Time-Memory Trade-Off Attacks on Multiplications and T-functions

Joydip Mitra[1] and Palash Sarkar[2]

[1] Management Development Institute
Post Box No. 60, Mehrauli Road, Sukhrali
Gurgaon 122001, Haryana, India
joydip@mdi.ac.in
[2] Cryptology Research Group
Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108.
palash@isical.ac.in

Abstract. T–functions are a new class of primitives which have recently been introduced by Klimov and Shamir. The several concrete proposals by the authors have multiplication and squaring as core nonlinear operations. Firstly, we present time-memory trade-off algorithms to solve the problems related to multiplication and squaring. Secondly, we apply these algorithms to two of the proposals of multi-word T-functions. For the proposal based on multiplication we can recover the 128 unknown bits of the state vector in 2^{40} time whereas for the proposal based on squaring the 128 unknown bits can be recovered in 2^{21} time. The required amount of key stream is a few (less than five) 128-bit blocks. Experimental data from implementation suggests that our attacks work well in practice and hence such proposals are not secure enough for stand-alone usage. Finally, we suggest the use of conjugate permutations to possibly improve the security of T–functions while retaining some attractive theoretical properties.
Keywords: stream cipher, T–functions, multiplication, cryptanalysis, time-memory trade-off.

1 Introduction

Stream ciphers are a fundamental primitive in cryptography. Encryption is performed by XORing the message bit sequence with a pseudo-random bit sequence while decryption is performed by XORing the cipher bit sequence once more with the same pseudo-random bit sequence.

The cryptographic strength of a stream cipher depends on the unpredictability of the pseudo-random bit sequence. The other important issue is efficiency of the pseudo-random generator. Most practical proposals for stream ciphers strive to achieve a good balance between speed and security. Typically stream ciphers

are built out of linear feedback shift registers, nonlinear Boolean functions and S-boxes. See [4] for various models of stream ciphers.

Recently Klimov and Shamir [1–3] have proposed a new class of primitives for design of stream ciphers. They call their primitive T–functions and have developed a nice theory for analysing T–functions. From an efficiency point of view, T–functions are extremely attractive, since they can be built using fast and easily available operations on most processors. From a security point of view, there are many nice features including the single cycle property of the underlying permutation.

Klimov and Shamir [3] have also introduced multi-word T–functions and have extended their theory to cover such functions. In [3], they present several concrete constructions of multi-word T–functions. A key constituent of their proposals is multiplication modulo 2^{64}.

Our Contributions: In the first part of the paper, we study the following problem related to multiplication. Suppose x, y and z are n-bit integers satisfying $xy \bmod 2^n = z$. Further, suppose the m most significant bits of x, y and z are known. The problem is to compute all possible combinations of the $(n-m)$ least significant bits of x and y such that the multiplication holds.

We present a time-memory trade-off algorithm to solve this problem and make a detailed study of the effectiveness of the algorithm under different scenarios. We also study the related problem of squaring, i.e., when $x = y$. It turns out that the algorithm for multiplication is not efficient for squaring and hence we develop a separate algorithm to solve this problem. Apart from the application to T–functions, our algorithm can possibly be used for analysing other ciphers based on multiplication.

The second part of the paper consists of analysing the security of two concrete proposals of multi-word T–functions from [3]. The first proposal involves multiplication and the T-function operates on a state vector consisting of four 64-bit words. The pseudo-random bit sequence obtained from the state vector consists of the 32 most significant bits of each of the four 64-bit words. Thus the state vector has 128 unknown bits. We perform a detailed analysis of this T-function. The major step in the analysis consists of an application of (a modification) of the algorithm to solve multiplication as mentioned above. The final result that we obtain is that the 128 unknown bits can be computed in 2^{40} time which makes this proposal unsafe for stand-alone use as a pseudo-random generator.

The second proposal that we consider also operates on a state vector of four 64-bit words and produces 128 bits as before. The difference is that this proposal involves squaring instead of multiplication. Consequently, our analysis of this proposal involves the algorithm to solve squaring. In this case, we obtain an algorithm that determines the 128 unknown bits in 2^{21} time. Hence this proposal is much more insecure than the one based on multiplication.

The required amount of known pseudo-random key stream for both the above attacks is only a few (less than five) 128-bit consecutive key stream blocks. In

most cases, we expect the attack to work with only three 128-bit consecutive key stream blocks. This shows that these two proposals, and probably other similar proposals, are not secure enough for stand-alone usage.

One possibility for improving the security is to extract less number of bits from each state vector. We consider this possibility for the multiplication based T–function mentioned above, where only 16 most significant bits of each 64-bit word of the state vector is produced as output. Thus a total of 64 bits are produced from each state vector and 192 bits are unknown. Our attack also applies to this situation and the 192 unknown bits can be obtained in 2^{112} time. Though infeasible in practice, this constitutes a theoretical attack on the system.

We have implemented the algorithm to solve multiplication and our estimate of the expected run-time is supported by experimental data. We have also implemented the attack on a scaled down version of the multiplication based T–function. Instead of a state vector consisting of four 64-bit words we have worked with four 32-bit words. In this case, we can actually recover the 64 unknown bits of the state vector. This shows that our attack works quite well in practice. We have also implemented the algorithm to solve the squaring problem and the corresponding attack on the 64-bit version of the squaring based T-function proposal. Experiments show that the attack performs as predicted by the theoretical analysis.

Finally, we suggest a method based on conjugate permutations to possibly improve the security of T-functions while maintaining some desirable features such as the single cycle property.

2 Multiplication

We consider the following problem. Suppose two n-bit integers x and y are multiplied modulo 2^n to obtain an n-bit integer z. The m most significant bits (MSBs) of x, y and z are known and we have to find all possible solutions for the $(n-m)$ least significant bits (LSBs) of x and y such that $xy \bmod 2^n = z$. This problem can be stated more precisely as follows:

Problem : Mult

Input : Three integers $x^{(1)}$, $y^{(1)}$ and $z^{(1)}$ such that, $0 \leq x^{(1)}, y^{(1)}, z^{(1)} < 2^m$.

Task : Find all pairs of integers $(x^{(0)}, y^{(0)})$ such that, $0 \leq x^{(0)}, y^{(0)} < 2^{n-m}$, $x = 2^{n-m}x^{(1)} + x^{(0)}$, $y = 2^{n-m}y^{(1)} + y^{(0)}$ and

$$\left\lfloor \frac{xy \bmod 2^n}{2^{n-m}} \right\rfloor = \left\lfloor \frac{xy}{2^{n-m}} \right\rfloor \bmod 2^m = z^{(1)}. \tag{1}$$

Note that the operation $x \bmod 2^t$ returns the t LSBs of x and the operation $\lfloor x/2^t \rfloor$ returns $x \gg t$, i.e. the binary representation of x right shifted t times. The number of unknown bits in the pair $(x^{(0)}, y^{(0)})$ is $2(n-m)$ and the m known bits on the right hand side of (1) imposes m restrictions on these unknowns. Hence, *on an average*, one should expect $2^{2(n-m)-m} = 2^{2n-3m}$ distinct pairs of $(x^{(0)}, y^{(0)})$ to be solutions to Mult. See Section 5 for an emperical justification of this statement.

We first consider the naive approaches to solve Mult. There are $(2n - 2m)$ unknown bits and one approach is to try all possible combinations of these unknown bits. This approach requires 2^{2n-2m} time. The second naive approach using an offline table computation can be described as follows. For each possible pair of n-bit integers (x, y) compute the product $z = xy \bmod 2^n$. Store in $\mathsf{Tab}[x^{(1)}, y^{(1)}, z^{(1)}]$ the set of all pairs $(x^{(0)}, y^{(0)})$ which are solutions to Mult for the instance $x^{(1)}, y^{(1)}, z^{(1)}$. This table takes 2^{2n} time to prepare and store. The preparation of the table can be done offline. Given a particular instance $x^{(1)}, y^{(1)}, z^{(1)}$ of Mult the solutions can be directly obtained from the entries of the row $\mathsf{Tab}[x^{(1)}, y^{(1)}, z^{(1)}]$. Since, on an average, there are 2^{2n-3m} solutions, at least this amount of online time will be required in producing the solutions.

Thus the online time will be at least 2^{2n-3m} (requiring 2^{2n} precomputation time and a table of size 2^{2n}) and at most 2^{2n-2m} (using exhaustive online search but without using any look-up table). We describe solutions to Mult whose online time complexity is between the two extreme values and which uses a table of moderate size. Thus our algorithms can be considered to be time-memory tradeoff algorithms.

To improve readability, we will use the same notation for an integer and its binary representation. Also the length of a binary string will be denoted by $|.|$. Thus a binary string x of length $|x| = k$ denotes an integer $x \in \{0, \ldots, 2^k - 1\}$. For two binary strings x_1 and x_2, by (x_2, x_1) we denote the binary string x obtained by concatenating x_2 and x_1. Using the integer representation of x_1, x_2 and x we have $x = 2^{|x_1|}x_2 + x_1$.

Using this notation, we write $x = x^{(1)}2^{n-m} + x^{(0)}$, $y = y^{(1)}2^{n-m} + y^{(0)}$ and $z = z^{(1)}2^{n-m} + z^{(0)}$, where $|x| = |y| = |z| = n$, $|x^{(1)}| = |y^{(1)}| = |z^{(1)}| = m$ and $|x^{(0)}| = |y^{(0)}| = |z^{(0)}| = n - m$. We now introduce parameters n_0, n_1 and n_2 defined by the following equations.

$$\left.\begin{array}{l} x = X^{(2)}2^{n_1+n_0} + X^{(1)}2^{n_0} + X^{(0)} \\ y = Y^{(1)}2^{n_1} + Y^{(0)} \\ z = Z^{(1)}2^{n_1+n_0} + Z^{(0)} \end{array}\right\} \tag{2}$$

where $|X^{(2)}| = n_2$, $|X^{(1)}| = n_1$, $|X^{(0)}| = n_0$, $|Y^{(1)}| = n - n_1$, $|Y^{(0)}| = n_1$, $|Z^{(1)}| = n_2$ and $|Z^{(0)}| = n_1 + n_0$. We require these parameters to satisfy certain conditions. These conditions are given below.

1. $n_0 + n_1 + n_2 = n$: This is required since x, y and z are n-bit integers.
2. $n_0 \leq n - m$: This ensures that $X^{(0)}$ is a suffix of $x^{(0)}$.
3. $n_2 \leq m$: This ensures that $X^{(2)}$ is a prefix of $x^{(1)}$.
4. $n_1 \leq n - m$: This ensures that $Y^{(0)}$ is a suffix of $y^{(0)}$.
5. $n_1 \leq n_2$: This ensures that the expected number of entries in each row of $\mathsf{Tab}[\,]$ (see later) is one. The case $n_1 > n_2$ is also feasible but does not provide better results.
6. $n_2 + n_1 > m$: The case $n_2 + n_1 \leq m$ is also feasible, but does not provide better results and hence we do not consider it.

We now define binary strings $U^{(1)}, U^{(2)}$ and $V^{(1)}$ in the following manner. The strings $U^{(1)}$ and $U^{(2)}$ are such that $x^{(0)} = (U^{(1)}, X^{(0)})$ and $X^{(1)} = (U^{(2)}, U^{(1)})$,

where $|U^{(1)}| = n - m - n_0$, $|U^{(2)}| = m - n_2$. Then $x^{(1)} = (X^{(2)}, U^{(2)})$. The string $V^{(1)}$ is such that $y^{(0)} = (V^{(1)}, Y^{(0)})$, where $|V^{(1)}| = n - m - n_1$. Then $Y^{(1)} = (y^{(1)}, V^{(1)})$. Note that the portion $U^{(2)}$ of $X^{(1)}$ is provided as part of the input whereas the part $U^{(1)}$ of $X^{(1)}$ has to be determined. Also the string $V^{(1)}$ is part of $y^{(0)}$ and has to be determined. These substrings are shown in Figure 1.

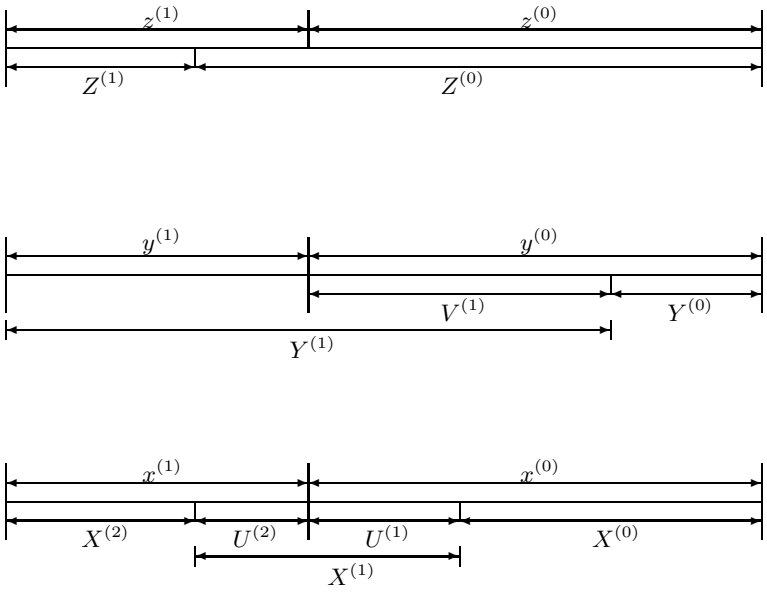

Fig. 1. Definitions of substrings

Our algorithm is based on the following result.

Proposition 1. $\left\lfloor \frac{xy}{2^{n-n_2}} \right\rfloor = r + d + c$, *where* $r = \left\lfloor \frac{Y^{(1)} x 2^{n_1}}{2^{n-n_2}} \right\rfloor$, $d = Y^{(0)}X^{(2)} + \left\lfloor \frac{Y^{(0)}X^{(1)}2^{n_0}}{2^{n-n_2}} \right\rfloor$ *and* $c \in \{0, 1, 2\}$.

Proof : We write

$$\left\lfloor \frac{xy}{2^{n-n_2}} \right\rfloor = \left\lfloor \frac{xY^{(1)}2^{n_1}}{2^{n-n_2}} + Y^{(0)}X^{(2)} + \frac{Y^{(0)}X^{(1)}2^{n_0}}{2^{n-n_2}} + \frac{Y^{(0)}X^{(0)}}{2^{n-n_2}} \right\rfloor.$$

Note that $r = \left\lfloor \frac{xY^{(1)}2^{n_1}}{2^{n-n_2}} \right\rfloor = \left\lfloor X^{(2)}Y^{(1)}2^{n_1} + X^{(1)}Y^{(1)} + \frac{X^{(0)}Y^{(1)}2^{n_1}}{2^{n_0+n_1}} \right\rfloor$. Now

$$\left\lfloor \frac{X^{(0)}Y^{(1)}2^{n_1}}{2^{n_0+n_1}} + \frac{Y^{(0)}X^{(1)}2^{n_0}}{2^{n-n_2}} + \frac{Y^{(0)}X^{(0)}}{2^{n-n_2}} \right\rfloor$$

$$= \left\lfloor \frac{X^{(0)}Y^{(1)}2^{n_1}}{2^{n_0+n_1}} \right\rfloor + \left\lfloor \frac{Y^{(0)}X^{(1)}2^{n_0}}{2^{n-n_2}} \right\rfloor + \left\lfloor \frac{Y^{(0)}X^{(0)}}{2^{n-n_2}} \right\rfloor + c$$

for some $c \in \{0, 1, 2\}$. Further, since $Y^{(0)} < 2^{n_1}$, $X^{(0)} < 2^{n_0}$, we have $X^{(0)}Y^{(0)} < 2^{n_0+n_1} = 2^{n-n_2}$ and hence $\left\lfloor Y^{(0)}X^{(0)}/2^{n-n_2} \right\rfloor = 0$. Putting all these together gives us the required result. □

Based on Proposition 1 we have the following algorithm to solve Mult. The algorithm uses a table Tab[ ] which is prepared in the first phase and is used to solve Mult in the second phase.

Algorithm 1
Input: $x^{(1)}, y^{(1)}$ and $z^{(1)}$.

1. Write $x^{(2)} = (X^{(2)}, U^{(2)})$, where $|X^{(2)}| = n_2$ and $|U^{(2)}| = m - n_2$;
2. set $Z^{(1)}$ to the n_2 most significant bits of $z^{(1)}$;
3. for $U^{(1)} \in \{0,1\}^{n-m-n_0}$
4. set $X^{(1)} = (U^{(2)}, U^{(1)})$;
5. for $Y^{(0)} \in \{0,1\}^{n_1}$
6. compute $d^{(1)} = Y^{(0)}X^{(2)} + \lfloor (Y^{(0)}X^{(1)}2^{n_0})/2^{n-n_2} \rfloor \bmod 2^{n_2}$;
7. $\mathsf{Tab}[d^{(1)}] = \mathsf{Tab}[d^{(1)}] \cup \{Y^{(0)}\}$;
8. end for;
9. for $(X^{(0)}, V^{(1)}) \in \{0,1\}^{n_0} \times \{0,1\}^{n-m-n_1}$
10. set $Y^{(1)} = (y^{(1)}, V^{(1)})$; set $x = (X^{(2)}, X^{(1)}, X^{(0)})$;
11. compute $r^{(1)} = \lfloor (xY^{(1)}2^{n_1})/2^{n-n_2} \rfloor \bmod 2^{n_2}$;
12. for $c \in \{0, 1, 2\}$
13. compute $d^{(2)} = Z^{(1)} - r^{(1)} - c \pmod{2^{n_2}}$;
14. for each $Y^{(0)} \in \mathsf{Tab}[d^{(2)}]$
15. set $y = (Y^{(1)}, Y^{(0)})$;
16. if $(\lfloor (xy)/2^{n-m} \rfloor \bmod 2^m = z^{(1)})$ then
17. set $x^{(0)} = (U^{(1)}, X^{(0)})$ and $y^{(0)} = (V^{(1)}, Y^{(0)})$;
18. output $(x^{(0)}, y^{(0)})$;
19. end if;
20. end for;
21. end for;
22. end for;
23. end for;

2.1 Complexity of Algorithm 1

The space complexity of Algorithm 1 is the space required to store Tab[ ]. By construction Tab[ ] has $2^{|d^{(1)}|} = 2^{n_2}$ rows and a total of $2^{|Y^{(0)}|} = 2^{n_1}$ entries in all the rows. By Condition 5 after Equation (2) we have $n_1 \leq n_2$ and hence on an average the number of entries in each row of Tab[ ] is at most one.

The time required by Algorithm 1 depends on the number of entries in a row of Tab[ ]. The expected number of such entries is one and this allows us to obtain

the expected run-time R of Algorithm 1:

$$\left.\begin{aligned} R &= 2^{|U^{(1)}|}\left(2^{|Y^{(0)}|} + 3 \times 2^{|X^{(0)}|} \times 2^{|V^{(1)}|}\right) \\ &= 2^{n-m-n_0}(2^{n_1} + 3 \times 2^{n-m-n_1+n_0}) \\ &= 2^{n-m-n_0+n_1} + 3 \times 2^{2(n-m)-n_1} \end{aligned}\right\} \quad (3)$$

We now consider two cases and obtain the value of R in each case.
Case 1: $n_2 = m$. Hence $n_0 + n_1 = n - m$. In this case $R = 2^{2n_1} + 3 \times 2^{2(n-m)-n_1}$.
Subcase 1a: $n = 64$ and $m = 32$. Then $R = 2^{2n_1} + 3 \times 2^{64-n_1}$. This expression is minimized when $n_1 = 22$, whence $R = 2^{44} + 3 \times 2^{42} = 7 \times 2^{42}$.
Subcase 1b: $n = 64$ and $m = 16$. Then $R = 2^{2n_1} + 3 \times 2^{96-n_1}$. Since $n_1 \leq n_2 = m = 16$, the maximum value of n_1 is 16. Choosing $n_1 = 16$ gives $R = 2^{32} + 3 \times 2^{80}$.
Case 2: $n_2 < m$. This case is more complicated to analyse and we first perform a special case analysis by setting $n_2 = n_1$. Then $n_0 = n - 2n_1$ and $R = 2^{3n_1-m} + 3 \times 2^{2(n-m)-n_1}$.
Subcase 2a: $n = 64$ and $m = 32$. Choosing $n_1 = 24$ we have $R = 2^{40} + 3 \times 2^{40} = 2^{42}$.

In general $n_2 \neq n_1$. However, we have verified that for $n = 64$ and $m = 32$ and for all possible distinct values of n_0, n_1 and n_2, the value of R is minimized for $n_2 = n_1 = 24$ and $n_0 = 16$. Thus the special case is also optimal for the general case. In fact, for $n = 64$ and $m = 32$, $R = 2^{42}$ is the minimum possible expected run-time for Algorithm 1.

2.2 Offline Table Preparation

The expected run-time of Algorithm 1 can be made optimal by using a larger table which can be prepared offline. We describe this idea for $n = 64$ and $m = 32$. Write $x = 2^{32}x^{(1)} + x^{(0)}$ and $y = 2^{32}y^{(1)} + y^{(0)}$. We write $\lfloor (xy)/2^{32} \rfloor = d + r$, where $d = \lfloor (xy^{(0)})/2^{32} \rfloor$ and $r = xy^{(1)}$.

In the offline table preparation phase, for each $(x^{(1)}, x^{(0)}, y^{(0)}) \in \{0,1\}^{32} \times \{0,1\}^{32} \times \{0,1\}^{32}$, we compute $d = \lfloor (xy^{(0)})/2^{32} \rfloor \bmod 2^{32}$ and set
$\mathsf{Tab}[x^{(1)}, x^{(0)}, d] = \mathsf{Tab}[x^{(1)}, x^{(0)}, d] \cup \{y^{(0)}\}$.

In the online phase, we are given $x^{(1)}, y^{(1)}$ and $z^{(1)}$. For each possible value of $x^{(0)} \in \{0,1\}^{32}$, we compute $r = xy^{(1)} \bmod 2^{32}$; $d = r - z^{(1)} \bmod 2^{32}$ and for each $y^{(0)} \in \mathsf{Tab}[x^{(1)}, x^{(0)}, d]$ output $(x^{(0)}, y^{(0)})$.

The run-time for table preparation is 2^{96}; the space required to store $\mathsf{Tab}[\,]$ is also 2^{96} and the (expected) runtime of the online phase is 2^{32}. Since there are 2^{32} solutions, the online run-time is the minimum possible. This comes at an expense of huge offline processing time and space.

3 Squaring

In the case $x = y$, the problem Mult reduces to squaring which can be formally stated as follows.

Problem : Sqr
Input : Two integers $x^{(1)}$ and $z^{(1)}$ such that, $0 \leq x^{(1)}, z^{(1)} < 2^m$.
Task : Find all integers $x^{(0)}$ such that, $0 \leq x^{(0)} < 2^{n-m}$, $x = 2^{n-m}x^{(1)} + x^{(0)}$ and

$$\left\lfloor \frac{x^2 \bmod 2^n}{2^{n-m}} \right\rfloor = \left\lfloor \frac{x^2}{2^{n-m}} \right\rfloor \bmod 2^m = z^{(1)}. \tag{4}$$

Note that there are $(n-m)$ unknown bits and m constraints. Hence the expected number of solutions is $\max(1, 2^{n-2m})$. If $n = 2m$, then the expected number of solutions is one. Algorithm 1 is not very efficient for Sqr so that we have to deal with the problem separately.

Let n_0, n_1 be such that $n_0 + n_1 = n - m$ and $x = 2^{n-m}X^{(2)} + 2^{n_0}X^{(1)} + X^{(0)}$, where $|X^{(2)}| = m$, $|X^{(1)}| = n_1$ and $|X^{(0)}| = n_0$ with $n_0 \leq m$.

Proposition 2. $\left\lfloor \frac{x^2}{2^{n_1+2n_0}} \right\rfloor = \left\lfloor \frac{z^{(1)}}{2^{n_0}} \right\rfloor = r + d + c$, *where* $r = 2^{n_1}\left(X^{(2)}\right)^2 + \left\lfloor \frac{\left(X^{(1)}\right)^2}{2^{n_1}} \right\rfloor + 2X^{(2)}X^{(1)}$, $d = \left\lfloor \frac{2X^{(2)}X^{(0)}}{2^{n_0}} \right\rfloor$ *and* $c \in \{0, 1, 2, 3\}$.

Based on Proposition 2, we have the following algorithm to solve Sqr.

Algorithm 2
Input: $x^{(1)}, z^{(1)}$.

1. set $k = n - (n_1 + 2n_0) = m - n_0$;
2. for $X^{(0)} \in \{0,1\}^{n_0}$
3. compute $d = \lfloor (2X^{(2)}X^{(0)})/2^{n_0} \rfloor \bmod 2^k$;
4. $\mathsf{Tab}[d] = \mathsf{Tab}[d] \cup \{X^{(0)}\}$;
5. end for;
6. for $X^{(1)} \in \{0,1\}^{n_1}$
7. compute $r = 2^{n_1}(X^{(2)})^2 + \lfloor (X^{(1)})^2/2^{n_1} \rfloor + 2X^{(2)}X^{(1)} \bmod 2^k$;
8. for each $c \in \{0, \ldots, 3\}$
9. compute $d = \lfloor z^{(1)}/2^{n_0} \rfloor - r - c \bmod 2^k$;
10. for each $X^{(0)} \in \mathsf{Tab}[d]$
11. if $(\lfloor x^2/2^{n-m} \rfloor \bmod 2^m = z^{(1)})$ then output $(X^{(1)}, X^{(0)})$;
12. end for;
13. end for;
14. end for;

The space complexity of Algorithm 2 is 2^{m-n_0} and the (expected) time complexity is $R = 2^{n_0} + 4 \times 2^{n_1}$.
Case 1: $n = 64$, $m = 32$. In this case we choose $n_0 = n_1 = 16$. Then the space complexity is 2^{16} and the time complexity is $R = 2^{16} + 4 \times 2^{16} = 5 \times 2^{16}$. This particular choice of n_0 and n_1 minimizes the value of R.
Case 2: $n = 64$, $m = 16$. Choosing $n_0 = 8$ and $n_1 = 40$ gives a run-time $R = 2^8 + 4 \times 2^{40}$.

4 Attacks on T-functions

We consider two specific proposals of multiword T-functions from [3] and describe attacks on them. These T–functions operate on an internal state vector which consists of four 64-bit words. Applying a T–function once to the state vector changes the value of each of the four 64-bit words. As suggested in [3], the extracted output consists of the most significant 32 bits of each of the four 64-bit words of the state vector. Thus applying the T–function repeatedly to the state vector produces a sequence of 128-bit (four 32-bit words) output blocks. These output blocks are treated as the generated pseudo-random sequence. The secret key consists of the initial 256-bit (four 64-bit words) value of the state vector.

For the attack we will assume that several consecutive output blocks are known. We actually require only two consecutive output blocks to perform the attack and a few more to verify the correctness. The goal of our attack is to obtain the complete 256-bit (four 64-bit words) value of the internal state vector at some point of time.

For a 64-bit word w, let $\mathsf{msb}(w)$ (resp. $\mathsf{lsb}(w)$) denote the 32 most (resp. least) significant bits of w. Let (x_0, x_1, x_2, x_3) be the internal state vector at some point of time. Let (y_0, y_1, y_2, y_3) be the state vector after application of T to (x_0, x_1, x_2, x_3), i.e., $(y_0, y_1, y_2, y_3) = T(x_0, x_1, x_2, x_3)$. The outputs corresponding to (x_0, x_1, x_2, x_3) and (y_0, y_1, y_2, y_3) are $(\mathsf{msb}(x_0), \mathsf{msb}(x_1), \mathsf{msb}(x_2), \mathsf{msb}(x_3))$ and $(\mathsf{msb}(y_0), \mathsf{msb}(y_1), \mathsf{msb}(y_2), \mathsf{msb}(y_3))$ respectively. We assume that these outputs are known and our attack is to compute $(\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2), \mathsf{lsb}(x_3))$.

There are a total of 128 unknown bits in (x_0, x_1, x_2, x_3) and a method to obtain them in time less than 2^{128} constitutes an attack on the system. Our algorithms are much more efficient – the attacks in Section 4.1 and Section 4.2 require time 2^{40} and 2^{21} respectively to compute the 128 unknown bits.

4.1 Attack on Multiplication Based T–function

Consider the following T–function:

$$T\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_0 \oplus s & \oplus (2(x_1 \vee C_1)x_2) \\ x_1 \oplus (s \wedge a_0) & \oplus (2x_2(x_3 \vee C_3)) \\ x_2 \oplus (s \wedge a_1) & \oplus (2(x_3 \vee C_3)x_0) \\ x_3 \oplus (s \wedge a_2) & \oplus (2x_0(x_1 \vee C_1)) \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{5}$$

where $a_0 = x_0$, $a_i = a_{i-1} \wedge x_i$, $1 \leq i < 4$, $s = a_3 \oplus (a_3 + C_0)$. Also, C_0 is odd and known, $C_1 = (12481248)_{16}$ and $C_3 = (48124812)_{16}$ (Equation (13) in Klimov and Shamir [3]). Each of C_1 and C_3 are considered to be 64-bit words where the leading 32 bits are all zeros.

During use of this T-function as pseudo-random generator, the quantities $\mathsf{msb}(x_i), \mathsf{msb}(y_i)$ are known for $i = 0, 1, 2, 3$. Our attempt will be to obtain $\mathsf{lsb}(x_i)$ for $i = 0, 1, 2, 3$. This proceeds in several steps.

Step 1:

First note that $\mathsf{msb}(w_1 \oplus w_2) = \mathsf{msb}(w_1) \oplus \mathsf{msb}(w_2)$ and $\mathsf{msb}(w_1 \wedge w_2) = \mathsf{msb}(w_1) \wedge \mathsf{msb}(w_2)$. Hence we have $\mathsf{msb}(a_0) = \mathsf{msb}(x_0)$, $\mathsf{msb}(a_1) = \mathsf{msb}(x_0) \wedge \mathsf{msb}(x_1)$, $\mathsf{msb}(a_2) = \mathsf{msb}(x_0) \wedge \mathsf{msb}(x_1) \wedge \mathsf{msb}(x_2)$ and $\mathsf{msb}(a_3) = \mathsf{msb}(x_0) \wedge \mathsf{msb}(x_1) \wedge \mathsf{msb}(x_2) \wedge \mathsf{msb}(a_3)$. The quantity s involves an addition mod 2^{64} and cannot be directly tackled in this manner. However, we can determine the upper part of s with only one bit of uncertainty in the following manner. First note that we have, $\mathsf{msb}(a_3 + C_0 \bmod 2^{64}) = \mathsf{msb}(a_3) + \mathsf{msb}(C_0) + \epsilon \bmod 2^{32}$ where ϵ is the carry of $\mathsf{lsb}(a_3) + \mathsf{lsb}(C_0)$ and hence $\epsilon = 0, 1$. Thus

$$\mathsf{msb}(s) = \mathsf{msb}(a_3) \oplus \left(\mathsf{msb}(a_3) + \mathsf{msb}(C_0) + \epsilon \bmod 2^{32}\right)$$

and hence $\mathsf{msb}(s)$ can take only two values as determined by ϵ.

Thus, with respect to the known 32 most significant bits, equation 5 reduces to

$$\begin{pmatrix} \mathsf{msb}\left(2(x_1 \vee C_1)x_2\right) \\ \mathsf{msb}\left(2x_2(x_3 \vee C_3)\right) \\ \mathsf{msb}\left(2(x_3 \vee C_3)x_0\right) \\ \mathsf{msb}\left(2x_0(x_1 \vee C_1)\right) \end{pmatrix} = \begin{pmatrix} \mathsf{msb}(y_0) \oplus \mathsf{msb}(x_0) \oplus \mathsf{msb}(s) \\ \mathsf{msb}(y_1) \oplus \mathsf{msb}(x_1) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_0)) \\ \mathsf{msb}(y_2) \oplus \mathsf{msb}(x_2) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_1)) \\ \mathsf{msb}(y_3) \oplus \mathsf{msb}(x_3) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_2)) \end{pmatrix}. \quad (6)$$

Equation (6) gives a relation between known quantities. Let $W_0 = 2x_0$, $W_1 = (x_1 \vee C_1)$, $W_2 = 2x_2$ and $W_3 = (x_3 \vee C_3)$. Also let $K_0 = \mathsf{msb}(y_0) \oplus \mathsf{msb}(x_0) \oplus \mathsf{msb}(s)$, $K_1 = \mathsf{msb}(y_1) \oplus \mathsf{msb}(x_1) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_0))$, $K_2 = \mathsf{msb}(y_2) \oplus \mathsf{msb}(x_2) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_1))$ and $K_3 = \mathsf{msb}(y_3) \oplus \mathsf{msb}(x_3) \oplus (\mathsf{msb}(s) \wedge \mathsf{msb}(a_2))$. Our next step is to solve for W_0, W_1, W_2 and W_3 such that

$$\mathsf{msb}(W_1W_2) = K_0, \mathsf{msb}(W_2W_3) = K_1, \mathsf{msb}(W_3W_0) = K_2, \mathsf{msb}(W_0W_1) = K_3 \quad (7)$$

Since there are two choices of ϵ, the rest of the steps have to be carried out for each value of ϵ.

Step 2:

We use Algorithm 1 to solve (7). There are, however, a few adjustments, which improve the run-time of Algorithm 1. Note that due to masking with C_1 and C_3, eight bits of each of $\mathsf{lsb}(W_1)$ and $\mathsf{lsb}(W_3)$ are fixed and known. Also $W_0 = 2x_0$. We know $\mathsf{msb}(x_0)$ which means we do not know the last bit of $\mathsf{msb}(W_0)$ which is equal to the first bit of $\mathsf{lsb}(x_0)$. To apply Algorithm 1 we have to know all the 32 bits of $\mathsf{msb}(W_0)$. This means that we have to guess the last bit of $\mathsf{msb}(W_0)$. On the other hand, since $W_0 = 2x_0$, the last bit of W_0 (and hence of $\mathsf{lsb}(W_0)$) is zero. Similar considerations hold for W_2.

Now suppose we are solving for $\mathsf{lsb}(W_0)$ and $\mathsf{lsb}(W_1)$ from the equation $\mathsf{msb}(W_0W_1) = K_3$. While invoking Algorithm 1, we let W_1 play the role of x and W_0 play the role of y. Further, we choose $n_2 = n_1 = 24$ (Subcase 2a in Section 2.1). Then in Algorithm 1, $|U^{(1)}| = 16$, $|Y^{(0)}| = 24$, $|X^{(0)}| = 16$ and $|V^{(1)}| = 8$. As mentioned before, due to the masking of $W^{(1)}$ with C_1, four bits of each of $U^{(1)}$ and $X^{(0)}$ are fixed to be one. Hence the number of choices of $U^{(1)}$ in Step 3 and $X^{(0)}$ in Step 9 of Algorithm 1 both reduces to 2^{12} from 2^{16}. The last bit of $Y^{(0)}$ is zero and hence the number of choices of $Y^{(0)}$ in Step 5

of Algorithm 1 reduces to 2^{23} from 2^{24}. The length of $V^{(1)}$ is eight. However, we also need to guess the last bit of $\mathsf{msb}(W_0)$, which is the next bit after $V^{(1)}$. Thus the number of possible choices of $V^{(1)}$ in Step 9 of Algorithm 1 increases to 2^9 from 2^8.

In Step 18, Algorithm 1 produces $(\mathsf{lsb}(W_1), \mathsf{lsb}(W_0))$ as output. The modification described above also determines the last bit of $\mathsf{msb}(W_1)$. Suppose this bit is b. By definition, the last bit of $\mathsf{lsb}(W_0)$ is zero. Then $\mathsf{lsb}(x_0)$ is obtained by prefixing b to $\mathsf{lsb}(W_0)$ and dropping the last bit. We assume that the modified Algorithm 1 produces $(\mathsf{lsb}(W_1), \mathsf{lsb}(x_0))$ as output. Similar considerations hold for the other equations in (7).

Recall from Equation (3) that the original expression for the expected runtime of Algorithm 1 is $R = 2^{|U^{(1)}|}\left(2^{|Y^{(0)}|} + 3 \times 2^{|X^{(0)}|} \times 2^{|V^{(1)}|}\right)$. Due to the changes in the number of possible choices of $U^{(1)}, Y^{(0)}, X^{(0)}$ and $V^{(1)}$, as explained above, this expression reduces to

$$\begin{aligned} R &= 2^{16-4}(2^{24-1} + 3 \times 2^{16-4} \times 2^9) \\ &= 2^{35} + 3 \times 2^{33} = 7 \times 2^{33} < 2^{36}. \end{aligned}$$

The time for solving one equation in (7) is approximately 2^{36} and hence the total time to solve all four equations is 2^{38}. The solutions to (7) are stored in separate lists, as we explain below. Define $w_i = \mathsf{lsb}(W_i)$ for $i = 1, 3$ and $w_i = \mathsf{lsb}(x_i)$ for $i = 0, 2$.

- Lst10 stores (w_1, w_0), sorted on w_1, such that $\mathsf{msb}(W_0W_1) = K_3$.
- Lst12 stores (w_1, w_2), sorted on w_1, such that $\mathsf{msb}(W_1W_2) = K_0$.
- Lst30 stores (w_3, w_0), sorted on w_3, such that $\mathsf{msb}(W_3W_0) = K_2$.
- Lst32 stores (w_3, w_2), sorted on w_3, such that $\mathsf{msb}(W_2W_3) = K_1$.

Step 3:

The next task is to "merge" the four lists to obtain solutions (w_0, w_1, w_2, w_3) which are consistent with all four equations. This is done as follows.

- Merge Lst10 and Lst12 on w_1 to obtain list Lst102 containing pairs of the form (w_0, w_2, w_1).
- Merge Lst30 and Lst32 on w_1 to obtain list Lst302 containing pairs of the form (w_0, w_2, w_3).
- Sort each of Lst102 and Lst302 on (w_0, w_2).
- Merge Lst102 and Lst302 on (w_0, w_2) to obtain a list Fin which contains tuples of the form (w_0, w_1, w_2, w_3) which are solutions to (7).

The time for merging and sorting (ignoring logarithmic factors) is 2^{32} and hence the above steps can be completed in approximately 2^{34} steps.

We consider the expected number of solutions to (7). There are 24 unknown bits in each of $\mathsf{lsb}(W_1)$ and $\mathsf{lsb}(W_3)$. On the other hand, there are 31 unknown bits in each of $\mathsf{lsb}(W_0)$ and $\mathsf{lsb}(W_2)$. In addition, we have to determine the last bits of both $\mathsf{msb}(W_0)$ and $\mathsf{msb}(W_2)$. Thus there are a total of 112 unknown bits in (7). Each of the equations in (7) provide 32 restrictions on these unknown

bits. Hence there are a total of 128 restrictions on these 112 unknown bits. Thus, on an average, we can expect the solution to (7) to be unique. See Section 5 for an emperical justification of this statement.

Step 4:

The list Fin contains the possible solutions (w_0, w_1, w_2, w_3). Now $w_i = \mathsf{lsb}(W_i)$ for $i = 1, 3$ and we want $\mathsf{lsb}(x_i)$. As mentioned before, the masking of x_1 and x_3 by C_1 and C_3 respectively fixes 8 bits each of W_1 and W_3. Thus from $\mathsf{lsb}(W_1)$ and $\mathsf{lsb}(W_3)$ we do not obtain the values of these 16 bits of $\mathsf{lsb}(x_1)$ and $\mathsf{lsb}(x_3)$. Instead, for each possible solution (w_0, w_1, w_2, w_3) in Fin and each possible value of these 16 bits, we construct a possible solution $(\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2), \mathsf{lsb}(x_3))$ and verify it using the definition of the T–function given in (5) and a few more outputs of the pseudo-random generator. The expected number of solutions $(\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2), \mathsf{lsb}(x_3))$ is also one and the complexity of this step is 2^{16}.

This completes the description of the attack. By combining all the complexities, we see that the complexity of the attack is less than 2^{40} in determining the 128 unknown bits of $(\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2), \mathsf{lsb}(x_3))$. This makes the attack quite practical and suggests that this T–function should not be used as a stand-alone pseudo-random generator.

4.2 Attack on Squaring Based T–Function

Consider the following T–function:

$$T\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_0 \oplus s & \oplus\, x_1^2 \wedge M \\ x_1 \oplus s \wedge a_0 & \oplus\, x_2^2 \wedge M \\ x_2 \oplus s \wedge a_1 & \oplus\, x_3^2 \wedge M \\ x_3 \oplus s \wedge a_2 & \oplus\, x_0^2 \wedge M \end{pmatrix} \tag{8}$$

where, $a_0 = x_0$, $a_i = a_{i-1} \wedge x_i$, $1 \le i < 4$, $s = a_3 \oplus (a_3 + 1)$, and $M = 1 \ldots 1110_2$. This is Equation (10) in [3]. Since, $\mathsf{msb}(a_0) = \mathsf{msb}(x_0)$ and $\mathsf{msb}(a_i) = \mathsf{msb}(a_{i-1}) \wedge \mathsf{msb}(x_i)$, $1 \le i < 4$, we know $\mathsf{msb}(a_i)$ for $0 \le i < 4$. Now,

$$\begin{aligned} s &= \big(\mathsf{msb}(a_3) \times 2^{32} + \mathsf{lsb}(a_3)\big) \oplus \big(\mathsf{msb}(a_3) \times 2^{32} + \mathsf{lsb}(a_3) + 1\big) \\ &= 2^{32}\left\{\mathsf{msb}(a_3) \oplus (\mathsf{msb}(a_3) + \epsilon)\right\} + \left\{\mathsf{lsb}(a_3) \oplus \big(\mathsf{lsb}(a_3) + 1 \bmod 2^{32}\big)\right\} \end{aligned}$$

where ϵ is the carry of $\mathsf{lsb}(a_3) + 1$. If $\epsilon = 1$, then $\mathsf{lsb}(a_3)$ equals $1 \ldots 111_2 = 2^{32} - 1$. But, $\mathsf{lsb}(a_3) = \mathsf{lsb}(x_0) \wedge \mathsf{lsb}(x_1) \wedge \mathsf{lsb}(x_2) \wedge \mathsf{lsb}(x_3)$ and hence $\mathsf{lsb}(x_i) = 1 \ldots 111_2$ for $0 \le i < 4$. In other words, $\epsilon = 1 \Rightarrow \mathsf{lsb}(x_i) = 1 \ldots 111_2$ for $0 \le i < 4$ and we can verify if this is indeed the case.

If this is not the case, then $\epsilon = 0$ and so, $\mathsf{msb}(s) = \mathsf{msb}(a_3) \oplus \mathsf{msb}(a_3) = 0 \ldots 000_2$. But then, $\mathsf{msb}(s \wedge a_i) = \mathsf{msb}(s) \wedge \mathsf{msb}(a_i) = 0 \ldots 000_2$. Also, the 32 most significant bits of M are all ones and hence, $\mathsf{msb}(x \wedge M) = \mathsf{msb}(x)$ for all x. Hence, with respect to the 32 most significant bits, Equation (8) reduces to

$$T\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_0 \oplus x_1^2 \\ x_1 \oplus x_2^2 \\ x_2 \oplus x_3^2 \\ x_3 \oplus x_0^2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{9}$$

with $\mathsf{msb}(x_i)$ and $\mathsf{msb}(y_i)$ known for $0 \leq i < 4$. Let $z_i = \mathsf{msb}(x_{i-1}) \oplus \mathsf{msb}(y_{i-1})$, where the computation on the subscripts is done modulo 4. Then z_i, $0 \leq i \leq 3$ are known and we need to solve for (x_0, x_1, x_2, x_3) such that the following equation holds for $i = 0, 1, 2, 3$.

$$\mathsf{msb}\left(x_i^2 \bmod 2^{64}\right) = z_i \tag{10}$$

We use Algorithm 2 to solve these four equations. Solving each equation takes time $5 \times 2^{16} < 2^{19}$ (see Case 1 of Section 3) and hence the time to solve all four equations is less than 2^{21}. The possible solutions for $\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2)$ and $\mathsf{lsb}(x_3)$ are kept in four lists L_0, L_1, L_2 and L_3 respectively. Since $n = 64 = 2 \times 32 = 2m$, the expected number of entries in each list is one. We then form a list Fin which contains tuples $(\mathsf{lsb}(x_0), \mathsf{lsb}(x_1), \mathsf{lsb}(x_2), \mathsf{lsb}(x_3))$ such that $\mathsf{lsb}(x_i)$ is in L_i. Then for each entry in Fin, we verify the solution by evolving the T–function in the forward direction a few times and comparing the output with the already available pseudo-random bits.

Thus, we get an algorithm to determine the 128 unknown bits in the input of Equation (8). It is easy to verify that the entire attack takes 2^{21} time. This shows that the T-function based on squaring is completely insecure as a stand-alone pseudo-random generator.

4.3 Extracting Lesser Bits

In this subsection, we use the notation $\mathsf{msb}_k(x)$ (resp. $\mathsf{lsb}_k(x)$) to denote the k most (resp. least) significant bits of x. The state vector for the T–function in (5) is (x_0, x_1, x_2, x_3). Suppose that instead of producing 128-bit output only the 64 bits $(\mathsf{msb}_{16}(x_0), \mathsf{msb}_{16}(x_1), \mathsf{msb}_{16}(x_2), \mathsf{msb}_{16}(x_3))$ are produced as output. Thus there are 192 unknown bits in (x_0, x_1, x_2, x_3) which have to be determined. We consider the effectiveness of our attack for this situation. The attack described in Section 4.1 goes through for this case.

The complexity of the total attack depends on the complexity of solving Equation 7. We use Subcase 1b of Section 2.1 along with the modification described in Step 2 of Section 4.1. Then the run-time of the modified Algorithm 1 becomes $2^{16-4}(2^{16-1} + 3 \times 2^{32-4} \times 2^{33}) = 2^{27} + 3 \times 2^{73}$. The number of unknown bits in (7) is $2 \times (48 - 8) + 2 \times 48 = 176$. The number of constraints in (7) is $4 \times 16 = 64$. Hence the expected number of solutions in Fin is 2^{112}. The correct solution can be determined by iterating the T–function and comparing the output with the available pseudo-random string. Thus the time taken to determine the 192 unknown bits will be 2^{112}. Though this is infeasible in practice, it still constitutes a theoretical attack on the system.

5 Implementation

We have performed some experiments to verify some of the assumptions about the average case behaviour. In this section, we briefly describe these results.

The first thing to consider is the expected number of solutions to Mult. As mentioned in Section 2, the expected number of solutions is 2^{2n-3m}. We describe

some experimental results for $n = 16$ and $m = 8$. The expected number of solutions is $2^8 = 256$. The total number of possible instances $(x^{(1)}, y^{(1)}, z^{(1)})$ is 2^{24}. The number of instances such that the number of solutions is at most 256 is around 55% of the total number of instances while the number of instances such that the number of solutions is at most 512 is more than 99 %. The maximum number of solutions occurs for the case $(x^{(1)}, y^{(1)}, z^{(1)}) = (0, 0, 0)$ and such pathological situations are extremely rare.

We have implemented the attack on a reduced version of the multiplication based T–function described in Section 4.1. We have chosen the state vector to be four 32-bit words instead of four 64-bit words. Correspondingly, we have extracted the top 16 bits of each word. Also the constants C_1 and C_3 have been suitably scaled down. The attack has been implemented on randomly chosen instances and in each case the size of Fin was found to be one. This provided 2^8 choices for the state vector and the unique one could be found using only one more block of the available pseudo-random bit string. Thus the attack worked extremely well with only three consecutive blocks of output. We expect the attack to scale up quite well when applied to the T–function having state vector consisting of four 64-bit words.

For the problem on squaring, we have implemented Algorithm 2. In the case $n = 64$ and $m = 32$, the complexity of Algorithm 2 is 5×2^{16}. Our experiments confirm this theoretical result and hence the attack on the squaring based T–function in Section 4.2 works as expected.

6 Possible Countermeasure

Our attacks show that T–functions are probably not secure enough for standalone use especially when half of the bits of the state vector are produced as output. As suggested by Klimov and Shamir, T–functions can be used in conjunction with S-boxes for design of stream ciphers. We provide one suggestion for possibly improving the security of T–functions while retaining some of the nice theoretical properties.

There is a large and easily identifiable subclass $\mathcal{C}$ of T–functions such that any function in $\mathcal{C}$ defines a single cycle permutation on the state space. This is an attractive theoretical property. In our suggestion, we would like to preserve this property. To do this we apply the notion of conjugate permutations. (A similar idea has been used in the context of one-way permutations [5].) If π and τ are any two permutations of a set S, then $\sigma = \tau^{-1} \circ \pi \circ \tau$ has the same cycle structure as π; further, σ and π are called conjugate permutations.

We apply it to the context of T–functions in the following manner. Suppose π is the permutation on the set of all state vectors induced by a T–function from $\mathcal{C}$. Then π has a single cycle and any conjugate of π also has a single cycle. Note that this property does not depend on the choice of the permutation τ. Hence we can choose τ so as to improve the security of the overall mapping.

In our attack, the basic weakness that we exploit is that there is insufficient intermixing of higher and lower bits. One simple operation which can help in

improving such intermixing is the circular shift (which is not a T–function). Thus we can construct a permutation τ on the state space by using circular shifts and other nonlinear operations. These operations can be arbitrarily chosen (in particular they need not be T–functions) to ensure higher security as long as τ is a permutation and that they are efficient to apply.

One penalty for introducing this countermeasure will be reduction in speed. The exact amount of speed reduction will depend on the concrete proposal. Developing such a concrete proposal based on our guideline is a future research problem.

7 Conclusion

In this paper, we studied multiplication, squaring and T–functions. In the first part of the paper, we presented a time-memory trade-off algorithm to solve the problems of multiplication and squaring. These algorithms are used in the second part of the paper to analyse two concrete proposal of multi-word T–functions from [3]. For the proposal based on multiplication, the 128 unknown bits of the state vector can be determined in 2^{40} time while for the proposal based on squaring, these bits can be determined in 2^{21} time. Experimental results from our implementation suggests that our attack works well in practice. Hence one can conclude that these two (and other similar) constructions of T–functions are not secure enough for stand-alone use. We also suggest the use of conjugate permutations for possibly improving the security of T–functions while maintaining some nice theoretical properties.

Notes: An anonymous reviewer of the paper has suggested that the problems Mult and Sqr can be formulated as closest vector problems in a two-dimensional lattice. Using this approach, the time complexity of Algorithm 1 will be 2^{32} with minimal storage space. At the time of preparing this final version, we have not been able to obtain the details of such an algorithm. We hope to present such details in a later communication.

References

1. A. Klimov and A. Shamir. A New Class of Invertible Mappings, *Proceedings of CHES 2002*, LNCS, 2002, pp 470–483.
2. A. Klimov and A. Shamir. Cryptographic Applications of T–functions, *Proceedings of SAC 2003*, LNCS.
3. A. Klimov and A. Shamir. New Cryptographic Primitives Based on Multiword T–functions, *Proceedings of FSE 2004*, LNCS, to appear.
4. A. Menezes, P. C. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1997.
5. M. Naor, O. Reingold. Constructing Pseudo-Random Permutations with a Prescribed Structure, *Journal of Cryptology*, 15(2): 97-102 (2002).