GENERALIZATION AND IMPLEMENTATION OF PAILLIER PROTOCOL

ANDREW TSENG: ART2589

Abstract. Electronic voting has been tested throughout by countries like Estonia and security has been a main issue. We look into the general characteristics of the Paillier protocol and how it is relevant to electronic voting in general. There is also an implementation of the Paillier protocol in Python 2.7 along with tests for its correctness and looks for vulnerabilities in the cryptographic system. We talk about the performance of the implementation and the provide explicit documentation to the code written in gen.py, encrypt.py, decrypt,py.

Introduction to Paillier Protocol

Electronic voting utilizes various forms of cryptographic protocols and one of them is the Paillier protocol. The Paillier protocol is a probabilistic encryption scheme that generates keys off computation of random values from the group Z_n^*. The encryption process of Paillier results from generating a random integer and using the RSA modulus of n. A favorable characteristic of the encryption process is the homomorphic flavor of this protocol which makes it favorable for electronic voting. The security of the Paillier protocol relies on the fact that it is difficult to figure out if two elements lie within the same coset or not.

Benefits to Paillier

The private key for a cryptographic system must have a hash function that can create signatures that do not duplicate across voters throughout the large population of participants. The Paillier protocol requires a finding two large prime numbers to generate a n in order to create a large subset of values to generate public and private keys during the key-gen phase.

Homomorphism allows for identity values with signatures to correlate to tallying while still encrypted. Say Alice sends cypher c_1 for m_1 and Bob sends cypher c_2 for m_2, then the cypher c_1c_2 is an encryption of m_1m_2. This allows for candidates to be tallied by the signature of the voter and prevents the public from seeing who voted for who, which can be controversial. From the standpoint of the key-gen phase, it is very difficult to get the cipher text c, choice of g as this pair was randomly generated.

This allows the encrypted identity of the voters to stay encrypted when it comes to couting the number of votes for the candidates. The protocol is oftenly named one-way as a result of it.

Because Paillier is a public key encryption scheme, the need to send the cypher to more than twice as knowing the cypher does not help with decrypting the message. Sending c and c' wouldn't decrypt to the same message because of the scheme and c will blind itself from the original message m.

Explanation of Protocol

Key Generator - The public and private keys are generated during this phase by choosing a random p, q such that they are independent and prime from eachother. The modulo $n = pq$

where the RSA modulos is set for the protocol. One way to ensure these two prime numbers are independent is to make sure that $p, q \in 0, 1^s$ meaning they have equivalent bit length, and prime but not equal. The goal of the key-gen phase is to generate $(n, g), (\lambda, u)$.

Encryption Phase - Given message m, a randomly generated $r \in Z_n^*$, and the public key (n, g). The cipher $c = g^m * r^n \mod n^2$ is the encryption of m.

Decryption Phase Given cipher c and the private key (λ, u) We decrypt $m = (L(c^\lambda \mod n^2) * u) \mod n$. A way to increase the performace of the decryption is the calculation of the L of the function.

The decryption process involves solving a discrete log problem by taking cipher text c and computing it by $m = L(c^\lambda \mod n^2) * eu \mod n)$, where

$$L(x) = \frac{x-1}{n}$$

$$\lambda = lcm(p-1, q-1)$$

Documentation of Implementation

gen.py generates the private key (λ, u) and public key (n, g).
The private key is used for the decryption process while the public is used for encryption. (decrypt.pyt and encrypt.py)

Function **gen_ab** generates α, β which are used to randomly generate g under the group.

Function **gen_g** is a second way to generate a $g \in Z_{n^2}^*$ where $\gcd(\frac{g^\lambda \mod (n^2-1)}{n}, n) = 1$

Function **calc_u** calculates the u used for the private key given g, n, λ.

encrypt.py Given message m r is selected randomly from the group $r \in Z_n^*$ The ciphertext $c = g^m * r^n \mod n^2$, where g is randomly selected from $Z_{n^2}^*$ and suffices the following requirement: $\gcd(\frac{g^\lambda \mod (n^2-1)}{n}, n) = 1$ The program takes in a c, λ, u, n in that order from input.txt

decrypt.py Given ciphertext $c \in Z_{n^2}^*$, the message
$m = L(c^\lambda \mod n^2) * u \mod n$.

Weaknesses and Attacks

Producing a small p, q where the values are small can lead to the easy finding of n which destroys the homomorphism of the protocol. The reason for this is from finding n would reduce solving c to a easily factoring n. Increasing the lower bound that p, q can be generated to from the random function will prevent the n from being too small for the protocol to encrypt the message.

Performance of the Implementation

As a result of the random generating p, q and to have them be prime and independent from one another, the time it takes for the key generator to execute and produce the public key and private key varies greatly on how quickly **gen_pq** function can find the pair (p, q) that satisfies

the requirement.

Varying results of the runtime of gen.py specifically.

84.1 sec
7.43 sec
27.9 sec
70.9 sec

The time it takes to find a pair of p, q depends on how quickly the random number generator generates a pair of prime numbers with the same bit length such that $p, q \in \{0, 1\}^{s-1}$ where s is the bit length.

Reference

Jurik, Mads and Ivan Damgard, *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting*, Aarhus University.

Hoffstein, Jeffrey and Pipher, Jill and Silverman, Joseph H, *An Introduction to Cryptography, 2nd Edition*, Springer 2014.