Cryptography:
An Investigation of a New Algorithm vs. the RSA

Sarah Flannery, Blarney, Co. Cork, Ireland

Contents

Cryptography:

An Investigation of A New Algorithm vs. the RSA

Introduction

As long as there are creatures endowed with language there will be the desire for confidential communication - messages intended for a limited audience. Governments, companies and individuals have a need to send or store information in such a way that only the intended recipient is able to read it. Generals send orders, banks send fund transfers and indviduals make purchases using credit cards. Cryptography is the study of methods to 'disguise' information so that only the intended reciepient can obtain knowledge of its content. Public-Key Cryptography was first suggested in 1976 by Diffie and Hellman and a public-key cryptosystem is one which has the property that someone who knows only how encipher ('disguise') a piece of information CANNOT use the enciphering key to find the deciphering key wothout a prohibitively lenghtly computation. This means that the information necessary to send private or secret messages, the enciphering algorithm along with the enciphering key, can be made public-knowledge by submitting them to a public directory. The first public-key cryptosystem, the RSA Algorithm, was developed by Ronald Rivest, Adi Shamir and Leonard Adleman at MIT in 1977. This system, described below, has stood the test of time and is today recognised as a standard of encryption worldwide.

Aim

This project investigates a possible new public-key algorithm, entitled the Cayley-Purser (CP) Algorithm and compares it to the celebrated RSA public-key algorithm. It is hoped to show that the CP Algorithm is

- As secure as the RSA Algorithm and
- FASTER than the RSA Algorithm.

Firstly both algorithms are presented and why they both work is illustrated. A mathematical investigation into the security of the Cayley-Purser algorithm is discussed in the main body of the report. Some differences between the RSA and the CP algorithms are then set out. Both algorithms are programmed using the mathematical package *Mathematica* and the results of an empirical run-time analysis are presented to illustrate the relative speed of the CP Algorithm.

RSA Public Key Cryptosystem

The RSA scheme works as follows:

Start Up: [This need only be done once.]

- Generate at *random* two prime numbers p and q of 100 digits or more.
- Calculate $n = pq$ and $\phi(n) = (p-1)(q-1) = n - (p+q) + 1$.
- Generate at random a number $c < \phi(n)$ such that $(c, \phi(n)) = 1$.
- Calculate the *multiplicative inverse*, d, of $c \pmod{\phi(n)}$ using the Euclidean algorithm.

Publish: Make public the enciphering key.

$$K_E = (n, e)$$

Keep Secret: Conceal the deciphering key,

$$K_D = (n, d)$$

Enciphering: The enciphering transformation is,

$$C = f(P) \equiv P^e \pmod{n}$$

Deciphering: The deciphering transformation is,

$$P = f^{-1}(C) \equiv C^d \pmod{n}$$

Why the deciphering works:- The correctness of the deciphering algorithm is based on the following result due to Euler, which is a generalization of what is known as Fermat's little theorem. This result states that,

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

whenever $(a, n) = 1$, where $\phi(n)$, Euler's-ϕ function, is the number of positive integers less than n which are relatively prime to n.

When $n = p$, a prime, $\phi(n) = p - 1$, and we have Fermat's theorem:

$$a^{p-1} \equiv 1 \pmod{p} \quad ; \quad (a, p) = 1$$

If $p|a$ then $a^p \equiv a \equiv 0 \pmod{p}$, so that for any a,

$$a^p \equiv a \pmod{p}$$

Now since d is the multiplicative inverse of e, we have

$$ed \equiv 1 \pmod{\phi(n)} \quad \Rightarrow \quad ed = 1 + k\phi(n), \qquad k \in \mathbb{Z}$$

Now

$$f^{-1}(f(P)) = (P^e)^d \equiv P^{ed} \pmod{n}$$

and

$$P^{ed} \equiv P^{1+k\phi(n)} \pmod{n} \qquad \text{(for some integer } k\text{)}$$

Now for P with $(P, p) = 1$, we have

$$P^{p-1} \equiv 1 \pmod{p} \quad \Rightarrow \quad P^{k\phi(n)+1} \equiv P \pmod{p} \quad \text{as} \quad p - 1 | \phi(n).$$

This is trivially true when $P \equiv 0 \pmod{p}$, so that *for all* P, we have,

$$P^{ed} \equiv P^{1+k\phi(n)} \equiv P \pmod{p}$$

Arguing similarly for q, we have for all P,

$$P^{ed} \equiv P^{1+k\phi(n)} \equiv P \pmod{q}$$

Since p and q are relatively prime, together these equations imply that for all P,

$$P^{ed} \equiv P^{1+k\phi(n)} \equiv P \pmod{n}.$$

The Cayley-Purser Algorithm

Introduction

Since this algorithm uses 2×2 matrices and ideas due to Purser it is called the Cayley-Purser Algorithm. The matrices used are chosen from the multiplicative group $G = GL(2, \mathbb{Z}_n)$. The modulus $n = pq$, where p and q are both primes of 100 digits or more, is made public along with certain other parameters which will be described presently. Since

$$|GL(2, \mathbb{Z}_n)| = n\phi(n)^2(p+1)(q+1)$$

we note that the order of G cannot be determined from a knowledge of n alone.

Plaintext message blocks are assigned numerical equivalents as in the RSA and placed four at a time in the four positions (ordered on the first index) of a 2×2 matrix . This message matrix is then transformed into a cipher matrix by the algorithm and the corresponding ciphertext is then extracted by reversing the assignment procedures used in encipherment.

Because this algorithm uses nothing more than matrix multiplication (modulo n) and not modular exponentiation as required by the RSA it might be expected to encipher and decipher considerably faster than the RSA.. This question was investigated, using the mathematical package **Mathematica**, by applying both algorithms to large bodies of text (see Tables I - IX) and it was found that the Cayley-Purser algorithm was approximately twenty - two times faster the RSA with respect to a $200 - digit$ modulus.

Needless to say if it could be shown that this algorithm is as secure as the RSA then it would recommend itself on speed grounds alone. The question of the security of this algorithm is discussed after we have described it and explained why it works.

The CP Algorithm

Start Up procedure to be followed by B (the receiver):

- Generate two large primes p and q.
- Calculate the modulus $n = pq$.
- Determine χ and $\alpha \in GL(2, \mathbb{Z}_n)$ such that $\chi\alpha^{-1} \neq \alpha\chi$.
- Calculate $\beta = \chi^{-1}\alpha^{-1}\chi$.
- Calculate $\gamma = \chi^r \quad ; \quad r \in \mathbb{N}$.

Publish: The modulus n and the parameters α, β, and γ

Start Up procedure to be followed by A (the sender):

In order to encipher the matrix μ corresponding to a plaintext message unit for sending to B, person A must consult the parameters made public by B and do the following:

- Generate a random $s \in \mathbb{N}$
- Calculate $\delta = \gamma^s$
- Calculate $\epsilon = \delta^{-1}\alpha\delta$
- Calculate $\kappa = \delta^{-1}\beta\delta$

Enciphering Procedure When the above parameters are calculated, A enciphers μ via

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

and sends μ' and ϵ to B

Deciphering Procedure When A receives μ' and ϵ (s)he does the following:

Calculates

$$\lambda = \chi^{-1}\epsilon\chi$$

and deciphers μ' via

$$\mu = \lambda\mu'\lambda$$

Why the deciphering works.

The deciphering works since

$$\begin{aligned}
\lambda &= \chi^{-1}\epsilon\chi \\
&= \chi^{-1}(\delta^{-1}\alpha\delta)\chi \\
&= \delta^{-1}(\chi^{-1}\alpha\chi)\delta && : \quad (\delta, \text{ being apower of } \chi, \text{ commutes with } \chi) \\
&= \delta^{-1}(\chi^{-1}\alpha^{-1}\chi)^{-1}\delta \\
&= \delta^{-1}\gamma^{-1}\delta && : \quad (\text{recall that } \beta = \chi^{-1}\alpha^{-1}\chi) \\
&= (\delta^{-1}\beta\delta)^{-1} \\
&= \kappa^{-1} && : \quad (\text{B's enciphering key})
\end{aligned}$$

so that

$$\begin{aligned}
\lambda\mu'\lambda &= \lambda(\kappa\mu\kappa)\lambda \\
&= (\kappa^{-1}\kappa)\mu(\kappa\kappa^{-1}) \\
&= \mu.
\end{aligned}$$

Wherein lies the security of the Cayley-Purser Algorithm ?

To find the secret matrix χ, known to B alone, one might attempt to solve either the equation

$$\beta = \chi^{-1} a^{-1} \chi$$

or

$$\gamma = \chi^r$$

In the first of these equations the matrix β is public and the matrix a^{-1} can be computed since both the matrix a and the modulus n are public.

In the second equation only the matrix γ is known and it is required to solve for both the exponent r and the base matrix χ. Assuming that one knew r, solving this equation would involve extracting the r^{th} - roots of a matrix modulo the composite integer n. Even in the simplest case, where $r = 2$, extracting the square root of a 2×2 matrix modulo n requires that one be able to solve the ordinary *quadratic* conruence

$$x^2 \equiv a \pmod{n}.$$

when $n = pq$. It is known that the ability to solve this 'square root' problem is equivalent to being able to factor n. Thus we may regard an attack on χ via the public parameter γ as being computationally prohibitive.

Solving the equation

$$\beta = \chi^{-1} a^{-1} \chi$$

would appear the easier option for an attack on the private matrix χ as it only involves solving the set of linear equations given by

$$\chi \beta = a^{-1} \chi$$

However the number of possible solutions to this equation is given by the order of $C(a)$, the centraliser of a in $GL(2, \mathbf{Z}_n)$. By ensuring that the order of this group is extremely large one can make it computationally prohibitive to search for χ.

To see why this is the case suppose that

$$\beta = \chi^{-1} a^{-1} \chi \text{ and } \beta = \chi_1^{-1} a^{-1} \chi_1$$

Then

$$\chi^{-1} a^{-1} \chi = \chi_1^{-1} a^{-1} \chi_1$$

if and only if

$$a^{-1} \chi \chi_1^{-1} = \chi \chi_1^{-1} a^{-1}$$

if and only if

$$\chi \chi_1^{-1} \in C(a^{-1})$$

if and only if

$$\chi \in C(a^{-1}) \chi_1$$

Thus the number of distinct solutions of the equation is given by $|C(a)|$ as $C(a^{-1}) = C(a)$.

Now $C(a)$ will have a large order if the matrix element a has a large order.

By choosing our primes p and q to be of the form

$$p = 2p^1 + 1$$

and

$$q = 2q^1 + 1.$$

where p^1 and q^1 are themselves prime, we can show that it is almost certainly the case that an element a chosen at random from $GL(2, \mathbf{Z}_n)$ has a large order.

To see why, we begin by considering the homomorphism ϕ of $GL(2, \mathbf{Z}_n)$ onto $\mathbf{Z}_n$ defined by sending a matrix into its determinant. The order of a matrix in $GL(2, \mathbf{Z}_n)$ is at least that of the order of its image in $\mathbf{Z}_n$ since ...

If r is the order of A in $GL(2, \mathbf{Z}_n)$ and $\phi(A) = u$ then $A^r = I$ with

$$1 = \phi(I) = \phi(A^r) = \phi(A)^r = u^r$$

shows that m divides r where m is the order of u in $\mathbf{Z}_n$.

Thus the order of A in $GL(2, \mathbf{Z}_n)$ is at least m. In fact

$$\phi(A^m) = \phi(A)^m = u^m = 1$$

shows that A^m lies in $SL(2, \mathbf{Z}_n)$ so the matrix A will have order μ iff $A^m = I$ in $SL(2, \mathbf{Z}_n)$.

We note also that since the maximum achievable order of an element in $\mathbf{Z}_n$ is

$$[p-1, q-1] \leq \frac{(p-1)(q-1)}{2} = \frac{\phi(n)}{2}$$

(as $(p-1, q-1) \geq 2$) and since the order of $SL(2, \mathbf{Z}_n)$ is $n\phi(n)(p+1)(q+1)$ the maximum achievable order of a matrix in $GL(2, \mathbf{Z}_n)$ is

$$[p-1, q-1]\, n\phi(n)(p+1)(q+1) \leq n\phi(n)2(p+1)(q+1)/2 = |GL(2, \mathbf{Z}_n)|/2.$$

Thus if we can show that the probability of an element having a small order in $\mathbf{Z}_n$is negligibly small then we will have shown that the order of an element chosen at random from $GL(2, \mathbf{Z}_n)$ is almost certainly of 'high order.'

If

$$p = 2p^1 + 1$$

and

$$q = 2q^1 + 1$$

then

$$\phi(n) = \phi(pq) = (p-1)(q-1) = 2p^1 2q^1 = 4p^1 q^1$$

with

$$[p-1, q-1] = [2p^1, 2q^1] = 2p^1 q^1 = \phi(n)/2$$

Now the possible orders of the elements in $\mathbf{Z}_n$ are divisors of $\phi(n)\ 2 = 2p^1 q^1$ and so are

$$1,\ 2,\ p^1,\ q^1,\ 2p^1,\ 2q^1,\ p^1 q^1,\ 2p^1 q^1$$

and all of these orders are achieved by some elements. In fact by counting exactly how many elements correspond to each order we show that the probability of finding a unit in $\mathbf{Z}_n$ of *order less than* $p^1 q^1$ is negligibly small.

Recall that if a in $\mathbb{Z}_p$ has order k and b in $\mathbb{Z}_q$ has order l then the order of c in $\mathbb{Z}_n$ where

$$c \equiv a \pmod{p}$$

and

$$c \equiv b \pmod{q}$$

is $[k, l]$, the least common multiple of k and l.
Now the possible orders of a and b in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ are divisors of

$$p-1 = 2p^1 \quad ; \quad q-1 = 2q^1$$

respectively.

The following table lists the possible orders along with the number of elements of each order.

$\mathbb{Z}^*_p$		$\mathbb{Z}^*_q$	
Possible Orders	No. of elements of that order	Possible Orders	No. of elements of that order
1	1	1	1
2	1	2	1
p^1	p^1-1	q^1	q^1-1
$2p^1$	p^1-1	$2q^1$	q^1-1

By lifting elements in pairs via the CRT we obtain the elements corresponding to the different orders in $\mathbb{Z}_n$ along with number of elements of each order.

Order	Number	Reason
1	1	$[1, 1] = 1$
2	3	$[1, 2] = [2, 1] = [2, 2] = 2$
p^1	p^1-1	$[p^1, 1] = p^1$
q^1	q^1-1	$[1, q^1] = q^1$
$2p^1$	$3p^1-3$	$[2p^1, 1] = [p^1, 2] = [2p^1, 2] = 2p^1$
$2q^1$	$3q^1-3$	$[1, 2q^1] = [2, q^1] = [2, 2q^1] = 2q^1$
p^1q^1	$p^1q^1-p^1-q^1+1$	$[p^1, q^1] = p^1q^1$
$2p^1q^1$	$3p^1q^1-3p^1-3q^1+3$	$[2p^1, {}^1] = [p^1, 2q^1] = [2p^1, 2q^1] = 2p^1q^1$

Note that if we sum all the individual counts we get exactly $4p^1q^1$ which is the number of elements of $\mathbb{Z}_n$.

Explanation: To see how the number of elements corresponding to an order is obtained consider the last entry in the above array. An element of order $2p^1q^1$ in $\mathbb{Z}_n$ can be obtained in 3 different ways by lifting pairs of elements from $\mathbb{Z}_p$ and $\mathbb{Z}_q$: One way is lifting the pair (a, b) where a has order $2p^1$ and b has order q^1; another by lifting the pair (a, b) where a has order p^1 and b has order $2q^1$ and another by lifting the pair (a, b) where a has order $2p^1$ and b has order $2q^1$.

Regarding elements of order less than p^1q^1 as elements of 'low order' we obtain the probability of choosing an element of order less than p^1q^1 to be

$$\frac{4p^1 + 4q^1 - 4}{4p^1q^1}$$

This is equivalent to

$$\frac{1}{p^1} + \frac{1}{q^1} - \frac{1}{p^1q^1}$$

In the case where p and q are both of order of magnitude 10^{100} this probability is approximately

$$2.10^{-100}$$

which, by any standards, is negligibly small.

Some Differences between the RSAand Cayley-Purser Algorithms

1. The most significant difference between the RSA and the Cayley-Purser algorithm is the fact that the Cayley-Purser algorithm uses only modular matrix multiplication to encipher plaintext messages whereas the RSA uses modular exponentiation which requires a considerably longer computation time. Even with the powerful Mathematica function PowerMod the RSA appears (see Tables I - IX) to be over 20 times slower than the Cayley-Purser Algorithm.

2. In the RSA the parameters needed to encipher - (n. e) - are published for the whole world to see and anyone who wishes to send a message to Bob raises their messages' numerical equivalents to the power of e modulo n. However in the Cayley-Purser algorithm the enciphering key is not made public ! Only the parameters for calculating one's own key are published. This means that every sender in this system also enjoys a certain measure of secrecy with regard to their own messages. One consequence of this is that the Cayley-Purser algorithm is not susceptible to a repeated encryption attack because the sender, Alice, is the only one who knows the encryption key she used to encipher. In the RSA, however if the order of e can be found then an eavesdropper can decipher messages.

3. Alice can choose to use a new enciphering key every time she wishes to write to Bob. In the unlikely event that an eavesdropper, Eve, should find an enciphering key, she gains information about only one message and no information about the secret matrix c. By contrast, if a piece of intercepted RSA ciphertext leads to Eve being able to decipher (through repeated encryption etc.), then she would be able to decipher all intercepted messages which were enciphered using the public exponent e.

4. In the Cayley-Purser algorithm the sender, Alice, has the ability to decipher the ciphertext which she generates using Bob's public parameters even if she loses the original message (because she knows d and therefore can get the deciphering key, $\kappa^{-1} = \lambda$!) Contrast this to the RSA - Alice cannot decipher her own message once she has enciphered it using Bob's public key parameters. There is a possible advantage in this for Alice in that she could store encrypted messages on her computer ready for sending to Bob.

RSA vs. Cayley-Purser

Empirical Time-Analysis

The times taken by the Cayley-Purser and RSA algorithms (using a modulus n of the order 10^{200}) to encipher single and multiple copies of the Desiderata (1769 characters) by Max Ehrman are given in the following tables along with the times taken by both algorithms to decipher the corresponding ciphertext.

Table I

Running Time (Seconds) Message = 1769 characters				
Trial No.	1	2	3	Average
RSA encipher	41.94	42.1	41.78	41.94
RSA decipher	40.99	41.009	41.019	41.009
C-P encipher	1.893	1.872	1.893	1.886
C-P decipher	1.502	1.492	1.492	1.4953

Table II

Running Time (Seconds) Message = 2 * 1769 = 3538 characters				
Trial No.	1	2	3	Average
RSA encipher	72.364	72.274	72.364	72.334
RSA decipher	70.942	70.952	72.144	71.346
C-P encipher	3.305	3.305	3.325	3.3016
C-P decipher	2.734	2.864	2.864	2.8206

Table III

Running Time (Seconds) Message = 3 * 1769 = 5307 characters				
Trial No.	1	2	3	Average
RSA encipher	103.078	102.808	103.489	103.125
RSA decipher	101.246	101.076	104.06	102.1273
C-P encipher	4.757	4.737	4.747	4.747
C-P decipher	3.976	4.086	4.066	4.0426

Table IV

Running Time (Seconds) Message $= 4 * 1769 = 7076$ characters				
Trial No.	1	2	3	Average
RSA encipher	134.434	134.323	134.333	134.363
RSA decipher	131.128	134.734	134.734	133.532
C-P encipher	6.159	6.048	6.109	6.1053
C-P decipher	5.227	4.967	4.967	5.05536

Table V

Running Time (Seconds) Message $= 12 * 1769 = 21228$ characters				
	RSA enc	RSA dec	C-P enc	C-P dec
Time Taken	378.078	371.254	17.435	14.371

Table VI

Running Time (Seconds) Message $= 24 * 1769 = 42456$ characters				
	RSA enc	RSA dec	C-P enc	C-P dec
Time Taken	509.523	511.455	22.583	18.767

Table VII

Running Time (Seconds) Message $= 48 * 1769 = 84912$ characters				
	RSA enc	RSA dec	C-P enc	C-P dec
Time Taken	1019.24	1023.95	44.894	36.823

Table VIII

Running Time (Seconds) Message $= 144 * 1769 = 254736$ characters				
	RSA enc	RSA dec	C-P enc	C-P dec
Time Taken	3154.21	3036.24	142.775	129.416

With respect to a 133MHz machine the Cayley-Purser Algorithm is on average approximately 22 times faster than the RSA where in each case the modulus n is of the order 10^{200}.

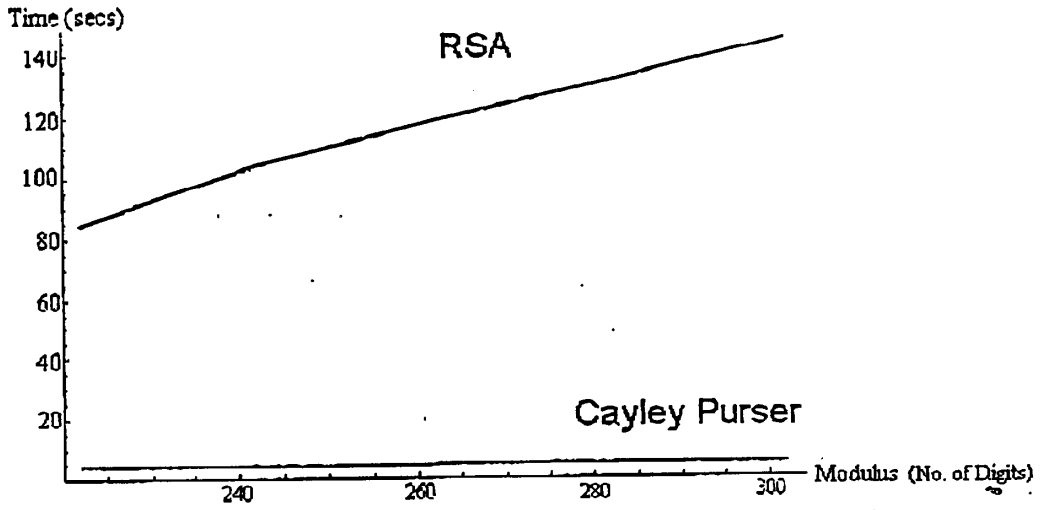
Table IX

The following table illustrates the time taken for the RSA and CP Algorithms to encipher a piece of text (7076 characters in length) with varying size moduli. The ratio of the enciphering speeds is also given.

Running Time (Seconds) Message μ containing 7076 characters			
Modulus	RSA	CP	Ratio
222 digits	84.641	3.916	21.6:1
242 digits	104.71	4.036	25.9:1
262 digits	118.841	4.276	27.8:1
282 digits	131.739	4.326	30.5:1
302 digits	145.689	4.487	32.5:1

Note: . The difference in times taken to encipher and decipher in the RSA depends on the binary weight of the exponents e and d.

Graph 1: Comparison Of Enciphering Times - Cayley-Purser vs. RSA



Number of Desiderata Enciphered

The piece of text used (Desiderata) contains 1769 characters.

Conclusions

This project

- (a) Shows mathematically that the CP Algorithm is as secure as the RSA Algorithm.
- (b) Illustrates through an empirical run-time analysis that the CP Algorithm is FASTER to implement than the RSA Algorithm: the speed factor increasing with modulus size as shown by following table: -

Running Time (Seconds) Message $= 4 * 1769 = 7076$ characters			
Modulus	RSA	CP	Ratio
222 digits	84.641	3.916	21.6:1
242 digits	104.71	4.036	25.9:1
262 digits	118.841	4.276	27.8:1
282 digits	131.739	4.326	30.5:1
302 digits	145.689	4.487	32.5:1

Post Script: An Attack on the CP algorithm.

We describe an attack on the Cayley-Purser algorithm which shows that anyone with a knowledge of the public parameters α, β and γ can form a multiple χ' of χ. This matrix χ' can then be used in conjunction with ϵ to form $\lambda = \kappa^{-1}$ which is the deciphering key. Thus the system as originally set out is 'broken'.

If $\chi' = v\chi$ for some constant v and if ϵ is known to an adversary then the calculation

$$\chi'^{-1}\epsilon\chi = (v^{-1}\chi^{-1})\,\epsilon\,(v\chi) = \chi^{-1}\epsilon\chi = \kappa^{-1}$$

yields the deciphering key κ^{-1} . Thus any multiple of χ can be used to decipher.

In the CP system the matrix γ is made to commute with χ so as to enable the deciphering process. This is done using the construction $\gamma = \chi^r$ for some r and herein lies the weakness of the algorithm. Were γ to be generated more efficiently using a linear combination of χ and the identity matrix $\mathbf{I}$ (higher order polynomials in χ reduce via the Cayley-Hamilton theorem to linear expressions in χ) the system is still compromised.

If the matrix γ is non-derogatory (i.e. when γ is reduced mod p and mod q neither of the two matrices obtained are scalar multiples of the identity) then

$$\chi = u\mathbf{I} + v\gamma$$

(If the matrix γ is derogatory then n can be factorised by calculating GCD $(\gamma_{11} - \gamma_{22}, \gamma_{12}, \gamma_{21}, n)$)

Now since γ is non-derogotary $(v, n) = 1$ and

$$\chi' = v^{-1}\chi = v^{-1}u\mathbf{I} + \gamma = d\mathbf{I} + \gamma$$

for some $d \in \mathbb{Z}_n$.

Since

$$\begin{aligned} \beta &= \chi^{-1}\alpha^{-1}\chi \\ &= v\chi^{-1}\alpha^{-1}v^{-1}\chi \\ &= (v^{-1}\chi)^{-1}\alpha^{-1}(v^{-1}\chi) \\ \Rightarrow \beta &= \chi'^{-1}\alpha^{-1}\chi' \\ \Rightarrow \chi'\beta &= \alpha^{-1}\chi' \end{aligned}$$

Substituting $dI + \gamma$ for χ' in this last equation gives

$$\begin{aligned} [dI + \gamma]\beta &= \alpha^{-1}[dI + \gamma] \\ \Rightarrow d\beta + \gamma\beta &= dI\alpha^{-1} + \alpha^{-1}\gamma \\ \Rightarrow d[\beta - \alpha^{-1}] &= [\alpha^{-1}\gamma - \gamma\beta] \end{aligned}$$

Since $\alpha \neq \beta^{-1}$ these matrices differ in at least one position. For argument's sake let $\alpha_{11} \neq \beta_{11}^{-1}$. Comparing the $(1,1)$ entries in the above matrix identity gives

$$d(\beta_{11}^{-1} - \alpha_{11}) \equiv e \pmod{n} \quad : \quad e \in \mathbb{Z}_n$$

If $(\alpha_{11} - \beta_{11}^{-1})^{-1}$ exists mod n the above linear congruence is uniquely solvable for d. If not a factorisation of n is obtained.

Remark 1:- This attack shows that anyone with a knowledge of the public parameters α, β and γ can form a multiple χ' of χ. This matrix χ' can then be used to form $\lambda = \kappa^{-1}$ *provided* e *is known*. If e is transmitted securely on a once off basis then knowledge of a χ' on its own is not enough to break the system, though then the Cayley-Purser Algorithm would no longer be public-key in nature.

Remark 2: The fact that a derogotary γ leads to a factorisation of the modulus n was further investigated on the asumption that knowledge of n might not severely compromise the system. However in this case also a multiple of χ is obtainable.

Remark 3: An analysis of the CP algorithm based on 3×3 matrices, though slightly more involved in its details, leads to conclusions similar to the ones just described.

Remark 4: For the sake of efficiency δ should be calculated as $\delta = a\gamma + bI$ rather than as $\delta = \gamma^n$

■ Mathematica Code for RSA & CP Algorithms

```
FirstPrimeAbove[n_Integer] :
 (Clear[k]; k = n; While[! PrimeQ[k], k = k + 1]; k)
```

```
ConvertString[str_String] :=
Fold[Plus[256 #1, #2]&, 0, ToCharacterCode[str]]
```

```
StringToList[text_String] := Module[{blockLength =
Floor[N[Log[256, n]]], strLength = StringLength[text]},
ConvertString /@ Table[StringTake[text,{i, Min[strLength, i +
blockLength - 1]}],{i, 1, strLength, blockLength}]]
```

```
ConvertNumber[num_Integer]:=
FromCharacterCode /@ IntegerDigits[num,256]
```

```
ListToString[l_List] :=  StringJoin[ConvertNumber /@ l]
```

■ Mathematica Code for RSA Algorithm

```
GeneratePQNED[digits_Integer] := (p = FirstPrimeAbove[
    prep = Random[Integer, {10^(digits-1), 10^digits - 1}]];
Catch[Do[preq = Random[Integer, {10^(digits - 1), 10^digits - 1}];
If[preq != prep, Throw[q = FirstPrimeAbove[preq]]], {100}]]
   n = p q; e = Random[Integer, {p, n}];
  While[GCD[e, (p - 1) (q - 1)] != 1,
   e = Random[Integer, {p, n}]]; e;
  d = PowerMod[e, -1, (p - 1) (q - 1)];)
```

```
RSAencNumber[num_Integer] := PowerMod[num, e, n]
```

```
RSAdecNumber[num_Integer] :=  PowerMod[num, d, n]
```

```
RSAenc[text_String]:=
RSAencNumber[#]& /@ StringToList[text]
```

```
RSAdec[cipher_List]:=ListToString[RSAdecNumber[#]&/@cipher]
```

■ Mathematica Code for Cayley Purser Algorithm

```
StringToMatrices[text_String]:= Partition[Partition[Flatten
[Append[StringToList[text],{32,32,32}]],2],2]
```

```
MatricesToString[l_List] :=
 StringJoin [ ConvertNumber/@ Flattan[l]]
```

```
CPpqn[digits_Integer] :=Module[{
  p1 = FirstPrimeAbove[ Random[Integer,
       {10^(Floor[digits/2] - 1), 10^(Floor[digits/2]) - 1}]],
  q1 = FirstPrimeAbove[Random[Integer,
       {10^(Floor[digits/2] - 1), 10^(Floor[digits/2]) - 1}]]},
 While[PrimeQ[p = 2 p1 + 1], p1 = FirstPrimeAbove[p1 + 1]]; p;
  While[PrimeQ[q = 2 q1 + 1], q1 = FirstPrimeAbove[q1 + 1]];
  q; n = p q; ]
```

```
randmatrix := (Catch[
   Do[m = Table[Random[Integer, {0, n}], {i, 1, 2}, {j, 1, 2}];
If[GCD[Mod[Det[m], n], n] == 1, Throw[m]], {1000}]])
```

```
inv[a_] := (d = Mod[Det[a], n]; i = PowerMod[d, -1, n];
{{Mod[i * a[[2, 2]], n], Mod[-i * a[[1, 2]], n]},
   {Mod[-i * a[[2, 1]], n], Mod[i * a[[1, 1]], n]}})
```

```
mmul[j_, k_] := Mod[
  {{Mod[j[[1, 1]] * k[[1, 1]], n] + Mod[j[[1, 2]] * k[[2, 1]], n],
    Mod[j[[1, 1]] * k[[1, 2]], n] + Mod[j[[1, 2]] * k[[2, 2]], n]},
   {Mod[j[[2, 1]] * k[[1, 1]], n] + Mod[j[[2, 2]] * k[[2, 1]], n],
    Mod[j[[2, 1]] * k[[1, 2]], n] + Mod[j[[2, 2]] * k[[2, 2]], n]}},
  n]
```

■ Mathematica Code for Cayley Purser Algorithm contd.

```
CPparameters := (identity = {{1, 0}, {0, 1}};
  alpha = randmatrix; Catch[Do[chi = randmatrix;
If[mmul[chi, alpha] != mmul[alpha, chi],
     Throw[chi]], {10000000}]];
  chiinv = inv[chi]; alphainv = inv[alpha];
  Catch[Do[s = Random[Integer, {2, 50}];
    gamma = Mod[MatrixPower[chi, s], n];
    If[gamma != identity, Throw[gamma]], {10000000}]];
Catch[Do[delta = Mod[Mod[Random[Integer, {1, n - 1}] gamma, n]
         + Mod[Random[Integer, {1, n - 1}] identity, n], n]
       If[delta != identity &&
         mmul[delta, alpha] != mmul[alpha, delta], Throw[delta]],
    {10000000}]];
  beta = mmul[mmul[chiinv, alphainv], chi];
  deltainv = inv[delta];
  epsilon = mmul[mmul[deltainv, alpha], delta];
kappa = mmul[mmul[deltainv, beta], delta];
  lambda = mmul[mmul[chiinv, epsilon], chi];)
```

```
CPenc[plain_String] := CPEncNum [ StringToMatrices[plain]]
```

```
CPDecNum[l_List] :=
 Table[mmul[mmul[lambda, l[[i]]], lambda], {i, Length[l]}]
```

```
CPEncNum[l_List] :=
 Table[mmul[mmul[kappa, l[[i]]], kappa], {i, Length[l]}]
```

```
CPdec[cipher_List] := MatricesToString[CPDecNum [ cipher]]
```

Bibliography:-

Higgins. J and Cambell. D: Mathematical Certificates. Math. Mag 67. (1994). 21-28

Mackiw. George: Finite Groups of 2 x 2 Integer Matrices. Math. Mag 69 (1996). 356-361

Meijer. A.R.: Groups. Factoring and Cryptography. Math. Mag 69. (1996). 103-109

Menezes. van Oorschot. Vanstone: Handbook of Applied Cryptography, CRC Press 1996

Salomaa. Arto: Public-Key Cryptography (2 ed.). Springer Verlag 1996

Schneier. Bruce: Applied Cryptography. Wiley 1996

Stangl. Walter D.: Counting Squares in n. Math. Mag 69 (1996). 285-289

Sullivan. Donald: Square Roots of 2 x 2 Matrices. Math. Mag 66 (1993). 314-316