On a new fast public key cryptosystem

Samir Bouftass

E-mail : crypticator@gmail.com.

July 20, 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 satisfability 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 againt quantum computing attacks [3]. The advent of low computing ressources mobile devices such wirless rfid sensors, smart cellphones, ect 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 is l 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 $U = (X \times Z)Mod(2^p)Div(2^q)$, 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 $V = (Y \times Z)Mod(2^p)Div(2^q)$, and sends it to Bob.

- 5 Bob computes number $W_a = (X \times V)Mod(2^{p-q})Div(2^{m+r})$.

- 6 Alice computes number $W_b = (Y \times U)Mod(2^{p-q})Div(2^{m+r})$.

Our experiments shows us that $Pr[W_a = W_b] = 1 - 0.3 * 2^{-r}$.

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

Secrete key exchanged is the number :

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

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 gots his public key composed by integers [l, m, p, q, r, Z, U] , satisfying :

$q = l + m - p$, $p > m + q + r$, $U = (X \times Z)Mod(2^p)Div(2^q)$, Z is l bits long.

[X] is the m bits long private key of Bob.

- 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 recieved from Alice, Bob performs the following steps :

- 1 With his private key X and number V recieved 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 recieved from Alice.

4 Digital signature Algorithm :

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

$q = l + m - p$, $p > l + q + r$, $U = (X \times Z)Mod(2^p)Div(2^q)$, X is m bits whereas Z is l bits long.

4.1 Signature :

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

- 1 He chooses randomely 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 :

  $S1 = (Y \times Z)Mod(2^p)Div(2^q)$ 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 gots 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 $Wa = (H \times (S1+U))Mod(2^{p-q})Div(2^{l+r})$ and $Wb = (Z \times S2)Mod(2^{p-q})Div(2^{l+r})$.

- 4 Compares Wa to Wb , Msg is sent by Bob if $Wa = Wb$

A python implementation of this algorithm is provided in Appendix B

5 Efficiency :

In comparaison to the standardised key exchange algorithms such as Diffie-Hellman in the multiplicatif group and RSA which needed in average N multiplications modulo operations to exchange a secret key (N being private key's lenght).
The key exchange algorithm presented in this paper needed just 4 multiplications ,
Meaning that presented public key cryptosystem is very fast and efficient compared to Diffie-Hellman and RSA cryptosystems.

6 Security :

The Security of presented public key cryptosystem is based on the difficulty of finding X and Y while knowing Z, 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, an attacker should :

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

Puting 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.

To our knowlege 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 rescpectively 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 algebric 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* $n-1$) :

$$y_j = ((\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j) Mod(2) \quad \text{and} \quad c_{j+1} = ((\sum_{i=0}^{m-1} a_{(j-1-i)} \times x_i) + c_j) Div(2)$$

For ($j = n-1$ *to* $m+n-1$) :

$$y_j = ((\sum_{i=j-n+1}^{m-1} a_{(j-i)} \times x_i) + c_j) Mod(2) \quad \text{and} \quad c_{j+1} = ((\sum_{i=j-n}^{m-1} a_{(j-i)} \times x_i) + c_j) Div(2)$$

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

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

$$y_j = ((\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + c_j) Mod(2) \quad \text{and} \quad c_{j+1} = ((\sum_{i=0}^{m-1} a_{(j-1-i)} \times x_i) + c_j) Div(2)$$

For ($j = n-1$ *to* p) :

$$y_j = ((\sum_{i=j-n+1}^{m-1} a_{(j-i)} \times x_i) + c_j) Mod(2) \quad \text{and} \quad c_{j+1} = ((\sum_{i=j-n}^{m-1} a_{(j-i)} \times x_i) + c_j) Div(2)$$

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

If $(j \leq m)$ $c_j = F_j(x_{(j-1)}, ..., x_{(k+1)}, x_{(k)}, ..., x_0, c_{(j-1)})$

If $(m \leq j)$ $c_j = F_j(x_m, ..., x_{(k+1)}, x_{(k)}, ..., x_0, c_{(j-1)})$

$\wedge_{j=q}^{m}((\oplus_{i=0}^{j}(a_{(j-i)} \wedge x_i) \oplus c_j) = y_j) = true$

$\wedge_{j=m}^{n}((\oplus_{i=j-m}^{j}(a_{(j-i)} \wedge x_i) \oplus c_j) = y_j) = true$

$\wedge_{j=n}^{p}((\oplus_{i=j-n}^{n}(a_{(j-i)} \wedge x_i) \oplus c_j) = y_j) = true$

Notice this set of logical equation is practicaly 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 at solving this sort of SAT instances.

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

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

$$y_j = ((\sum_{i=0}^{j} a_{(j-i)} \times x_i) + F_j(x_0, ...x_m)) Mod(2)$$

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

$$y_j = ((\sum_{i=0}^{m-1} a_{(j-i)} \times x_i) + F_j(x_0, ...x_m)) Mod(2)$$

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

$$y_j = ((\sum_{i=j-n+1}^{m-1} a_{(j-i)} \times x_i) + F_j(x_0, ...x_m)) Mod(2)$$

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) whith 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 : $F(x) = (a \times x)Mod(2^p)Div(2^q)$.
Mod is modulo operation , Div is integer division operation , a , p and q are known integers where $(p > q)$.

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 a enormous progress of SAT Solvers but they are still inefficient in solving logical statements containg 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 constraindness 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 : $F(x) = (a \times x)Mod(b^p)Div(b^q)$.
Mod is modulo operation , 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

[1] Whitfield Diffie, Martin E.Hellman. *New Directions in cryptography, IEEE Trans. on Info. Theory, Vol. IT-22, Nov. 1976 (1976)*

[2] R.L. Rivest , A. Shamir , L. Adleman. *A method of obtaining digital signatures and public key cryptosystems , SL Graham, RL Rivest* Editors (1978)*

[3] Daniel J Bernstein, Johannes Buchmann, Erik Dahman. *Post-Quantum Cryptography*, (2009), Springer Verlag , Berlin Heidelberg .

[4] Thomas J Schaefer. *The complexity of satisfability problems* , (1978), Departement of Mathematics University of California, Berkeley .

[5] Stefan Schoenmackers, Anna Cavender. *Satisfy This: An Attempt at Solving Prime Factorization using Satisfiability Solvers* , (2005), Department of Computer Science, University of Washington .

[6] Mate Soosy, Karsten Nohlz, and Claude Castellucciay. *Extending SAT Solvers to Cryptographic Problems*, (2009), INRIA Rhone-Alpes, University of Virginia .

[7] Tero Laitinen, Tommi Junttila, Ilkka Niemel*Conflict-Driven XOR-Clause Learning* , (2014), Logic in Computer Science .

[8] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, Yoav Shoham *Understanding Random SAT: Beyond the Clauses-to-Variables Ratio* , (1978), Principles and Practice of Constraint Programming CP 2004 Lecture Notes in Computer Science Volume 3258, 2004 .

[9] Satoshi Horie, Osamu Watanabe *Hard instance generation for SAT* , Algorithm and Computation, Lecture Notes in Computer Science Volume 1350,1997, pp 22-31 , Date 29 Jul 2005

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 : "
" Wa = Wb ± 1, this is due to bit carry propagation, if r is increased by one "
" the probability that Wa is diffrent to Wb is devided 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("====================")
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("====================")
print("")

print("Alice Private Parameter X = %d" %X)
print("Bob Private Parameter Y = %d" %Y)

print("")
```

```
    print("Shared Parameters :")
    print("==============")
    print("")

    print("Parameter shared with Bob by Alice U = %d" %U)
    print("Parameter shared with Alice by Bob V = %d" %V)

    print("")

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

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

    print("")

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]
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)
    " 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)
```

```
    print("")
    print("Public Parameters :")
    print("====================")
    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("Public Parameter H = %d" %H)
    print("")

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

    print("Private key X = %d" %X)
    print("Ephemeral key Y = %d" %Y)

    print("")

    print("Signature:")
    print("==========")
    print("")

    print("S1 = %d" %S1)
    print("S2 = %d" %S2)

    print("")

    print("Verification :")
    print("==============")
    print("")

    print("Wa = %d" %Wa)
    print("Wb = %d" %Wb)

    print("")

sys.exit
```

==========

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