On a new fast public key cryptosystem

Samir Bouftass

E-mail: crypticator@gmail.com.

September 1, 2015

Abstract

This paper presents a new fast public key cryptosystem namely: a key exchange algorithm, a public key encryption algorithm and a digital signature algorithm, based on the difficulty to invert the following function: \( F(x) = (a \times x) \mod (2^p) \div (2^q) \).

\( \mod \) is modulo operation, \( \div \) is integer division operation, \( a, p \) and \( q \) are integers where \( (p > q) \).

In this paper we also evaluate the hardness of this problem by reducing it to SAT.

Keywords: key exchange, public key encryption, digital signature, boolean satisfiability problem, multivariate polynomials over \( F(2) \).

1 Introduction:

Since its invention by Withfield Diffie and Martin Hellman [1], Public key cryptography has imposed itself as the necessary and indispensable building block of every IT Security architecture. But in the last decades it has been proven that public key cryptosystems based on number theory problems are not immune against quantum computing attacks [3]. The advent of low computing resources mobile devices such wireless rfid sensors, smartphones, etc has also put demands on very fast and lightweight public key algorithms.

Public key cryptosystem presented in this paper is not based on number theory problems and is very fast compared to Diffie-Hellman [1] and RSA algorithms [2]. It is based on the difficulty to invert the following function: \( F(x) = (a \times x) \mod (2^p) \div (2^q) \).

\( \mod \) is modulo operation, \( \div \) is Integer division operation, \( a, p \) and \( q \) are known integers where \( (p > q) \). In this paper we construct three public key algorithms based on this problem namely a key exchange algorithm, a public key encryption algorithm and a digital signature algorithm.

We prove its efficiency compared to Diffie-Hellman and RSA, and that the underlying problem can be a hard SAT instance [4] or a equations set of multivariate polynomials over \( F(2) \).
2 Secret key exchange algorithm:

Before exchanging a secret key, Alice and Bob shared a knowledge of:

Integers \([ l, m, p, q, r, Z ]\) satisfying following conditions:

\[
q = l + m - p \ , \ p > m + q + r, \ Z \text{ is } l \text{ bits long.}
\]

To exchange a secret key:

- 1 Bob chooses randomly a integer \(X\). \([X]\) is \(m\) bits and a private knowledge of Bob.
- 2 Computes number \(U = (X \times Z)\text{Mod}(2^p)\text{Div}(2^q)\), and sends it to Alice.
- 3 Alice chooses randomly a integer \(Y\). \([Y]\) is \(m\) bits and a private knowledge of Alice.
- 4 Computes number \(V = (Y \times Z)\text{Mod}(2^p)\text{Div}(2^q)\), and sends it to Bob.
- 5 Bob computes number \(W_a = (X \times V)\text{Mod}(2^{p-q})\text{Div}(2^{m+r})\).
- 6 Alice computes number \(W_b = (Y \times U)\text{Mod}(2^{p-q})\text{Div}(2^{m+r})\).

Our experiments shows us that \(Pr[W_a = W_b] = 1 - 0.3 \times 2^{-r}\).

If Bob and Alice chooses \(r\) great enough say superior to 128, \(Pr[W_a \neq W_b]\) will be negligible, they can use then this protocol as a key exchange algorithm.

Secret key exchanged is the number:

\[
W = (X \times V)\text{Mod}(2^{p-q})\text{Div}(2^{m+r}) = (Y \times U)\text{Mod}(2^{p-q})\text{Div}(2^{m+r})
\]

A python implementation of this algorithm is provided in Appendix A.
3 Public key encryption algorithm :

3.1 Encryption :

In order to send a encrypted message to Bob, Alice performs the following steps:

- 1 She gets his public key composed by integers \([ l, m, p, q, r, Z, U ]\), satisfying:

  \[ q = l + m - p, \quad p > m + q + r, \quad U = (X \times Z) Mod(2^p) Div(2^q), \]  
  \[ Z \text{ is } l \text{ bits long}. \]

- 2 She chooses randomly a integer \( Y \) which is \( m \) bits long.

- 3 She computes number \( V = (Y \times Z) Mod(2^p) Div(2^q) \), then the secret key

  \[ W = (Y \times U) Mod(2^{p-q}) Div(2^{m+r}). \]

- 4 She encrypts with secret key \( W \) her plaintext and sends corresponding ciphertext

  and number \( V \) to Bob.

3.2 Decryption :

In order to decrypt the ciphertext received from Alice, Bob performs the following steps:

- 1 With his private key \( X \) and number \( V \) received from Alice,

  he computes secret key

  \[ W = (X \times V) Mod(2^{p-q}) Div(2^{m+r}). \]

- 2 With secret key \( W \), he decrypts the ciphertext received from Alice.
4 Digital signature Algorithm:

Bob’s public key is composed by integers \([ l, m, p, q, r, Z, U ]\), satisfying:

\[ q = l + m - p, \quad p > l + q + r, \quad U = (X \times Z)Mod(2^p)Div(2^q), \quad X \text{ is } m \text{ bits whereas } Z \text{ is } l \text{ bits long}. \]

4.1 Signature:

In order to sign a Message \(Msg\), Bob performs the following steps:

- 1 He chooses randomly an \(m\) bit long integer \(Y\).

- 2 Hashes \(Msg\) by a hash function \(HF\) and gets a digest \(H\) which length in bits is the same as elements \(Z\) of his public key. With his private key \([X]\), he computes:

\[ S1 = (Y \times Z)Mod(2^p)Div(2^q) \text{ and } S2 = (H \times (X + Y))Mod(2^p)Div(2^q) \]

- 3 Sends Message \(Msg\) and signature \((S1, S2)\) to Alice.

4.2 Verification:

In order to verify that Message \(Msg\) is sent by Bob, Alice performs the following steps:

- 1 She gets his public key.

- 2 Hashes \(Msg\) by \(HF\) and gets a digest \(H\) which the length in bits is \(l\) the same as \(Zs\).

- 3 From digest \(H\), signature \((S1, S2)\) and the elements \([p, q, l, U, Z]\) of Bob’s public key, she computes \(W_a = (H \times (S1 + U))Mod(2^{p-q})Div(2^{l+r})\) and \(W_b = (Z \times S2)Mod(2^{p-q})Div(2^{l+r})\).

- 4 Compares \(W_a\) to \(W_b\), \(Msg\) is sent by Bob if \(W_a = Wb\)

A python implementation of this algorithm is provided in Appendix B.
5 Efficiency:

The key exchange algorithm presented in this paper can be realised by a multiplication circuit where some leftmost and rightmost output bits are discarded, meaning that it has a time complexity of $O(n^2)$. In comparison to standardised key exchange algorithms such as Diffie-Hellman in the multiplicative group or RSA whose time complexities are $O(n^3)$, under the same security parameters, presented key exchange algorithm is $O(n)$ time faster. The same can be said about presented public key and digital signature algorithms since they are basically applications of the key exchange algorithm.

6 Security:

The Security of presented public key cryptosystem is based on the difficulty of finding $X$ and $Y$ while knowing $l, m, p, q, r, U = (X \times Z) Mod(2^p) Div(2^q), V = (Y \times Z) Mod(2^p) Div(2^q), l + m = p + q, p > m + q + r, Z$ is $l$ bit long, $X$ and $Y$ are $m$ bits.

To get $X$ from $U$ and $Y$ from $V$, a attacker should:

1. Invert $F(X) = U = (Z \times X) Mod(2^p) Div(2^q)$
2. Invert $F(Y) = V = (Z \times Y) Mod(2^p) Div(2^q)$.

Putting it otherwise, presented public key cryptosystem is based on the difficulty to invert the following function:

$$F(x) = y = (a \times x) Mod(2^p) Div(2^q).$$

$a, x, p$ and $q$ are known integers, while $a$ and $x$ are respectively $n$ and $m$ bits long, $(n > m)$ and $(p > q)$.

At first glance we can notice that it is easy to verify a solution but it is difficult to find one, implying that this problem is in NP.

In our knowledge it has never been mentioned in the literature. In subsequent section we will reduce it to SAT in order to evaluate its hardness.
6.1 Hardness evaluation:

Let $A$, $X$, $Y$, $n$, $m$ be integers where $A$, $X$ are respectively $n$ and $m$ bits long ($m \leq n$).

The binary representation of $A$ is $a_{(n-1)}...a_{(i+1)}a_i...a_0$.

The binary representation of $X$ is $x_{(m-1)}...x_{(i+1)}x_i...x_0$.

The binary representation of $Y$ is $y_{(n+m-1)}...y_{(i+1)}y_i...y_0$.

$Y$ is the arithmetic product of $A$ and $X$, $Y$’s bits in function of $A$’s bits and $X$’s bits can be translated by the following set of algebraic equations (1):

$c_0 = 0$

For ($j = 0$ to $m-1$):\[ y_j = (\sum_{i=0}^{j} a_{(j-i)} \times x_i) + c_j \mod(2) \quad \text{and} \quad c_{j+1} = (\sum_{i=0}^{j-1} a_{(j-i)} \times x_i) + c_j \div(2) \]

For ($j = m - 1$ to $n - 1$):\[ y_j = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \mod(2) \quad \text{and} \quad c_{j+1} = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \div(2) \]

For ($j = n - 1$ to $m + n - 1$):\[ y_j = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \mod(2) \quad \text{and} \quad c_{j+1} = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \div(2) \]

$c_j$ is the retenue bit of multiplication product ($Y = A \times X$) at column $j$.

In our problem we have as unknowns bits $Y_{0\rightarrow q}$ and $Y_{p\rightarrow(m+n-1)}$ (1) become then following set of algebraic equations (2):

For ($j = q$ to $m-1$):\[ y_j = (\sum_{i=0}^{j} a_{(j-i)} \times x_i) + c_j \mod(2) \quad \text{and} \quad c_{j+1} = (\sum_{i=0}^{j-1} a_{(j-i)} \times x_i) + c_j \div(2) \]

For ($j = m - 1$ to $n - 1$):\[ y_j = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \mod(2) \quad \text{and} \quad c_{j+1} = (\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j \div(2) \]
For \( j = n - 1 \) to \( p \):

\[
y_j = \left( \sum_{i=j-n+1}^{m-1} a_{j-i} \times x_i \right) + c_j \mod 2
\]

and

\[
c_{j+1} = \left( \sum_{i=j-n}^{m-1} a_{j-i} \times x_i \right) + c_j \div 2
\]

Set of algebraic equations (2) can be translated to following set of logical equations (3).

If \( j \leq m \)

\[
c_j = F_j(x_{j-1}, \ldots, x_{(k+1)}, x_k, \ldots, x_0, c_{j-1})
\]

If \( m \leq j \)

\[
c_j = F_j(x_m, \ldots, x_{(k+1)}, x_k, \ldots, x_0, c_{j-1})
\]

\[
\land_{j=0}^{m} ((\oplus_{i=0}^{j}(a_{j-i} \land x_i) \oplus c_j) = y_j) = true
\]

\[
\land_{j=m}^{n} ((\oplus_{i=m-j}^{j}(a_{j-i} \land x_i) \oplus c_j) = y_j) = true
\]

\[
\land_{j=n}^{p} ((\oplus_{i=n-j}^{j}(a_{j-i} \land x_i) \oplus c_j) = y_j) = true
\]

Notice this set of logical equation is practically the same set resulting from reducing FACT to SAT.

In paper [9] Authors had suggested to use FACT as a source of Hard SAT Instances, moreover SAT Solvers to this day are still inefficient in solving this sort of SAT instances.

Every logical function can be realised by nands gates, if we replace \( \neg x_i \) by \( (1 - x_i) \), \( (x_i \land x_j) \) by \( (x_i \times x_j) \) (3) can also be translated to following set of multivariate polynomials equations:

If \( q \leq j \leq m - 1 \):

\[
y_j = (\sum_{i=0}^{j} a_{j-i} \times x_i) + F_j(x_0, \ldots, x_m) \mod 2
\]

If \( m - 1 \leq j \leq n - 1 \):

\[
y_j = (\sum_{i=0}^{m-1} a_{j-i} \times x_i) + F_j(x_0, \ldots, x_m) \mod 2
\]

If \( n - 1 \leq j \leq p \):

\[
y_j = (\sum_{i=j-n+1}^{m-1} a_{j-i} \times x_i) + F_j(x_0, \ldots, x_m) \mod 2
\]

Where \( F_j \)'s are multivariate polynomials corresponding to carries \( c_j \)

Summing it up, to break presented public key cryptosystem one had to solve SAT instances resulting from logical equations containing lot of Xors which is not that evident if the number of unknown bits are high enough [6].

Or solve sets of multivariate polynomials equations over \( F(2) \) with degrees superior or equal to parameter \( q \).
7 Conclusion, open question and future work:

In this paper we have presented a new fast public key cryptosystem based on the difficulty of inverting the following function: $F(x) = (a \times x) \text{Mod}(2^p) \text{Div}(2^q)$.

$\text{Mod}$ is modulo operation, $\text{Div}$ is integer division operation, $a$, $p$ and $q$ are known integers where $(p > q)$.

We have proved its efficiency compared to Diffie Hellman and RSA cryptosystems. We have also proved that its security is based on a new problem that can be viewed as a hard sat instance or a set of multivariate polynomial equations over $F(2)$.

The fact that its security is not based on number theory problems is also a proof of its resistance against current quantum computing attacks [3].

The last decade have seen an enormous progress of SAT Solvers but they are still inefficient in solving logical statements containing a lot of xors which is the case of our problem [5][6][7].

SAT is NP complete, meaning that solving it can take exponential time. It is has been found that the hardest instances of a SAT problem depends on its constraintness which is defined as the ratio of clauses to variables [8].

This lead us to ask what forms should have the integers composing public parameters of our PKCS in order to produce hard SAT instances even to a eventual SAT Solver that have not problems with xor clauses.

Recently we have found a way to build public key cryptosystems based on the difficulty of inverting the following function: $F(x) = (a \times x) \text{Mod}(b^p) \text{Div}(b^q)$.

$\text{Mod}$ is modulo operation, $\text{Div}$ is integer division operation, $a$, $b$, $p$ and $q$ are known integers where $(p > q)$.

This work will be the subject of a future paper.
References


8 Appendix A :

Following python script is a ” practical ” proof of correctness of key exchange algorithm presented in this paper. (pycrypto library is needed )

```
import sys
from Crypto.Util.number import getRandomNBitInteger

def ModDiv(A,B,C) :
    return (A % B ) // C

l = sys.argv[1]
m = sys.argv[2]
p = sys.argv[3]
q = sys.argv[4]
r = sys.argv[5]

try:
    l = int(l)
m = int(m)
p = int(p)
q = int(q)
r = int(r)

except ValueError:
    print(‘Invalid Arguments’)

if m + q + r ≥ p :
    print(“”)
    print(“Public Parameter l = %d” %l)
    print(“Public Parameter m = %d” %m)
    print(“Public Parameter p = %d” %p)
    print(“Public Parameter q = %d” %q)
    print(“Public Parameter r = %d” %r)
    print(“”)
    print(‘Condition (p > m + q + r) is not fulfilled !’)
else :
    ” Size in bits of public pararameter Z is l ”
```
Z = getRandomNBitInteger(l,randfunc=None)

" Size in bits of private parameters X and Y is m "
X = getRandomNBitInteger(m,randfunc=None)
Y = getRandomNBitInteger(m,randfunc=None)

M = pow(2,p)
M1 = pow(2,p-q)
D = pow(2,q)
D1 = pow(2,m+r)

" If r = 0, In 30 % percents, the keys computed by Alice and Bob are not identical : ",
" W_a = W_b ± 1, this is due to bit carry propagation, if r is increased by one ",
" the probability that W_a is different to W_b is divided by two. "

U = ModDiv(Z*X,M,D)
V = ModDiv(Z*Y,M,D)
Wa = ModDiv(U*Y,M1,D1)
Wb = ModDiv(V*X,M1,D1)

print("")
print("Public Parameters :")
print("="*30)
print("")

print("Public Parameter l = %d" %l)
print("Public Parameter m = %d" %m)
print("Public Parameter p = %d" %p)
print("Public Parameter q = %d" %q)
print("Public Parameter r = %d" %r)
print("Public Parameter Z = %d" %Z)
print("")

print("Private Parameters :")
print("="*30)
print("")

print("Alice Private Parameter X = %d" %X)
print("Bob Private Parameter Y = %d" %Y)
print(""")
print("Shared Parameters :")
print("=================")
print(
"Parameter shared with Bob by Alice U = %d" %U)
print("Parameter shared with Alice by Bob V = %d" %V)

print("\n")

print("Exchanged Secret Key :")
print("======================")
print("\n")

print("Secret key computed by Alice Wa = %d" %Wa)
print("Secret key computed by Bob Wb = %d" %Wb)

print("\n")
sys.exit

=================================================================================================

You can download this script from: https://github.com/Crypticator/ModDiv/blob/master/Kex.py

9 Appendix B:

Following python script is a "practical" proof of correctness of digital signature algorithm presented in this paper. (pycrypto library is needed)

=================================================================================================

import sys
from Crypto.Util.number import getRandomNBitInteger

def ModDiv(A,B,C):
    return (A % B) // C

l = sys.argv[1] // C
m = sys.argv[2]
p = sys.argv[3]
q = sys.argv[4]
r = sys.argv[5]
try:
    l = int(l)
    m = int(m)
    p = int(p)
    q = int(q)
    r = int(r)

except ValueError:
    print('Invalid Arguments')

if m + q + r ≥ p:
    print('')
    print("Public Parameter l = %d" %l)
    print("Public Parameter m = %d" %m)
    print("Public Parameter p = %d" %p)
    print("Public Parameter q = %d" %q)
    print("Public Parameter r = %d" %r)
    print('')
    print('Condition (p > m + q + r) is not fulfilled !')

else:
    " Size in bits of public parameter Z is l ")
    Z = getRandomNBitInteger(l, randfunc=None)
    " a hash value of a hypothetical file ")
    H = getRandomNBitInteger(l, randfunc=None)
    " Size in bits of private parameters X and Y is m ")
    X = getRandomNBitInteger(m, randfunc=None)
    Y = getRandomNBitInteger(m, randfunc=None)

    M = pow(2, p)
    M1 = pow(2, p-q)
    D = pow(2, q)
    D1 = pow(2, l+r)

    U = ModDiv(Z*X, M, D)
    S1 = ModDiv(Z*Y, M, D)
    S2 = ModDiv(H*(X+Y), M, D)

    Wa = ModDiv(H*(S1+U), M1, D1)
    Wb = ModDiv(Z*S2, M1, D1)
Public Parameters :
===================

Public Parameter l = %d  %l
Public Parameter m = %d  %m
Public Parameter p = %d  %p
Public Parameter q = %d  %q
Public Parameter r = %d  %r
Public Parameter Z = %d  %Z
Public Parameter H = %d  %H

Private Parameters :
====================

Private key X = %d  %X
Ephemeral key Y = %d  %Y

Signature:
==========

S1 = %d  %S1
S2 = %d  %S2

Verification :
==============

Wa = %d  %Wa
Wb = %d  %Wb

sys.exit

You can download this script from: https://github.com/Crypticator/ModDiv/blob/master/moddiv/Sig.py