A String-Based Public Key Cryptosystem

M. Andrecut

October 3, 2014

Unlimited Analytics Inc.
Calgary, AB, Canada
mircea.andrecut@gmail.com

Abstract

Traditional methods in public key cryptography are based on number theory, and suffer from problems such as dealing with very large numbers, making key creation cumbersome. Here, we propose a new public key cryptosystem based on strings only, which avoids the difficulties of the traditional number theory approach. The security mechanism for public and secret keys generation is ensured by a recursive encoding mechanism embedded in a quasi-commutative-random function, resulted from the composition of a quasi-commutative function with a pseudo-random function. Due to the inherent construction algorithm of the the proposed cryptosystem, the resulted mathematical inversion problem is likely to be harder than the classical discrete logarithm or integer factorization problems. Furthermore, the proposed cryptosystem is immune against the known quantum algorithm attacks.

1 Introduction

In a symmetrical key cryptosystem [1], such as AES (Advanced Encryption Standard), two users Alice and Bob must first agree on a common secret key. If Alice communicates the secret key to Bob, a third party, Eve, might intercept the key, and decrypt the messages. In order to avoid such a situation Alice and Bob can use an asymmetric public key cryptosystem [1], which provides a secure mechanism to exchange information between two users.

In public key cryptography each user has a pair of cryptographic keys, consisting of a public key and a secret private key. These keys are related through a hard mathematical inversion problem, such that the private key cannot be practically derived from the public key.

The two main directions of public key cryptography are the public key encryption and the digital signatures. Public key encryption is used to ensure confidentiality. In this case, the data encrypted with the public key can only be decrypted with the corresponding private key, and vice versa. Digital signatures are used to ensure authenticity. In this case, any message signed with a user’s private key can be verified by anyone having the user’s public key, proving the authenticity of the message.

A standard implementation of public key cryptography is based on the Diffie-Hellman (DH) key agreement protocol [2]. The protocol allows two users to exchange a secret key over an insecure communication channel. The platform of the DH protocol is the multiplicative group \( \mathbb{Z}_p \) of integers modulo a prime \( p \). The DH protocol can be described as following:
1. Alice and Bob agree upon the public integer $g \in \mathbb{Z}_p$.

2. Alice chooses the secret integer $a$.

3. Alice computes $A = g^a \mod p$, and publishes $A$.

4. Bob chooses the secret integer $b$,

5. Bob computes $B = g^b \mod p$, and publishes $B$.

6. Alice computes the secret integer $K_A = B^a \mod p = g^{ba} \mod p$.

7. Bob computes the secret integer $K_B = A^b \mod p = g^{ab} \mod p$.

It is obvious that both Alice and Bob calculate the same integer $K = K_A = K_B$, which then can be used as a secret shared key for symmetric encryption, improving the performance of the communication channel, since symmetric-key algorithms are generally much faster.

Assuming that the eavesdropper Eve knows $p, g, A$ and $B$, she needs to compute the secret key $K$, that is to solve the discrete logarithm problem:

$$A = g^a \mod p,$$

for the unknown $a$. If $p$ is a very large prime of at least 300 digits, and $a$ and $b$ are at least 100 digits long, then the problem becomes computationally hard (exponential time in $\log p$), and it is considered infeasible. For maximum security $p$ should be a safeprime, i.e. $(p - 1)/2$ is also a prime, and $g$ a primitive root of $p$.

An important short coming of traditional public key cryptosystems is the computation with very large numbers and prime numbers (300 digit), which is difficult and not very efficient. For similar key sizes, the DH method has a similar key strength as the mathematically related RSA method, which is based on the hard integer factorization problem. Currently, no classical algorithm that can factor large numbers efficiently is known, which makes the RSA and the DH methods, widely used cryptographic protocols. However, this problem can be solved on a quantum computer in polynomial time using Shor’s algorithm. Therefore, the current public key cryptographic protocols are insecure if sufficiently large quantum computers become available, putting many sensitive systems like e-commerce and Internet banking at risk. This is why there is an ongoing search for other methods, where similar key exchange mechanisms can be implemented more efficiently, and are immune to quantum algorithms attacks.

Here, we propose a new public key cryptosystem based on strings only, which avoids the traditional number theory approach. The security of the proposed cryptosystem is ensured by a recursive encoding mechanism embedded in a quasi-commutative-random function, which is a composition of a quasi-commutative function with a pseudo-random function. Due to the inherent construction algorithm of the the proposed cryptosystem, the resulted mathematical inversion problem is likely to be harder than the classical discrete logarithm or integer factorization problems. Also, as a direct consequence of the string based approach, attacks based on existing quantum algorithms cannot be used against the proposed cryptosystem.

2 Quasi-commutative transformation

Let us consider the following set:

$$\Omega = \{0, 1, 2, ..., p - 1 | p = 2n, n > 1\}, \quad (2)$$
where the total number of elements is even. For example, if the intention is to build a cryptosystem based on strings where the characters \( \alpha \in \Omega \) are representable on a byte word (8 bits), then one can consider \( p = 2^8 = 256 \). However, we should emphasize that this choice is not a restriction, and the results obtained here are valid for any even integer \( p = 2n \). Thus, we assume that any element:

\[
s = \{s_0, s_1, ..., s_{N-1}\} \in \Omega^N
\]  

(3)

is a string of length \( N \) defined over the set \( \Omega \).

A function \( F : \Omega \times \Omega \rightarrow \Omega \) is said to be quasi-commutative if for any \( \xi, \alpha, \beta \in \Omega \) we have

\[
F(F(\xi, \alpha), \beta) = F(F(\xi, \beta), \alpha).
\]  

(4)

For example, the function used in both DH and RSA algorithms:

\[
F_p(\xi, \alpha) = \xi^\alpha \mod p,
\]  

(5)

is quasi-commutative, because:

\[
F_p(F_p(\xi, \alpha), \beta) = \xi^{\alpha \beta} \mod p = F_p(F_p(\xi, \beta), \alpha).
\]  

(6)

Here we consider the function \( G_w : \Omega \times \Omega \rightarrow \Omega \) defined as:

\[
G_w(\xi, \alpha) = [(w\alpha + 1)\xi + \alpha] \mod p, \quad w \in \mathbb{N}.
\]  

(7)

The function \( G_w \) is quasi-commutative, since we have:

\[
G_w(G_w(\xi, \alpha), \beta) = (w\beta + 1)[(w\alpha + 1)\xi + \alpha] + \beta =
\]

\[
= (w\alpha + 1)(w\beta + 1)\xi + \alpha(w\beta + 1) + \beta =
\]

\[
= (w\alpha + 1)(w\beta + 1)\xi + \beta(w\alpha + 1) + \alpha =
\]

\[
= (w\alpha + 1)[(w\beta + 1)\xi + \beta] + \alpha = G_w(G_w(\xi, \beta), \alpha),
\]

(8)

where we have omitted \( \mod p \), in order to simplify the notation.

The quasi-commutative property ensures that if one starts with an initial value \( \xi \in \Omega \) and a set of values \( \{\alpha_0, \alpha_1, ..., \alpha_{N-1}\} \in \Omega^N \), then the result of the composition:

\[
r = G_w(...(G_w(G_w(\xi, \alpha_0), \alpha_1), ..., \alpha_{N-1}))
\]  

(9)

would not change if the order of the values \( \alpha_n \) were permuted [6].

We should note that if the parameter \( w \) is even, \( w = 2q, q \in \mathbb{N} \), then the function \( G_w \) does not have fixed points. In order to prove this, let us consider the fixed point equation:

\[
G_w(\xi, \alpha) = [(w\alpha + 1)\xi + \alpha] \mod p = \xi,
\]  

(10)

which is equivalent to

\[
(w\alpha + 1)\xi + \alpha = \xi + \alpha mp,
\]  

(11)

and

\[
w\xi = mp - 1.
\]  

(12)

Since \( mp - 1 \) is odd, \( w \) must be also odd in order to have a solution of the above equation. For example, if \( w = 1 \) then \( \xi = (mp - 1) \mod p = p - 1 \) is a solution of the fixed point equation. Similarly, if \( w = 3 \), then \( \xi = p - 3 \) is a solution of the fixed point equation. However, if \( w \) is even, then the fixed point equation has no integer solutions. Thus, the parameter \( w \) in the function \( G_w \) must be even, \( w = 2q, q \in \mathbb{N} \), in order to avoid a fixed point in repeated iterations.
3 Pseudo-random transformation

Let us now consider the following problem. Given a string “seed”:

\[ s^{(0)} = \{s^{(0)}_0, s^{(0)}_1, \ldots, s^{(0)}_{N-1}\} \in \Omega^N, \]  

find a transformation \( R \), that generates a non-repeating and pseudo-random sequence of strings, through a recursive application:

\[ s^{(t+1)} = R(s^{(t)}) = \ldots = R^{(t+1)}(s^{(0)}) = R(R(R(\ldots R(s^{(0)})))) \]  

The transformation \( R \) can be for example a cryptographic hash function, which is considered impossible to invert, a random number generator, or a chaotic deterministic function. Here we prefer to use a secure cryptographic hash function, due to its ability to deal directly with input strings of any length. However, following a similar approach, one can also adapt a cryptographically secure random number generator (for example AES), or a chaotic deterministic function.

An ideal cryptographic hash function has the following properties: (a) it computes a hash value for any given string; (b) it is impossible to invert, i.e. to generate a string that has a given hash; (c) any change in the input string triggers a change in the hash value; (d) it is impossible to find two different strings with the same hash value. Therefore, a cryptographic hash function should provide a random transformation of the input string. Typical hash functions that can be used for the role of the \( R \) transformation are the standard SHA-1, SHA-2 and SHA-3 cryptographic hash functions, published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).

Most cryptographic hash functions are designed to take a string of any length as input and produce a fixed-length hash string value. Here, we assume that the computed output hash string has a fixed length \( L > 0 \), such that:

\[ R : \Omega^N \rightarrow \Omega^L, \forall N \geq 0. \]  

4 The proposed cryptosystem

For any element \( \xi \in \Omega \) and string \( s = \{s_0, s_1, \ldots, s_{N-1}\} \in \Omega^N \) we define the following recursive transformation:

\[ T : \Omega^{N+1} \rightarrow \Omega, \]  

where

\[ T(\xi, s) = G_w(\ldots(G_w(G_w(\xi, s_0), s_1), \ldots), s_N). \]  

According to the results from the previous sections, this transformation satisfies the quasi-commutative property. Also, we should note that from the algorithmic point of view, the transformation \( T \) can be calculated as following:

\[ T(\xi, s) = \begin{cases} \xi & \text{for } n = 0 \ldots N - 1 \\ \xi & \text{return } G_w(\xi, s_n) \end{cases} \]  

Let us now define a more complex transformation \( W \), as a composition of \( T \) and \( R \). For any two strings \( x = \{x_0, x_1, \ldots, x_{K-1}\} \in \Omega^K \) and \( s = \{s_0, s_1, \ldots, s_{N-1}\} \in \Omega^N \) we define:

\[ W : \Omega^{K+N} \rightarrow \Omega^K, \]  


with the components $W_k$, $k = 0, 1, \ldots, K - 1$, given by:

$$W_k = T(x_k, R^{(k+1)}(s)), \quad (20)$$

satisfying the quasi-commutative property, since $T$ is quasi-commutative. The whole transformation is also quasi-commutative, and algorithmically it can be computed as following:

$$W(x, s) = \begin{cases} 
\tilde{s} \leftarrow R(s) \\
\tilde{x}_0 \leftarrow T(x_0, \tilde{s}) \\
\text{for } k = 1 \ldots K - 1 \text{ } \tilde{s} \leftarrow R(\tilde{s}) \\
\tilde{x}_k \leftarrow T(x_k, \tilde{s}) \\
\text{return } \tilde{x}
\end{cases} \quad (21)$$

An important aspect of the $W$ function, is that each component $x_k$ of the input string $x$ is encoded using a different string $\tilde{s}$ obtained by recursively applying the pseudo-random cryptographic hash function $R$ to the secret string $s$.

Since now all the necessary ingredients have been defined, the proposed key generation and exchange protocol can be formulated as following:

1. Alice and Bob agree upon the public string $g = \{g_0, g_1, \ldots, g_{K-1}\} \in \Omega^K$.
2. Alice chooses the secret string $a = \{a_0, a_1, \ldots, a_N\} \in \Omega^N$.
3. Alice computes the string $A = W(g, a) \in \Omega^K$, and publishes $A$.
4. Bob chooses the secret string $b = \{b_0, b_1, \ldots, b_M\} \in \Omega^M$.
5. Bob computes the string $B = W(g, b) \in \Omega^K$, and publishes $B$.
6. Alice calculates the secret string $s_A = W(B, a) = W(W(g, b), a) \in \Omega^K$.
7. Bob calculates the secret string $s_B = W(A, b) = W(W(g, a), b) \in \Omega^K$.

Both Alice and Bob obtain the same secret key $s = s_A = s_B$, since $W$ satisfies the quasi-commutativity property:

$$W(W(g, b), a) = W(W(g, a), b). \quad (22)$$

The flow chart showing the above key generation and exchange protocol is given in Figure 1. Also, the C code implementing the proposed key generation and exchange protocol, and a typical key exchange simulation example, are given in the Appendix.

5 The hard problem

Assuming that the eavesdropper Eve knows $g$, $A$ and $B$ the hard problem is to compute the strings $a \in \Omega^N$ and respectively $b \in \Omega^M$. This is a hard problem, since even the length ($N$ and $M$) of the strings is kept secret by Alice and Bob, and the security is ensured by the above described recursive encoding mechanism embedded in the quasi-commutative-random function $W$, which is a composition of a quasi-commutative function $T$ with a pseudo-random function $R$.

The encoding mechanism can also be described with the following recursive equations for the evolution of $a$ and $A_k$, $k = 0, 1, \ldots, K - 1$:

$$a^{(k)} = R^{(k+1)}(a) = R(R(\ldots R(a))_{k+1}), \quad (23)$$
and respectively

\[ A_k = [g_k \prod_{n=0}^{L-1} (w a_n^{(k)} + 1) + \sum_{i=0}^{L-2} a_i^{(k)} \prod_{n=i+1}^{L-1} (w a_n^{(k)} + 1) + a_{N_k-1}^{(k)}] \mod p, \]  

(24)

with the unknowns \( a_0^{(k)}, a_1^{(k)}, ..., a_{L-1}^{(k)} \). We should note that for each \( k = 0, 1, ..., K - 1 \), any permutation \( \tilde{a}^{(k)} \) of \( a^{(k)} \) is also a solution of the equation (24). Assuming that by chance Eve finds a solution \( \tilde{a}^{(0)} \) for the first equation, \( k = 0 \), which is in fact a permutation of a hash of \( a \), then one may think that she may use the recursion equation (23) to find \( \tilde{a}^{(k)} \). This is not possible either, because the results \( \tilde{a}^{(k)} \) obtained via \( R^{(k+1)} \) are completely dependent on the order of the characters in the string \( \tilde{a}^{(0)} \). Therefore, any permutation \( \tilde{a}^{(0)} \) of \( a^{(0)} \) will give a different result \( \tilde{a}^{(k)} \). The recursion \( R^{(k+1)} \) gives the correct result if and only if \( \tilde{a}^{(0)} \equiv a^{(0)} \). For these reasons the proposed cryptosystem hinders eavesdroppers from discovering the secret string keys \( a \) and respectively \( b \).

6 Conclusion

In conclusion, we have presented a new public key cryptosystem based on strings, which avoids the cumbersome key generation using the traditional number theory approach. The security mechanism for public and secret keys generation is ensured by a recursive encoding mechanism embedded in a quasi-commutative-random function, resulted from the composition of a quasi-commutative function with a pseudo-random function. The public and secret keys can be easily created and the encoding-decoding process can be performed very rapidly, with low memory requirements and computational power. Due to the inherent construction algorithm of the the proposed cryptosystem, the resulted mathematical inversion problem is likely to be harder than the classical discrete logarithm or integer factorization problems. Also, due to the string-based approach, attacks based on known quantum algorithms cannot be used against the proposed cryptosystem.

References

[1] Douglas R. Stinson, Cryptography Theory and Practice, third edition, CRC Press, Boca Raton, Florida, 2006.

[2] W. Diffie, M. Hellman, New directions in cryptography, IEEE Transactions on Information Theory 22 (6): 644–654 (1976).

[3] Randall K. Nichols, ICSA guide to Cryptography, McGraw-Hill, New York, 1999.

[4] R. Rivest, A. Shamir, L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21 (2): 120–126 (1978).

[5] P. W. Shor, Algorithms for quantum computation: Discrete logarithms and factoring, Proc. 35nd Annual Symposium on Foundations of Computer Science (Shafi Goldwasser, ed.), IEEE Computer Society Press, 124-134 (1994).

[6] J. Benaloh, M. de Mare, One-way accumulators: a decentralized alternative to digital signatures, Advances in Cryptology-Eurocrypt’93, LNCS, vol. 765, Springer-Verlag, 274-285, 1993.
Figure 1: The flow chart of the key generation and exchange protocol.

Alice and Bob agree upon the public string $g$

Alice chooses the secret string $a$

Bob chooses the secret string $b$

Alice computes the public string $A = W(g, a)$ and publishes $A$

Bob computes the public string $B = W(g, b)$ and publishes $B$

Alice computes the secret string $s_A = W(B, a)$

Bob computes the secret string $s_B = W(A, b)$

$s_A = W(B, a) = W(W(g, b), a) = W(W(g, a), b) = W(A, b) = s_B$

Figure 1: The flow chart of the key generation and exchange protocol.
Appendix

Below we give the C code of the proposed key exchange protocol. The code is self-explanatory and follows the notation from the text.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <openssl/sha.h>

unsigned char *W(unsigned char *x, unsigned K, unsigned char *s, unsigned N)
{
    unsigned k, n, w = 2;
    unsigned char *y = malloc(K);

    unsigned char *z = SHA512(s, N, 0);
y[0] = x[0];
    for(n=0; n<SHA512_DIGEST_LENGTH; n++){
        y[0] = (w*z[n] + 1)*y[0] + z[n];
    }
    for(k=1; k<K; k++){
        z = SHA512(z, SHA512_DIGEST_LENGTH, 0);
y[k] = x[k];
        for(n=0; n<SHA512_DIGEST_LENGTH; n++){
            y[k] = (w*z[n] + 1)*y[k] + z[n];
        }
    }
    return y;
}

void sprint(unsigned char *x, unsigned N){
    unsigned n;
    for(n=0; n<N; n++){
        if(n % 32 == 0) printf("\n");
            else printf("%02 x", x[n]);
    }
    printf("\n");
}

main(int argc, char *argv[]){
    unsigned N = 128, M = 128, K = 128;
    unsigned n, m, k;
srand((unsigned char) time(0) + getpid());

    unsigned char *g = malloc(K);
    for(k=0; k<K; k++) g[k] = rand() % 256;
    printf("\nGenerator: public g "); sprint(g, K);

    unsigned char *a = malloc(N);
    for(n=0; n<N; n++) a[n] = rand() % 256;
    printf("\nAlice: secret key a" ); sprint(a, N);
}```
unsigned char *b = malloc(M);
for(m=0; m<M; m++) b[m] = rand() % 256;
printf("\nBob: secret key b"); sprint(b, M);

unsigned char *A = W(g, K, a, N);
printf("\nAlice: public key A"); sprint(A, K);

unsigned char *B = W(g, K, b, M);
printf("\nBob: public key B"); sprint(B, K);

unsigned char *Sa = W(B, K, a, N);
printf("\nAlice: secret key Sa"); sprint(Sa, K);

unsigned char *Sb = W(A, K, b, M);
printf("\nBob: secret key Sb"); sprint(Sb, K);

free(a); free(b); free(g); free(A);
free(B); free(Sa); free(Sb);
return 0;
}

The code example (keys.c) has been tested on a LINUX PC, running Ubuntu 14.04 LTS, using the standard GCC compiler and OpenSSL implementation of the Secure Socket Layer (SSL), and related cryptographic tools. In this example, the role of the secure cryptographic hash function \( R \) is taken by the SHA-2 function which provides hash strings with a length of 64 bytes (512 bits), and we set \( p = 256 \).

The required compilation and run steps are:

gcc -lssl -lcrypto keys.c -o keys
./keys

For illustration purposes, the results obtained for one instance run are given below in hexadecimal. One can see that both Alice and Bob compute the same secret key \((Sa = Sb)\), using completely different secret and public keys. In this example Alice and Bob secret strings, \(a\) and \(b\), have a length of \(N = M = 128\) bytes (1Kb); also the generator string \(g\), the public keys \(A\) and \(B\), and the shared secret key, all have a length \(K = 128\) bytes (1Kb). In practice, one can use different key lengths \(N \neq M \neq K\) if necessary.

Simulation results:

Generator: public g
a4b347edde38ce5f524b9801373045d6c36bce0a2a26c3c0837dd55d4b3e58e
195545770e14ed3246e46c0f2a3ac29e00ecc037a090b31e6df061ac51571
6ab565784a442b97b271d82594054ff4131cf88e404bf0b6446625a95b1609
cc5fbe9623e9e567bee8ccae41f66ddad8a61ed95c5b33a700f50f854

Alice: secret key a
d79233faf6b1506684bc67416ab93fb09c1866aaf1cb06356a0f1269946c2
59f556b55d73bac408773b49de489aa70a09139bd421c93e20eabfbd47dec
74d7034c14af0c9b23c0b84188d2681374fa4cb1bce845a0f67f1c4b7d89f8
61fcdb75ac49115089ca91129dfaa49175f143325a8952d0886f1d06781677
Bob: secret key b
13b45040fde090872a11a491cbeda91301f448a28965c30857937fd0fae58
6428e2e109f26a359484f8db143d8437477077ef1f9eda5a3d3072263b57a06
22687c2b5b66def6ad21ad37e4a12eeb21cd8a7b1eae50a450b35aca3974
a170977c56f7ebc0550ed8cf2103bde3109e92ca698fb4be82f253368199f

Alice: public key A
be5af5a6051be16a8d1ba161b6b361b76d0e5226f74d17b0536e03bc8919cb
cb67e04c0a064bc2da06e3582b954fc00440b2c706e3a3b45022b7b964a0
b6d517ec83a7f7e1205b0fdccac7900b64097b7211a36de2f41e49447993861
df2920b881641c89f92c606303180f004560a78ce8666e1971c7ede787829d

Bob: public key B
49616d7f977835d5ebe5ad712b0a78a4a7971b53bc0457829042f9d040e3b7
cb8eb2a31df4b846e3d2420554e98453523f4ef66613d9489c66a9e47ef58
f41981071e4b38acfe748217d44a4396d1fe6c2e7e0ef48681f9d151559f469e
e14c001aade0cde2c26542e909700259fd2dc9a180c96012c0f49d27be85

Alice: secret key Sa
c764f38474db9a39a9a2d768379e4d843fe49519bf8595744ab4717778faa786
c8e570746fa3276c1e0206f2111d21b251121a63aebe5eafbc900c3abe20d
70da30931f480c12c1b616a4824759675dc7c1317ce673f508538636d28c6
c2128a8cf359afed510fd1a42646a1463650d38f6daabafe5276c9d5804070

Bob: secret key Sb
c764f38474db9a39a9a2d768379e4d843fe49519bf8595744ab4717778faa786
c8e570746fa3276c1e0206f2111d21b251121a63aebe5eafbc900c3abe20d
70da30931f480c12c1b616a4824759675dc7c1317ce673f508538636d28c6
c2128a8cf359afed510fd1a42646a1463650d38f6daabafe5276c9d5804070