16-124-836
Marcel ZAUDER

10.1 ElGamal Encryption

10.1.a *Suppose you are given two ElGamal encryption of two unknown plaintexts $m_1, m_2 \in G$. Show how to construct a ciphertext that decrypts to their product $m_1 \cdot m_2$.*

We have given two encrypted plaintexts in the form of:

$$\begin{aligned} Enc(x, m_1) &= (R_1, C_1) = (g^{r_1}, m_1 \cdot Y^{r_1}) \\ Enc(x, m_2) &= (R_2, C_2) = (g^{r_2}, m_2 \cdot Y^{r_2}) \end{aligned}$$

We consider $R = g^{r_1} \cdot g^{r_2} = g^{r_1+r_2}$ and $C = m_1 \cdot Y^{r_1} \cdot m_2 \cdot Y^{r_2} = m_1 m_2 \cdot Y^{r_1+r_2}$. For the decoding process we then get:

$$\begin{aligned} Dec(x, (R, C)) &= C / R^x \\ &= \frac{m_1 m_2 \cdot Y^{r_1+r_2}}{g^{r_1+r_2 x}} \\ &= \frac{m_1 m_2 \cdot g^{x(r_1+r_2)}}{g^{(r_1+r_2)x}} \\ &= m_1 \cdot m_2 \end{aligned}$$

10.1.b *Suppose you are given an ElGamal encryption of an unknown plaintext $m \in G$. Show how to construct a different ciphertext that also decrypts to the same m.*

We consider the following:
$Y, x := KeyGen()$, $R_1, C_1 := Enc(Y, m)$
pick $r_2 \leftarrow \mathbb{Z}_q$ **such that** $Y^{r_2} \neq Y^{r_1}$.
Let $R_2, C_2 := (g^{r_2}, m \cdot Y^{r_2})$.
We then obvious have:

$$C_2 = m \cdot Y^{r_2} \neq m \cdot Y^{r_1} = C_1$$

But furthermore we get:

$$Dec(x, (R_2, C_2)) = m = Dec(x, (R_1, C_1))$$

10.1.c *A lazy Bob encrypts two distinct messages $m_1; m_2 \in G$ for Alice by using the same value R for both encryptions. How secure is this?*

If $R = g^r$ is reused to encrypt two messages m_1, m_2, then $C_1 = m_1 \cdot Y^r$, and $C_2 = m_2 \cdot Y^r$. Therefore we can compute the following:

$$\frac{C_1}{C_2} = \frac{m_1 \cdot Y^r}{m_2 \cdot Y^r} = \frac{m_1}{m_2}$$

Because we can get some information about a message if the other one is known, this method is not as secure as it would be if two different R are used.

10.2 Ciphertext size of CPA-secure public-key encryption

If the length of the ciphertext is $\alpha \log(\lambda)$, then there exist at most $n := 2^{\alpha \log(\lambda)}$ cipher texts. Note that for sufficiently large values of λ, $2^{\log(\lambda)} \leq \lambda$ - with equality in the case of the logarithm with base 2 - as such $n \leq \lambda^\alpha$ is bounded by a polynomial function.

If the number of ciphertexts is bounded by a polynomial function, a polynomial attacker is able to enumerate a non-negligible quantity of the cipher text space. Consider the following distinguisher:

```
Expected value of coupon collector's problem
tableSize := n · H_n
knownCtxts := Empty set
p_k := GETKEY()
for i = 1 to tableSize:
  knownCtxts « Σ.Enc(p_k, 0)
c := TEST(0, 1)
if c in knownCtxts:
  return 1
else:
  return 0
```

If n is bounded by a polynomial function then the expected value of the coupon collector's problem $n \cdot H_n$, with H_n the n-th harmonic number, is bounded by a polynomial function too, so such an enumeration can be done.

As such:

$$P\left[A \diamond L^{\Sigma}_{\text{pk-CPA-R}} \to 1\right] = 0$$

And with the chosen value of n:

$$P\left[A \diamond L^{\Sigma}_{\text{pk-CPA-L}} \to 1\right] = 0.5$$

Therefore the Bias is:

$$Bias(A) = \mid P\left[A \diamond L^{\Sigma}_{\text{pk-CPA-R}} \to 1\right] - P\left[A \diamond L^{\Sigma}_{\text{pk-CPA-L}} \to 1\right] = 1 - 0.5 = 0.5$$

Therefore it is not pk-CPA-secure.

10.3 Unbounded power

Consider the following brute-force algorithm which, given a ciphertext c and public key p_k computes the corresponding plaintext m:

```
COMPUTE(c, p_k):
while TRUE:
  if c = Σ.Enc(p_k, 0):
    return 0
  if c = Σ.Enc(p_k, 1):
    return 1
```

Such an algorithm clearly has non-polynomial running time as the size of the space of cipher texts is not polynomial in the security parameter λ. It is, however guaranteed to terminate in finite time, as each attempt at computing the plaintext has a probability of success greater than zero, so the cumulative probability of success approaches 1 as the amount of iterations increases. In other words the probability of not terminating approaches zero as the number of iterations approaches infinity.

Other possibly more efficient alternatives exist as well, such as repeatedly calling $(p'_k, s'_k) := KeyGen()$ until $p'_k = p_k$, in which case s'_k is the used secret key.