Cayley-Purser Algorithm

Analysis of the public-key algorithm

Introduction

The Cayley-Purser Algorithm is a public-key algorithm designed to encrypt a message through matrix algebra, relying on the non-communitive nature of matrix multiplication. The theory was published in 1999, which happens to be the year of my birth, by Sarah Flannery from Ireland. Flannery was only 16 when showed the elegant encryption method, as well as achieving the *EU Young Scientist of the Year Award* in the same year (Stinson). Congratulations to her accomplishments.

Theory

All the derivations and equations can be found in Flannery's original publication, reproduced on a website here.

Key generation:

Begin by generating two large primes: p, q and their product $n = p * q$. Create two large-numbered 2×2 matrices: χ, α, following modular arithmetic **mod** n, also known as $GL(2, n)$, and such that $\chi * \alpha^{-1} \neq \alpha * \chi$. Let $\beta = \chi^{-1} * \alpha^{-1} * \chi$. Choose a large natural number r, and let $\gamma = \chi^r$.

Then, the public key is n, α, β, γ. The private key is n, χ.

Encryption:

Sender must generate a random natural number s, and compute:

$$\text{i.}\ \delta = \gamma^s$$
$$\text{ii.}\ \epsilon = \delta^{-1} * \alpha * \delta$$
$$\text{iii.}\ \kappa = \delta^{-1} * \beta * \delta$$

Represent every message block as a number, thus generating a list of numbers.

Push the numbers to be elements of a $2x2$ Matrix μ, and for every μ:

$$\mu' = \kappa * \mu * \kappa$$

Send ϵ and every μ' to the receiver.

Decryption:

Recover original through $\lambda = \chi^{-1} * \epsilon * \chi$, and

$\mu = \lambda * \mu' * \lambda$, proven through the following substitutions and cancellations:

$$\mu = (\lambda * \kappa) * \mu * (\kappa * \lambda)$$
$$\mu = (\chi^{-1} * \epsilon * \chi * \kappa) * \mu * (\kappa * \chi^{-1} * \epsilon * \chi)$$
$$\mu = (\chi^{-1} * \epsilon * \chi * \delta^{-1} * \beta * \delta) * \mu * (\delta^{-1} * \beta * \delta * \chi^{-1} * \epsilon * \chi)$$
$$\mu = (\chi^{-1} * \delta^{-1} * \alpha * \gamma^s * \chi * \delta^{-1} * \chi^{-1} * \alpha^{-1} * \chi * \gamma^s) * \mu * (\delta^{-1} * \chi^{-1} * \alpha^{-1} * \chi * \gamma^s * \chi^{-1} * \delta^{-1} * \alpha * \gamma^s * \chi)$$
$$\mu = (\chi^{-1} * \chi^{-rs} * \alpha * \chi^{rs} * \chi * \chi^{-rs} * \chi^{-1} * \alpha^{-1} * \chi * \chi^{rs}) * \mu * (\chi^{-rs} * \chi^{-1} * \alpha^{-1} * \chi * \chi^{rs} * \chi^{-1} * \chi^{-rs} * \alpha * \chi^{rs} * \chi)$$

$$\mu = \left(\chi^{-(rs+1)} * \alpha * \chi^{rs+1} * \chi^{-(rs+1)} * \alpha^{-1} * \chi^{rs+1}\right) * \mu * \left(\chi^{-(rs+1)} * \alpha^{-1} * \chi^{rs+1} * \chi^{-(rs+1)} * \alpha * \chi^{rs+1}\right)$$
$$\mu = \left(\chi^{-(rs+1)} * \alpha * I * \alpha^{-1} * \chi^{rs+1}\right) * \mu * \left(\chi^{-(rs+1)} * \alpha^{-1} * I * \alpha * \chi^{rs+1}\right)$$

$$\mu = \left(\chi^{-(rs+1)} * I * \chi^{rs+1}\right) * \mu * \left(\chi^{-(rs+1)} * I * \chi^{rs+1}\right)$$
$$\mu = I * \mu * I = \mu.$$

Code

Overview:

First, an alphabet must be considered. For this algorithm, any alphabet will work, as it converts the message into a numeric representation through the use of the StringToList method used in the course. If any characters are found which do not exist in the Alphabet, they are discarded with a warning to the user. Then, the base is converted to base n.

The proper keys must also be generated. As mentioned previously, the public key is a collection of n, α, β, γ while the private is n, χ. To construct these, a method called getKeys is called, with a parameter to choose the magnitude of the prime numbers p, q. This procedure will generate random, valid values for the variables. p, q are obtained through getRandPrime, which will return the next known prime between $10^{magnitude}$ and $10^{magnitude+1}$ after a random value between this range is given through the BlumBlumShub method. Having p, q thus gives us n.

Next, α and χ are constructed by randomly choosing 8 elements to go into the two matrices, obtained through the getRandVal procedure with an arbitrary range between 0 to 25,000 $\mathbf{mod}\ n$, though this can be adjusted. They need just be checked that $\chi\alpha^{-1} \neq \alpha\chi$, if it is otherwise, then replace the elements and try again. β is obtained through matrix inversion and multiplication, while r is randomly chosen between 50 and 100, to produce $\gamma = \chi^r$. Many of these values can be larger, but for the sake of understandable running time, I didn't make it too large.

So, we have all the public and private parameters, as well as a list of numbers representing our super-secret message. The next step is to apply the ListToMatrix method, unambiguously converting our list of integers to matrices holding the integers as elements in a list. Not knowing how many matrices to produce beforehand, as the size of the original list will vary from one message to another, the elements of the list are inserted through a queue, and every time a matrix is filled, it is added to a matrix queue, when one matrix is filled up, it's enqueued and another takes its place. When the integer queue becomes empty, the remaining, empty elements of the final matrix are assigned 0. Finally, the matrix queue is converted to a list of matrices and returned.

Encryption is completed by taking $\delta = \gamma^s, \kappa = \delta^{-1} * \beta * \delta,$ and using modular matrix multiplication to come up with a unique $\mu' = \kappa * \mu * \kappa$ for every μ matrix of the matrix list. s is just a random positive integer chosen arbitrarily from 10 to 100. What must be noted is the need to also provide ϵ to the recipient, where $\epsilon = \delta^{-1} * \alpha * \delta$.

Finally, Decryption is completed by taking the given ϵ, which essentially communicates to the recipient the final piece to the puzzle of how the message was encrypted, and combining it with the private key χ in the form of $\lambda = \chi^{-1} * \epsilon * \chi$ to map every matrix μ' to the decrypted μ by solving $\mu = \lambda * \mu' * \lambda$. The last step is to take the matrix and turn it back to a string by taking the matrix elements, turning them back into a list, and then turning them back to the proper strings with the proper base n.

Weaknesses:

The method can be undermined since it is not just χ which works to calculate λ,in fact, any scalar multiple c will still satisfy $\lambda = \chi^{-1}\epsilon\chi = (c * \chi)^{-1} * \epsilon * (c * \chi)$. Therefore, a linear algebra attack is sufficient to deduce λ (Stinson).

Unfortunately, when the press learned about Flannery's success, they advertised it as comparable and faster than RSA, without giving enough time for scholarly research to evaluate it. Nonetheless, Flannery gave a very insightful technical description and analysis of the algorithm, including its insecurity, as well recounting her experience (Stinson).

Stinson: arxiv at arXiv:1803.05004,

Stinson, Douglas R. “A Brief Retrospective Look at the Cayley-Purser Public-Key Cryptosystem, 19 Years Later.” *International Association for Cryptologic Research*, 13 Mar. 2018, eprint.iacr.org/2018/270.pdf.