element of order q above.
- Elements of order $2q$: everything else. Those are not safe to use if you want the DDH assumption to hold. The main thing to note if that for all elements x of order $2q$ it holds that $x^q = -1 \bmod p$. This is because $1 = x^{2q} = (x^q)^2 \bmod p$ so x^q is a square root of 1 (so it's either 1 or -1) but it's not 1 (otherwise the order of x would be q, not $2q$).

First, you can now see why the methods for sampling g above work:

1. If you pick an arbitray g and check that $g^q = \bmod p$ you get the right order. The check tells you that the order of g is at most q. The order is not less than q: it cannot be 1 because you picked $g \neq 1$. It cannot be 2 since the only element such that $g^2 = g^q = 1 \bmod p$ is 1.
2. If you generate g as $g = x^2 \bmod p$ for some $x \notin \{1, -1\}$, you can see that $g^q = 1 \bmod p$ since $g^q = (x^2)^q = x^{2q} = 1 \bmod p$ and you can be sure that x doesn't have order 1 or 2 since you checked that $x \notin \{1, -1\}$.

If you don't work in the prime-order subgroup i.e., if you use g of order $p-1 = 2q$ then the DDH problem is not hard. (We ignore here g's of order 1 or 2 since this is clearly insecure for cryptography as the group is too small, but of course this is something that should be checked and can otherwise lead to attacks). Remember that the DDH problem asks you, given a tuple $(g, A, B, C) = (g, g^a, g^b, g^c)$ to determine whether $c = ab$ or not. Suppose now you've picked $g \in \mathbb{Z}_p^*$ of order $2q = p-1$. You can now attack the DDH assumption by computing the values (g^q, A^q, B^q, C^q).

Each of these values is going to be either 1 or -1 as discussed above. In particular $g^q = -1 \bmod p$. What about the others? If $a = 2t$ is even, then $A^q = (g^a)^q = (g^{2t})^a = (g^{2q})^t = 1 \bmod p$. If a is odd then $A^q = -1 \bmod p$. The same holds for b and c. Now note that when $c = ab$ then you can compute the parity of c from the parity of a, b. When c is random it will have random parity. So if you see a mismatch between the parity of c and the parity of a, b you can distinguish DDH-tuples from non DDH-tuples with much more than negligible probability.

Encoding the message - How? When you encrypt using ElGamal you need to encode the message $m < p$ into an element of the group generated by g. You can do this in several ways:

1. If m is small you can encode m as $M = g^m$: this guarantees that you end up in the right group. However this encoding cannot be efficiently inverted unless m is small (since you need to compute a discrete logarithm).
2. You can always use a rejection sampling method like the one discussed for finding g above, using m as the "starting point". For instance you encode m as $M = (m, ctr)$ for some counter ctr which starts at 0. You then check whether M is in the group or not. If it is not, you increase the counter. When decrypting, you recover $M = (m, ctr)$ and you simply discard ctr.
3. The best method was given in the cryptography notes: If you want to encrypt m, check if $(m+1)^q = 1 \bmod p$. If yes, encrypt $M = m + 1$. If not, encrypt $M = -(m+1)$. When you decrypt, if $M \leq q$ output $m = M - 1$m otherwise output $m = -M - 1$.

Encoding the message - Why? We have already argued that the first two methods produce an element of the right prime-order subgroup. For the third method, when $(m+1)^q = 1 \bmod p$ then by definition M is the right subgroup of order q. When this is not the case we know that $(m+1)^q = -1 \bmod p$. Therefore the method tells you to encrypt $M = (-1) \cdot (m+1)$, and therefore $M^q = (-1)^q(-1) = 1 \bmod p$. Decryption works

Why is it so important to encode the message within the group? Here are some examples of things that can go wrong if you encrypt something which is not in the right subgroup. Remember that an ElGamal encryption is composed of two parts $(c_1, c_2) = (g^r \bmod p, m \cdot h^r \bmod p)$ where r is a random number between 1 and $q-1$ and $h = g^\alpha \bmod p$ where α, the secret key, is also a random number between 1 and $q-1$.

1. First, it's very easy to see that encrypting 0 is a really bad idea. Note that 0 is of course not even an element of the multiplicative group $\mathbb{Z}_p^*$. If $m = 0$ you get of course that

   $$(c_1, c_2) = (g^r \bmod p, m \cdot h^r \bmod p) = (g^r \bmod p, 0)$$

   So the second component of the ciphertext is 0, which is easy to distinguish from an encryption of anything else in $\mathbb{Z}_p^*$.
2. It is also a bad idea to encrypt -1 (remember that $-1 = p - 1 \bmod p$). This is an element of $\mathbb{Z}_p^*$ but it is not an element of the subgroup G of order q. In fact -1 has order 2 since $(-1)^2 \bmod p$. In this case the attack is slightly more subtle, and involves computing $c_2^q \bmod p$. You can see that

   $$c_2^q = (mh^r)^q = m^q = (-1)^q = -1 \bmod p$$

   The second equality holds since h is of order q so $h^q = 1 \bmod p$. In this case it is easy to distinguish between an encryption of -1 and an encryption of any m of order q (if m has order q then $c_2^q = 1 \bmod p$).
3. In general, it is a bad idea to encrypt any other element which is not in the group of order q. Let's see what happens if you encrypt a message m of order $p - 1 = 2q$. We've already argued that in this case $m^q = -1 \bmod p$ and therefore $(c_2)^q = -1 \bmod p$ (as explained in the previous bullet). This means you can easily distinguish between messages of order $2q$ and elements of order q, which essentially means that one bit of the message leaks with the encryption.