# Key Recovery Attack on QuiSci

by

Nils Reimers
Rnils@web.de

October 11, 2009

## 1 Introduction

QUISCI(quick stream cipher) is a stream cipher designed by Stefan Müller (FGAN-FHR, a German research institute) in 2001. The base algorithm is very compact and simple. The main loop of algorithm simply consists of 6 operations:

| Step | Operation | |
|---|---|---|
| 1. | i := i + S[j] | |
| 2. | if i == 0 | else |
| 3. | i := not(S[j]) | k := i XOR S[j] |
| 4. | S[j] := k | S[j] := i |
| 5. | | j := i |
| 6. | C := C XOR i | C := C XOR k |

Table 1: Main loop of QuiSci

QuiSci is incredible fast, faster than most other ciphers. On modern CPUs it needs only arround 1 clock cycle per byte, so it is 10 times fast than most other well-known algorithm.

On the website of QuiSci [1] it is claimed that this algorithm is secure. With this paper I like to show a key recovery attack on QuiSci, exploiting the weak key setup.

When you are able to guess the beginning of the plaintext, its takes only a very small amount of time (less than a second) to decipher the whole message. You can find a implementation for this attack on [2].

## 2 Description of QuiSci

The base algorithm consists of these 6 operations [1]:

| Step | Operation | |
|---|---|---|
| 1. | i := i + S[j] | |
| 2. | if i == 0 | else |
| 3. | i := not(S[j]) | k := i XOR S[j] |
| 4. | S[j] := k | S[j] := i |
| 5. | | j := i |
| 6. | C := C XOR i | C := C XOR k |

Table 2: Encryption/Decryption with QuiSci

$i, j$ are $n$-bit integers, $S$ is an array of $n$-bit integers. The reference algorithm of QuiSci uses 64-bit integers and $S$ with 256 entries. So to calculate the new index $j$, $i$ have to by shifted by 56: $j := i >> 56$.

QuiSci can be run with various bit words/integers and various sizes of $S$. Also QuiSci has two modes of operations: The $N$ mode, the internal state is not influcend by the plaintext, and the $D$ mode, the internal state depends on the key and on the plaintext.

In this paper I will focus on the $N$ mode, with 64 bit integers and an internale state with 256 words. But it is easy to map this attack against other QuiSci implements with other word or internale state sizes. It should also be easy to map this attack against the $D$ mode of QuSci.

The following C# Code implements QuiSci. You can find the reference C code at http://quisci.awardspace.co

```
public class QuiSci
{
    private const int BOX_SIZE = 256;
    public const ulong INIT_NUMBER = 0x630b25afbfad135f;
    private const int INDEX_SHIFT = 56;

    public ulong[] SInit; //called 'boxInit' in the reference c
        code
    private ulong[] S; //called 'box' in the reference c code
    private ulong i; //called 'index' in the reference c code
    private ulong j; //called 'current' in the reference c code
    private ulong k; //called 'xor' in the reference c code

    //Initialize the QuiSci box: boxInit
    public QuiSci()
    {
        SInit = new ulong[BOX_SIZE];
        for (int l = 0; l < BOX_SIZE; l++)
            SInit[l] = INIT_NUMBER;

        i = 0;
        j = 0;
```

```
        for (ulong l = 0; l < BOX_SIZE; l++)
        {
            i += SInit[j] + INIT_NUMBER * l;
            SInit[j] = i;
            j = i >> INDEX_SHIFT;
        }
    }

    //Set up the key
    public void Init(byte[] key)
    {
        this.S = (ulong[])SInit.Clone();
        this.k = INIT_NUMBER;

        for (int l = 0; l < key.Length; l++)
        {
            this.k += key[l];
            this.S[l] = this.k;
        }

        this.k *= INIT_NUMBER;
        this.i = this.k;
        this.j = this.i >> INDEX_SHIFT;
        this.S[this.j] = this.k;
    }

    //Encrypt 1 word with QuiSci
    public ulong Encrypt(ulong data)
    {
        this.i += this.S[this.j];

        if (this.i == 0)
        {
            this.i = ~this.S[j];
            this.S[j] = this.k;
            data ^= i;
        }
        else
        {
            this.k = i ^ this.S[j];
            this.S[j] = i;
            this.j = i >> INDEX_SHIFT;
            data ^= k;
        }

        return data;
    }
}
```

## 3 Key Recovery Attack

($\oplus$ denotes exclusive or (XOR) and all additions are modulo $2^{64}$, because of the usage of 64 bit integers.)

The main attack is against the key setup (*Init* in the C# Code) of QuiSci.

In this QuiSci implementation, index is a 64 bit integer, so that the if statement 'index == 0' is in nearly all cases false. Because of that, it's okay just to focus on the else statement in the *Encrypt*-Methode (the chance that index is equal to 0 is 1 to $2^{64}-1$).

So the $n-th$[1] call of this encryption function can be written as (compare Table 2):

$$\begin{aligned}
i_{n+1} &:= i_n + S[j_n] \\
k_{n+1} &:= i_{n+1} \oplus S[j_n] \\
S[j_n] &:= i_{n+1} \\
j_{n+1} &:= (i_{n+1} >> 56) \\
C_n &:= P_n \oplus k_{n+1}
\end{aligned}$$

The values of $i_0$, $j_0$ are generated during the key setup (*Init*).

During the key setup (*Init*), you add all key bytes and save them in the variable $k$. So at the end of the loop $k$ (called $k_0$) is equal to:

$$k_0 := INIT\_NUMBER + key[0] + key[1] + key[2] + ... + key[m-1]$$

This result is multiplied by $INIT\_NUMBER$, so you get the final $k_0$:

$$k_0 := (INIT\_NUMBER + key[0] + ... + key[m-1]) \cdot INIT\_NUMBER$$

This $k_0$ is the first value of $i$ and $k >> 56$ is the first value of $j$ (called $i_0$ and $j_0$). Additional, the $j_0$ entry of the $S$-array is replaced by $k_0 = i_0$.

$$i_0 := k_0 \qquad j_0 := k_0 >> 56 \qquad S[j_0] := i_0 = k_0 \tag{1}$$

If you now encrypt the first plaintext, $P_0$, the resulting ciphertext is:

$$C_0 := P_0 \oplus k_1$$

$k_1$ is equal $i_1 \oplus S[j_0]$, so if you replace $k_1$ with this you get:

$$C_0 = P_0 \oplus k_1 = P_0 \oplus (i_1 \oplus S[j_0])$$

The $i_1$ value is the addition of $i_0$ and $S[j_0]$ so you get:

$$C_0 = P_0 \oplus k_1 = P_0 \oplus (i_1 \oplus S[j_0]) = P_0 \oplus ((i_0 + S[j_0]) \oplus S[j_0])$$

We know (see (1)) that $i_0 = k_0$ and $S[j_0] = i_0 = k_0$. When you now replace the variable of $i_0$ and $S[j_0]$ with $k_0$ you get:

$$C_0 = P_0 \oplus ((i_0 + S[j_0]) \oplus S[j_0]) = P_0 \oplus ((k_0 + k_0) \oplus k_0) \tag{2}$$

$$\Rightarrow C_0 \oplus P_0 = (k_0 + k_0) \oplus k_0 \tag{3}$$

$k_0 + k_0$ is equal to $k_0 << 1$, so you know that the least significant bit of $k_0 + k_0$ have to be a 0.

[1]beginning with 0

## 3.1 Known Plaintext Attack to get $k_0$

When you know the value of $P_0$, you can exploit equation (3) to get the value of $k_0$.

Be $x_0, x_1, ..., x_{n-1}$ the bits of $k_0$ and $y_0, ..., y_{n-1}$ the bits of $C_0 \oplus P_0$. Equation (3) can be written as:

$$\begin{array}{ccccccc} & x_{n-2} & x_{n-3} & ... & x_2 & x_1 & x_0 & 0 \\ \oplus & x_{n-1} & x_{n-2} & ... & x_3 & x_2 & x_1 & x_0 \\ \hline & y_{n-1} & y_{n-2} & ... & y_3 & y_2 & y_1 & y_0 \end{array}$$

Because you know the value of $y_0$ you can caculate $x_0$ ($x_0 = y_0$). After that, you can caculate $x_1$ ($x_1 = x_0 \oplus y_1$) and so on. At the end you can get the value of $k_0$.

This results into many attacks you can run against QuiSci.

### 3.1.1 Getting the digit sum of the key

As soon as you have calculated the value of $k_0$, you can get the digit sum ($key[0]+key[1]+...+key[m-1]$) of the key, because $k_0$ is equal to the digit sum of $key$ plus and multiplied by the $INIT\_NUMBER$:

$$k_0 = (INIT\_NUMBER + key[0] + ... + key[m-1]) \cdot INIT\_NUMBER$$

If you use a key with length of $m$ bytes, there are $255 \cdot m + 1$ possible digit sums. For example, for a 128 bit key, there are just 4081 possible digit sums. So just check all possible digit sum:

```
//ulong k0 : Use the above attack to get k0.
ulong keyLength = 16; //Length of the key in bytes
ulong tmp = 0;
int digitSum;
for (ulong i = 0; i <= (255 * keyLength); i++)
{
  tmp = (i+INIT_NUMBER)*INIT_NUMBER;
  if (tmp == k0)
  {
    digitSum = i;
    break;
  }
}
```

Listing 1: Caculates the digit sum of the key

There are just $C(s, m)$ possible keys with a digit sum of $s$ and a key length of $m$ bytes:

$$C(s, m) := \sum_{k=0}^{l} (-1)^k \cdot \binom{m}{k} \cdot \binom{s - 256k + nm - 1}{m - 1}$$

with $l := \min\left(m, \lfloor \frac{s}{256} \rfloor\right)$

```
C := (s,m) -> sum((-1)^k*binomial(m,k)*binomial(s-256*k+m-1, m
    -1), k=0..min(m, floor(s/256)));
```

Listing 2: C(s,m) for Maple

| Digit sum | Possible Keys |
|---|---|
| 100 | $2^{61}$ |
| 1000 | $2^{109}$ |
| 2000 | $2^{118}$ |
| 3000 | $2^{110}$ |

Table 3: Possible keys with 128 bits

For an 128 bit key, the peak of $C(s,m)$ is at $2^{118}$, so this attack reduces the key space to a maximum of $2^{118}$ keys. This is 1000 times faster than brute force.

But this isn't fast enough for a real key recovery attack against QuiSci.

## 3.2 Recovering the Internale State $S$

The security of a stream cipher is based on the internale state. If you are able to recover the internale state, you can break the cipher.

QuiSci uses a very weak key setup, so it is easy to recover the whole internale state:

```
//Set up the key
public void Init(byte[] key)
{
    this.S = (ulong[])SInit.Clone();
    this.k = INIT_NUMBER;

    for (int l = 0; l < key.Length; l++)
    {
        this.k += key[l];
        this.S[l] = this.k;
    }

    this.k *= INIT_NUMBER;
    this.i = this.k;
    this.j = this.i >> INDEX_SHIFT;
    this.S[this.j] = this.k;
}
```

Listing 3: Key Setup of QuiSci

A key with a length of $m$ bytes just affects the first $m$ values of our internale state $S$. The default internale state has 256 entries, so if you uses only a 16 bytes key (128 bit), just the first 16 entries of the $S$-Array are unknown. All other entries are equal to the initial values of $S$.

We know the value of $k_0$, $i_0$, $j_0$ and $S[j_0]$ so we can compute $i_1 := i_0 + S[j_0]$ and $j_1 := i_1 >> 56$. If $j_1 \geq m$, we know also the value of $S[j_1]$, because our $m$-byte key has just affected the first $m$ entries of $S$. With the value of $k_0$, you can decipher all blocks until $j_n < m$. With a 128 bits key you can on average break the next 14 cipher blocks (112 bytes of data), without any knowledge of them. Using just a 80 bits key increases the range to 40 cipher blocks (320 bytes of data).

```
//ulong[] c: An array with the ciphertext
//ulong k0: The value of k0

//The array with the decrypted ciphertext
ulong[] res = new ulong[c.Length];

ulong i, k, j;
ulong[] S = (ulong)QuiSci.SInit.Clone(); //The init value of S

//Calculate i1, j1
k = k0;
i = k0;
j = i >> 56;
S[j] = i;

for (int r = 0; r < c.Length; r++)
{
    i += S[j];

    if (i == 0)
    {
        i = ~S[j];
        S[j] = k;
        res[r] = c[r] ^ i;
    }
    else
    {
        k = i ^ S[j];
        S[j] = i;
        j = i >> 56;
        res[r] = c[r] ^ k;
    }

    //The value S[j] had been changed by the key
    if(j < keyLength)
        break;
}
```

Listing 4: Decrypt parts of the ciphertext without knowing the key

In the case of $j_n < m$, we do not know the value of $S[j_n]$. But that isn't such a problem. The value of $S[j_n]$ is:

$$S[j_n] = INIT\_NUMBER + key[0] + key[1] + ... + key[j_n]$$

This value of $S[j_n]$ is easy to brute force, because there are maximally $(j_n+1)\cdot 255$ possible values. When we know, or can guess, the value of $P_n$, the recovery of $S[j_n]$ is trivial.

## 4 Summary

If an attacker is able to guess the first word of the plaintext, he is able to decipher huge parts of the ciphertext. With a very small amount of time he is able to recover the whole internale state of QuiSci.

This paper shows a primitive known plaintext attack and a upper bound for the remaining key space can be found in Table 4. For this attack the attacker only knows, or can guess, the first $N$ plaintexts. The further plaintexts are considered to be random, so an attacker cannot use them to extend the attack [2].
This attack also works if you have to guess the values of the plaintext, because for a 16 bytes key you need only to guess maybe 25 plaintexts (some to verify your guess was right). The plaintext value of a cipher text is only needed if $j_n < m$. So it possible to know even less plaintexts.

| | Remaining Key Space | |
|---|---|---|
| N | 128 bit key | 256 bit key |
| 50 | $2^{94}$ | $2^{206}$ |
| 100 | $2^{74}$ | $2^{174}$ |
| 150 | $2^{57}$ | $2^{144}$ |
| 250 | $2^{35}$ | $2^{99}$ |
| 350 | $2^{23}$ | $2^{70}$ |
| 500 | $2^{14}$ | $2^{38}$ |
| 750 | $2^{6}$ | $2^{14}$ |

Table 4: Estimated key space for 128 and 256 bit keys

## 5 Conclusion

With this key setup, it is possible to decipher huge amounts of the ciphertext, only by knowing (or guessing) the first ciphertext. This attack shows, that QuiSci cannot be treated as secure algorithm. Instead, it is recommended to use other, well analysed algorithm like AES in a stream cipher mode or ISAAC.

## References


[1] http://quisci.awardspace.com/ current page of QUISCI, last update Mar 24, 2009 (accessed: Mar, 27 2009)

[2] www.php-einfach.de/crypt/quisci.php C# implementation to attack QuiSci


[2] truly random plaintext is really unrealistic, you can't even break a caesar cipher.