T-function based streamcipher TSC-3

Jin Hong, Dong Hoon Lee, Yongjin Yeom, Daewan Han, and Seongtaek Chee

National Security Research Institute
161 Gajeong-dong, Yuseong-gu
Daejeon, 305-350, Korea
{jinhong,dlee,yjyeom,dwh,chee}@etri.re.kr

Abstract. We describe a streamcipher based on T-functions. The cipher is intended for 80-bit security and is suitable for constrained hardware environments.

1 Introduction

This note presents a synchronous streamcipher named TSC-3, together with some security and implementation analysis. The main target environment of the cipher is constrained hardware with intended security level of 80 bits.

TSC-3 is a filter generator based on a T-function. The T-function was designed to allow a wide range of hardware implementation choices. Using 4×4 s-boxes at its core, the design leaves open the possibility for implementations of very low power consumption. In an attempt to minimize size, we have used 160-bit internal state, which is the minimum reachable by ciphers intended for 80-bit security.

The cipher TSC-3 is very scalable. Should it be desirable, we can easily accommodate our design to provide for security levels of up to 128 bits and as low as 64 bits.

This note is organized as follows. We start by describing TSC-3 itself. Next, implementation aspects are considered, followed by some security analysis. At the end, we give supplementary information useful for a better understanding of the cipher.

No hidden weakness No hidden weaknesses were inserted in TSC-3 by the designers.

2 Cipher specification

We give specifications of TSC-3 in this section. In short, it is a filter generator based on a T-function. The internal state of 160 bits is updated through the action of an operator named the T-function. This structure replaces the LFSR of a classical filter generator. After each update, a 32-bit output keystream is produced from the state through the use of a nonlinear filter.

Explanation of the internal state and notation will be given first. The cipher body is treated next. Finally, after touching on key and IV usage issues, the rekeying process that utilizes the cipher body is explained.

2.1 Internal state

The internal state consists of four words, each of 40-bit size, for a total of 160-bit state. The words will be denoted by x_k $(k = 0, 1, 2, 3)$ and the whole state will be written as $\mathbf{x} = (x_k)_{k=0}^{3}$, using the boldface character.

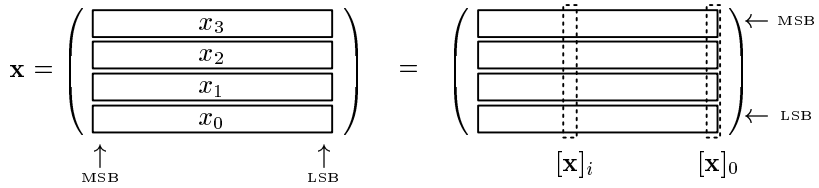
The i-th bit of any word y will be denoted by $[y]_i$, where we always count the indices starting from 0. We will sometimes view a given n-bit word y as an integer, in which case, we are reading it as

$$\text{word } y \text{ as an integer} = \sum_{i=0}^{n-1} [y]_i 2^i. \tag{1}$$

For any multi-word $\mathbf{y} = (y_k)_{k=0}^{m-1}$, the m-many i-th bits are collectively denoted as $[\mathbf{y}]_i$. We sometimes view $[\mathbf{y}]_i$ also as an m-bit integer by setting

$$[\mathbf{y}]_i \text{ as an integer} = \sum_{k=0}^{m-1} [y_k]_i 2^k. \tag{2}$$

In reading the rest of this submission, it will help to visualize the internal state and notation pictorially as follows.



Accordingly, we shall refer to $[\mathbf{x}]_i$ as the i-th column of state $\mathbf{x}$.

2.2 Main body

Operation of the cipher body is given in Algorithm 1. But, some explanation is needed to understand various parts of it.

Parameters We first define some temporary variables that change according to the contents of state $\mathbf{x}$.

$$\begin{aligned}
\pi(\mathbf{x}) &= x_0 \wedge x_1 \wedge x_2 \wedge x_3 \\
o(\mathbf{x}) &= \pi(\mathbf{x}) \oplus (\pi(\mathbf{x}) + \texttt{0x4910891089}) \\
e_1(\mathbf{x}) &= (x_0 + x_1)_{\ll 1} \oplus (x_2 + x_3)_{\ll 8} \\
e_0(\mathbf{x}) &= (x_0 + x_1)_{\ll 8} \oplus (x_2 + x_3)_{\ll 1} \\
p_1(\mathbf{x}) &= o(\mathbf{x}) \oplus e_1(\mathbf{x}) \\
p_0(\mathbf{x}) &= o(\mathbf{x}) \oplus e_0(\mathbf{x})
\end{aligned} \tag{3}$$

Algorithm 1 Cipher body

while more keystream is needed **do**
  # parameter
  compute $\mathbf{p}(\mathbf{x})$
  $\mathbf{tmp} \leftarrow \mathbf{p}(\mathbf{x})$
  # s-box application
  for $i = 0$ to $i = 39$ **do**
    if $[\mathbf{tmp}]_i = 3$ **then**
      $[\mathbf{x}]_i \leftarrow S([\mathbf{x}]_i)$
    else if $[\mathbf{tmp}]_i = 2$ **then**
      $[\mathbf{x}]_i \leftarrow S^2([\mathbf{x}]_i)$
    else if $[\mathbf{tmp}]_i = 1$ **then**
      $[\mathbf{x}]_i \leftarrow S^5([\mathbf{x}]_i)$
    else if $[\mathbf{tmp}]_i = 0$ **then**
      $[\mathbf{x}]_i \leftarrow S^6([\mathbf{x}]_i)$
    end if
  end for
  # filter
  for $k = 0$ to 3 **do**
    $y_k \leftarrow (x_k)_{\gg 8}$
  end for
  if $[x_0]_0 = 1$ **then**
    swap contents of y_0 and y_1
  end if
  if $[x_2]_0 = 1$ **then**
    swap contents of y_2 and y_3
  end if
  if $[x_1]_0 = 1$ **then**
    swap contents of y_1 and y_2
  end if
  if $[x_3]_0 = 1$ **then**
    swap contents of y_0 and y_3
  end if
  output $f(\mathbf{y})$ as keystream
end while

Here, notation $\wedge$ denotes bitwise `AND` operation of 40-bit words, $\oplus$ denotes bitwise `XOR` operation, and $\ll$ denotes left shift, that is, towards the significant bits. The additions are done modulo 2^{40}, where each word is viewed as an integer through (1). Hence all the above are 40-bit word values. Readers familiar with T-function theory will notice that o is an *odd parameter* and that e_k are *even parameters*. Using these, we define a multi-word parameter.

$$\mathbf{p}(\mathbf{x}) = (p_k(\mathbf{x}))_{k=0}^{1}. \tag{4}$$

Note that, through (2), we can view each column value

$$[\mathbf{p}(\mathbf{x})]_i = 2 \cdot [p_1(\mathbf{x})]_i + [p_0(\mathbf{x})]_i \tag{5}$$

as an integer between 0 and 3.

S-box application The state update function, denoted by $\mathbf{T}$, will be a *T-function*. We fix a 4×4 s-box S, given here in the C-language style.

```
S[16] = {3,5,9,13,1,6,11,15,4,0,8,14,10,7,2,12};
```
(6)

The action of T-function $\mathbf{T}$ on internal state $\mathbf{x}$ is defined through the following formula.

$$[\mathbf{T}(\mathbf{x})]_i = \begin{cases} S\ ([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 3, \\ S^2([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 2, \\ S^5([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 1, \\ S^6([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 0. \end{cases} \tag{7}$$

Here, the columns $[\mathbf{x}]_i$ and $[\mathbf{T}(\mathbf{x})]_i$ are treated as 4-bit integers through (2), so it makes sense to view them as input to and output from s-boxes. In plain terms, operator $\mathbf{T}$ applies different powers of the s-box to each column, depending on the value of $\mathbf{p}$ at the corresponding position.

Nonlinear filter The filter produces the actual output keystream from the current internal state. Take four temporary variables, each of 32-bit size, which we denote as $\mathbf{y} = (y_k)_{k=0}^{3}$. The first step of the filter initializes $\mathbf{y}$ with the 32 significant columns of $\mathbf{x}$.

In the second step, the order of the four words in $\mathbf{y}$ is mixed depending on the least significant state column $[\mathbf{x}]_0$. Word y_0 is swaped with y_1 if $[x_0]_0 = 1$, and left as is if otherwise. Likewise y_2 and y_3 are swapped if $[x_2]_0 = 1$. Next, bits $[x_1]_0$ and $[x_3]_0$ control the swapping of $y_1 \leftrightarrow y_2$ and $y_0 \leftrightarrow y_3$, respectively. Notice that the application order of these swapping operations do matter.

The last step calculates and outputs the 32-bit output

$$f(\mathbf{y}) = \big((y_0)_{\lll 7} + (y_1)_{\ggg 2}\big)_{\lll 8} + \big((y_2)_{\lll 7} + y_3\big)_{\ggg 9}. \tag{8}$$

Here, the additions are done modulo 2^{32} and the notation $\lll$ and $\ggg$ signifies rotations to the left and right, respectively.

Summary The cipher consists of two parts, the T-function and the nonlinear filter. At each clock tick, the T-function updates the 160-bit internal state and the nonlinear filter produces 32-bit output from the updated state. The actual encryption proceeds as usual by XORing keystream to plaintext. The LSB of the 32-bit word output should be used for encrypting the first plaintext bit.

The T-function part updates the internal state by applying various powers of an s-box to each column of state depending on some parameter values calculated from the current state.

2.3 Key size, IV size, and IV update

Supported and recommended sizes for key and IV are given. We also give some guidelines on the usage of IVs.

Recommended key size The following key and IV size pairs are recommended for 80-bit security.

1. 160-bit key with constant zero IV.
2. 128-bit key with 32-bit IV.
3. 96-bit key with 64-bit IV.

Should there be absolute need for 80-bit keys, use of 96-bit IVs is recommended. The structure of the rekeying process shall be so that the outcome do not depend on the length of IVs set to zero.

Supported key size The following sizes for key and IV are supported.

1. All key sizes between 80 and 160 bits, inclusive, may be used.
2. All IV sizes between 1 and 128 bits, inclusive, may be used. An IV should always be used, even if it is to be set to constant zero.

Under this specification, 40-bit, 80-bit, and 120-bit IVs are supported, but their use is discouraged.

IV usage guideline Irrespective of how strong a streamcipher is, there are guidelines on IV usage one should follow for secure use of streamciphers. For example, care is usually taken by system designers so that the same (key, IV) pair is never used more than once.

Recently, a work on time-memory tradeoffs[10] has appeared that has implications on the use of IVs. Even though these implications do not depend on the specific cipher in consideration, as the work is not yet widely known, as a precaution, we discuss this issue here in connection with our cipher.

In addition to the common sense of avoiding reuse of any (key, IV) pair, the use of key and IV with TSC-3 should adhered to the following rules in order to meet 80-bit security.

1. In many current implementations of streamciphers, the next IV to be used is frequently produced from the current IV through a deterministic process. Counter-supplied IVs is one common example. When such *incremental* update of IV is utilized with keys of k-bit size, the initial IV, i.e., the starting point of increment, should be chosen in a manner that provides for at least $(160 - k)$-bit entropy.
2. The above rule also applies to the degenerate case of only one IV being used per key agreement. That is, for k-bit keys, the unique IV should be chosen in a manner that provides at least $(160 - k)$-bit entropy.
3. The incremental update of IV should never continue for more than a certain *preset time period,* without a new random starting point being deployed. This rule applies even if extremely small amount of data was encrypted during one such period.

Put in simple terms, the use of IV should be so that, together with key, 160-bit entropy is reached from the view point of a time-memory tradeoff attacker that is trying to prepare tables for future use. The actual choice of *preset time period* should be made from this perspective and one year seems to be a reasonable choice for now.

For k-bit keys, these guidelines suggest the use of at least $(160 - k)$-bit size IVs. At the extreme of 160-bit keys, any use of IV, even setting it to constant, is allowed.

Choosing a new random starting point on each key agreement does not seem to be absolutely necessary, although this would be done in most implementations. As long as the continued incremental update of IV is restricted to an appropriate time period, it seems safe to pass on the incremented IV from one key to the next key.

But setting up the random IV starting point during key negotiation does seem most natural. Since at least 160 bits need to be agreed upon during such a key agreement, we see no reason to insist on usage of 80-bit keys. Rather it seems better to select the key size based on how long an IV is needed. This is the approach taken in our key and IV size recommendations.

For a fixed key, with the repeated adoption of new random IVs, comes the risk of using the same IV more than once. Compared to the intended security level, this risk is quite large when small IVs are used. This trivial observation seems to be one point that was not seriously considered before, probably because there was no need for random IVs until now. To be safe, for short IVs, it might be necessary to restrict new random IV starting point deployments to at most once per new key agreement.

Recommended IV usage example During the key negotiation phase, the two parties agree upon a 160-bit or longer (partially) secret random value. A segment of appropriate length is set aside as the first IV and the rest is used as key. The IV part need not be kept secret. Subsequent IVs are produced by incrementing

the current IV like a counter.[1] New key negotiation is done before the current key ages over the preset time period.

Comments The submitters are aware that the approach of these guidelines is quite different from what has been in wide practice. But we believe a full understanding of paper [10] will convince readers that such increase in key and IV size is inevitable. This reason for key and IV size increase applies equally well to all streamciphers and even to various modes of operation for blockciphers.

Still, we will be happy to change our key and IV size recommendations to follow whatever consensus is reached through future discussions on the work [10]. These guidelines are based on the new time-memory tradeoff approach considerations, rather than on the structure of TSC-3.

2.4 State initialization

Let us now explain how the state is initialized from given key and IV.

First, the key is simply padded into the state multiple times until the state is fully filled. One starts by filling the LSB of state word x_0 with the LSB of key. This will fill x_0 with the 40 less significant bits of the key. Then one continue onto word x_1, and so on. When the MSB of key is reached, one continues padding the state with another copy of the key.

For 80-bit keys, two copies of the key would be used. For 160-bit keys, a single copy would suffice to cover the entire state. For keys of intermediate lengths, a full copy and the less significant part of another copy would be used.

After the key has taken its place inside the state, the IV is mixed into the state through a slightly more complex process.

1. `XOR` IV onto state multiple times until state is fully covered; As with the key, one starts by filling the LSB of state word x_0 with the LSB of IV, continues onto state word x_1, and so on.
2. Run cipher body once to produce a single 32-bit output; That is, calculate parameters, apply s-boxes, and calculate output from updated state.
3. `XOR` output onto state multiple times until fully covered; As before, we start by `XOR`ing output to the 32 less significant bits of the 40-bit state word x_0. Another copy of the output is prepared and its 8 less significant bits are `XOR`ed to the remaining 8 more significant bits of x_0. Then, what remains of the output is used on x_1, and so on.
4. Repeat the above three steps two more times, for a total of three applications of the T-function.

This completes the key and IV setup.

[1] The use of incremental update was influenced by an observation in [6].

3 Structure related to implementation

The s-box and its powers we are using were chosen with various implementation considerations. Some of these will be explained here. Readers may refer to Section 7.5 for additional information concerning the s-box.

3.1 Regrouping s-box exponents

The application of various powers of s-box to each state column can be broken into two steps. Let us recall Algorithm 1 and write

$$\mathbf{tmp} = (\mathrm{tmp}_k)_{k=0}^{1} \tag{9}$$

so that $\mathbf{tmp} \leftarrow \mathbf{p}(\mathbf{x})$ stores each calculated 40-bit value $p_k(\mathbf{x})$ in temporary variable tmp_k. The lines 6 to 16 of Algorithm 1 can be replaced with the lines given in Algorithm 2, allowing for more implementation choices.

Algorithm 2 Two-step s-box application

```
 1: for i = 0 to i = 39 do
 2:    if [tmp_1]_i = 1 then
 3:       [x]_i ← S([x]_i)
 4:    else if [tmp_1]_i = 0 then
 5:       [x]_i ← S^5([x]_i)
 6:    end if
 7: end for
 8: for i = 0 to i = 39 do
 9:    if [tmp_0]_i = 1 then
10:       [x]_i ← [x]_i
11:    else if [tmp_0]_i = 0 then
12:       [x]_i ← S([x]_i)
13:    end if
14: end for
```

Basically, the new way of presenting the algorithm allows a two-stage application of s-boxes, with each stage governed by one of the two words in $\mathbf{p}(\mathbf{x})$. Notice that the two for-loops may be combined if desirable.

3.2 Boolean expression for s-boxes

The s-box was chosen so that the output bits of various powers of the s-box may be written as simple boolean functions of the input bits. Explicitly, if we let

$$(8 \cdot t_3 + 4 \cdot t_2 + 2 \cdot t_1 + t_0) \mapsto (8 \cdot s_3 + 4 \cdot s_2 + 2 \cdot s_1 + s_0)$$

denote the action of some power of the 4×4 s-box, the output bits s_k may be written as a function of input bits t_k in the following manner.

$$S \begin{cases} s_3 = t_1 \oplus (t_3 \wedge t_2 \wedge \bar{t}_0) \\ s_2 = t_0 \oplus (t_3 \wedge \bar{t}_2 \wedge \bar{t}_1) \\ s_1 = t_2 \oplus (t_3 \wedge t_1 \wedge t_0) \oplus (\bar{t}_3 \wedge \bar{t}_1 \wedge \bar{t}_0) \\ s_0 = \bar{t}_3 \oplus (t_2 \wedge \bar{t}_1 \wedge t_0) \end{cases} \tag{10}$$

$$S^2 \begin{cases} s_3 = t_2 \oplus (\bar{t}_3 \wedge \bar{t}_1 \wedge \bar{t}_0) \\ s_2 = \bar{t}_3 \oplus (t_2 \wedge \bar{t}_1 \wedge t_0) \oplus (\bar{t}_2 \wedge t_1 \wedge \bar{t}_0) \\ s_1 = t_0 \oplus (\bar{t}_3 \wedge t_2 \wedge t_1) \\ s_0 = \bar{t}_1 \oplus (t_3 \wedge t_2 \wedge \bar{t}_0) \oplus (\bar{t}_3 \wedge \bar{t}_2 \wedge t_0) \end{cases} \tag{11}$$

$$S^5 \begin{cases} s_3 = \bar{t}_1 \oplus (t_3 \wedge t_2 \wedge \bar{t}_0) \oplus (\bar{t}_3 \wedge \bar{t}_2 \wedge t_0) \\ s_2 = \bar{t}_0 \oplus (t_3 \wedge \bar{t}_2 \wedge \bar{t}_1) \oplus (\bar{t}_3 \wedge t_2 \wedge t_1) \\ s_1 = \bar{t}_2 \oplus (\bar{t}_3 \wedge \bar{t}_1 \wedge \bar{t}_0) \\ s_0 = t_3 \oplus (t_2 \wedge \bar{t}_1 \wedge t_0) \oplus (\bar{t}_2 \wedge t_1 \wedge \bar{t}_0) \end{cases} \tag{12}$$

$$S^6 \begin{cases} s_3 = \bar{t}_2 \\ s_2 = t_3 \oplus (\bar{t}_2 \wedge t_1 \wedge \bar{t}_0) \\ s_1 = \bar{t}_0 \\ s_0 = t_1 \oplus (\bar{t}_3 \wedge \bar{t}_2 \wedge t_0) \end{cases} \tag{13}$$

Here, the bar notation denotes bit complement. Notice that many of the cubic terms overlap, so that there is room for optimization when these four s-boxes are realized together.

4 Hardware implementation

In this section, we consider the hardware implementation aspects of TSC-3. A wide range of implementation choices, from very fast to very small, are possible. We shall focus on the two extremes.

4.1 Fast implementation

Let us first consider a fast straightforward implementation of TSC-3 under abundant resources.

Parameter The critical path for calculation of parameter o is two `AND` and one 40-bit addition. Critical path for parameter e_k is one 39-bit addition and one `XOR`. As these two parameters may be calculated simultaneously, and since `AND` and `XOR` operations take negligible time in comparison to additions, we can say that the time taken for calculation of multi-word parameter $\mathbf{p}(\mathbf{x})$ is a little over single 40-bit adder delay.

T-function We've already seen in Section 3.2 that all the s-boxes can be calculated within a depth of few logical operations. This will be a lot shorter than the times taken for one 40-bit addition. So we can have all four s-box values pre-computed for each column, this done for all columns simultaneously, while $\mathbf{p}(\mathbf{x})$ is being computed. When both are ready, one of the four s-box values may be selected through a multiplexer controlled by values of $\mathbf{p}(\mathbf{x})$ at each column, once again, simultaneously on every column.

Hence the time taken for the whole T-function operation, i.e., parameter calculation and s-box application, is little over time taken by $\mathbf{p}(\mathbf{x})$, which is about a single 40-bit adder delay.

Nonlinear filter As the time taken for $\mathbf{y}$ mixing, done with multiplexers controlled by $[\mathbf{x}]_0$, is negligible compared to 32-bit additions, one can easily see that the time taken for one filter output calculation is a little over twice a 32-bit adder delay.

Cipher body speed Notice that the T-function and filter may run in parallel, after the first invocation of $\mathbf{T}$. The filter is slower than the T-function and hence the speed of TSC-3 will be about

$$(32 \text{ bits})/(2 \cdot (32\text{-bit adder delay}) + \varepsilon). \tag{14}$$

For example, in an ASIC implementation that uses 32-bit adders of modest delay time 0.4ns, the cipher will run at somewhere around 32 Gbps.

State initialization speed The main part of state initialization from key and IV is three invocations of the cipher body, and for the first output keystream, one more invocation is needed. We can expect this delay to be almost nonexistent for a fast implementation.

4.2 Power-efficient implementation

This section discusses implementations of TSC-3 suitable for constrained hardware environments.

T-function: going down In line 4 of Algorithm 1, we had used a temporary variable **tmp**, because the value $\mathbf{p}(\mathbf{x})$ changes with the action of s-boxes on $\mathbf{x}$. We wanted to make sure that value of $\mathbf{p}(\mathbf{x})$ calculated *before* any s-box application was used in deciding which s-box power to apply.

But, through direct checking (or basic T-function theory), one can show that parameter value $\mathbf{p}(\mathbf{x})$ at a fixed column does not depend on the higher column inputs. That is, $[\mathbf{p}(\mathbf{x})]_i$ does not depend on any of the input columns $[\mathbf{x}]_j$ with $j \geq i$. So if we apply the s-boxes starting from the most significant column, there is no reason the store the parameter values. That is, if we replace line 6 of Algorithm 1 by

for $i = 39$ to $i = 0$ **do**

there is no need for the temporary variable **tmp**.

Suppose ripple carry adders were used to implement $\mathbf{p}(\mathbf{x})$. When an s-box is applied to $[\mathbf{x}]_i$, the values $[\mathbf{p}(\mathbf{x})]_j$ for $j > i$ will change after some delay, but the next target $[\mathbf{p}(\mathbf{x})]_{i-1}$ will be unaffected, so we can proceed to lower columns without any waiting. So, in principle,

1. 160-bit storage space for the cipher internal state,
2. three 40-bit adders and $40 \cdot$ (5 `XOR` & 3 `AND`) to implement parameters, and
3. one s-box to be reused on each column for various s-box powers

is all that is needed to implement the T-function. In practice, the control logic and interconnection between various parts of the cipher will not be negligible and we cannot determine exactly how much resources are needed.

As for the low power consumption requirement, s-boxes do not present a problem, as they are small and applied to each column in turn. It suffices to choose appropriate lower power implementation for adders.

T-function: going up Notice that to calculate parameter value $[o(\mathbf{x})]_i$ at the i-th column, it suffices to know a single carry bit that was generated during the calculation of parameter at the lower column $[o(\mathbf{x})]_{i-1}$. Then, from the i-th state column $[\mathbf{x}]_i$, one can generate and pass on data needed for calculation of the next column $[o(\mathbf{x})]_{i+1}$.

It suffices to keep constant track of 1-bit *information summary* of lower state columns to be able to calculate $o(\mathbf{x})$ at a column and pass on data needed for all subsequent higher column calculations.

Similarly, if one has access to appropriate 8 bits of (x_0+x_1), 8 bits of (x_2+x_3), and two carry bits, from the additional information of $\mathbf{x}$ at a single column, one can calculate the two $e_k(\mathbf{x})$ values corresponding to that column, and then generate and pass on data needed for sequential calculation of $e_k(\mathbf{x})$ at all higher columns.

In all, it suffices to store 19 bits and pass them onto the next significant column in order for the parameter to be calculated sequentially. In the language of [5], this means that our T-function is 19-narrow. We can destroy the content of $[\mathbf{x}]_i$ by applying s-box powers, without worrying about obtaining correct $[\mathbf{p}(\mathbf{x})]_{i+1}$ value, if we only store 19 appropriate additional bits.

With a very small amount of extra storage space, is we may move *up* the columns of state, applying s-boxes. Compared to the going-down approach, this method has the advantage of allowing parameters to be calculated as we proceed.[2] If slightly more storage is available, as long as the power requirements are met, one could try applying s-boxes to several columns at once, being careful to keep track of more data for later parameter computation.

[2] The mapping $\mathbf{T}^{-1}$ has the nice property that no extra storage is needed for its sequential calculation while retaining this advantage. This mapping could be worth studying.

Filter The words $\mathbf{y}$ need not occupy new storage space. Mixing of words in $\mathbf{y}$ can be done through two layers of 2 to 1 multiplexers. To keep the power consumption low, the three additions of (8) should be done one at a time and with proper choice of adders.

If bit-serial adders are used for all three, small amount of additional storage is needed. One would store $(y_0)_{\lll 7} + (y_1)_{\ggg 2}$ and maybe also the least significant 9 bits from $(y_2)_{\lll 7} + y_3$ temporarily.

4.3 Comments

We have not yet done any synthesis of these implementation techniques, but plan to do so in the future. It is not clear at this stage as to whether the above ideas will produce a cipher implementation of reasonable speed and of very small gate complexity.

We would like to compare the speed of a small implementation of TSC-3 with that of the current smallest AES realization[7]. Since we do not know the gate complexity of any TSC-3 implementation, the comparison could be unfair and misleading, but we do this to argue that small implementation of TSC-3 should be of reasonable speed. Considering the fact that AES produces 128-bit output in 10 rounds, we compare TSC-3, which produces 32-bit output, against 2.5 rounds of AES.

Let us first look at s-box applications. In each round of AES, an 8×8 s-box is applied 16 time. Translated to 2.5 rounds, this is 40 s-box applications. Our cipher does 40 applications of smaller 4×4 s-boxes. So if all four s-boxes are implemented, TSC-3 should be faster, but if only one s-box is repeatedly applied for other s-box powers, AES should be faster. With the two-step approach of Section 3.1 the two should be comparable.

Parameter and filter calculation will be considered next. Ignoring the more simple logical operations, these two combined will cost roughly about eight (40-bit and 32-bit) word additions and the speed will depend heavily on the adder realizations. In any case, the two parts combined should be much faster than the 2.5 rounds of key schedule, which again take more than 40 8×8 s-box applications. Notice that the implementation of AES in consideration has to recalculate all round keys on its every invocation.

In all, we believe a small realization of TSC-3 should perform better than the small AES realization outlined in [7]. Also, as the minimal data bus width is of 8 bits for AES, while we can cope with 4-bit buses, TSC-3 is at an advantage with respect to low power consumption requirement.

With these crude comparisons set aside, let us recall that, in this section, we have seen two extreme ways, i.e., fast and small, to implement TSC-3 in hardware. There are many ways to come somewhere in between, such as applying s-box powers to not just one or all columns, but to several columns at once. Also, the structure discussed in Section 3.1 allows implementations that realize two s-box powers, instead of one or four.

We want to stress that TSC-3 allows for a very wide range of hardware implementation choices.

5 Software implementation

As TSC-3 deals with 40-bit words, it is not suited to 32-bit platforms. In our semi-optimized implementation of TSC-3, we used two 32-bit words to store a single 40-bit word. The lower 8 bits out of the 40 bits were stored in one word and the significant 32 bits were stored in another. This gives us easy access to the carry bit appearing during the sum of two lower 8 bits.

To implement the s-boxes, bit-slice technique for implementing small s-boxes, previously used on blockciphers, should be used. This approach is a lot faster than ripping out each column from the four words and reconstructing the words after s-box applications. The boolean functions needed for this bit-slice implementation is given in Section 3.2.

After applications of all four types of s-boxes are calculated through the bit-slice technique simultaneously on all columns, appropriate use of `AND` operation combining these with the parameters can be used to choose one of the four s-boxes.

The structure of our cipher given in Section 3.1 can also be utilized. The two-step approach calculates the s-box only three times, whereas four s-boxes are calculated in the straightforward approach, so some computation is saved, and this results in a slight improvement.

The mixing of $\mathbf{y}$ can either be done in a straightforward manner, or with precomputed table of permutated indices. Also, as only the single power of s-box is applied to the least significant column $[\mathbf{x}]_0$, with proper initialization, one can use a separate s-box updated counter in place of $[\mathbf{x}]_0$.

Our current C-language implementation of TSC-3 shows the following performance.

machine	Pentium-IV 3.2GHz, 2GB RAM
OS	Windows XP (SP2)
compiler	Microsoft Visual C++ 6.0
encryption	50 cycles/byte = 200 cycles/block
key setup	1500 cycles

The key initialization speed includes time for both key and IV setup.

As the 40-bit words are divided into two separate storage space, in calculating the application of s-boxes, exactly the same sequence of instructions were run on the two separate set of words. Note that this situation fits in naturally with the SIMD instruction set available on modern Pentium processors. The above results were obtained without the use of any SIMD instructions, so there is room for much improvement.

Even though our encryption/decryption speed is not at all impressive, it should still be fast enough for most applications. As our main target is low-end hardware, speed should not be an issue when a PC is communicating with such a device. Also, as our key setup time is comparable to or slightly faster than modern streamciphers, for short messages, the performance of our cipher should not be too bad.

6 Security

Cipher TSC-3 is intended for 80-bit security. For the moment, the best attack on TSC-3 we know of is the time-memory-data tradeoff of complexity 2^{80}.

6.1 Statistical tests

We have done tests similar to the ones presented in [1] and have verified that this proposal gives good statistical results.

6.2 Minimum period

The period of TSC-3 is 2^{160}. To understand the proof presented in this section, full knowledge of the basic T-function theory[14–16] is needed. As the proof is a straightforward extension of the proof for period appearing in [11], we shall only supply what more is needed.

Recall the definition of the multi-word parameter $\mathbf{p}(\mathbf{x})$ given by (3) and (4). Define another (single-word) parameter

$$p(\mathbf{x}) = p_0(\mathbf{x}) \wedge p_1(\mathbf{x}), \tag{15}$$

where $\wedge$ denotes logical `AND` of 40-bit words. We start by writing down a property of $p(\mathbf{x})$.

Lemma 1. *The parameter $p(\mathbf{x})$ is odd.*

Proof. Let us first write

$$\begin{aligned} p &= (o \oplus e_0) \wedge (o \oplus e_1) \\ &= o \oplus \big(o \wedge (e_0 \oplus e_1)\big) \oplus (e_0 \wedge e_1), \end{aligned}$$

where we have omitted writing out all dependence on state $\mathbf{x}$. It suffices to show the first term o to be odd, and the other two terms, $o \wedge (e_0 \oplus e_1)$ and $e_0 \wedge e_1$, to be even.

The first term o can be shown to be odd through application of [16, Lemma 1] and some clever counting. To show the second term to be even, it suffices to consider just

$$o(\mathbf{x}) \wedge \big((x_0 + x_1)_{\ll 1} \oplus (x_2 + x_3)_{\ll 1}\big).$$

But this is even due to the symmetry between the two sub-terms residing inside the larger parentheses. Symmetry arguments also show the last term to be even. In conclusion, $p(\mathbf{x})$ is an odd parameter.

Now, using this lemma, we analyze how often each case appearing in (7) is applied. Recall that each column value $[\mathbf{p}(\mathbf{x})]_i$ is a number between 0 and 3. For integers $0 \leq a \leq 3$, let us set up the temporary notation

$$n_i^a = \#\{\mathbf{x} \mid [\mathbf{p}(\mathbf{x})]_i = a \text{ and } [\mathbf{x}]_j = 0 \text{ for all } j \geq i\}. \tag{16}$$

Notice that since parameter $p_0(\mathbf{x})$ is odd, the number $n_i^1 + n_i^3$ is odd. On the other hand, Lemma 1 essentially tells us that n_i^3 is odd for every i. Hence n_i^1 is even and similarly n_i^2 is also even. The remaining number n_i^0 must be odd for the total sum to be even.

n_i^0	n_i^1	n_i^2	n_i^3
odd	even	even	odd

The rest of the proof showing period of TSC-3 to be 2^{160} is an easy extension of the arguments in [11].

6.3 Algebraic attack

In many cases, algebraic attacks are possible on streamciphers built on LFSRs. Once a single equation connecting the internal state to the output keystream is worked out, the cipher logic can be run forward to produce more such equations. During this process, the linear property of LFSRs keep the degree of new equations equal to the first equation. And this is the main reason for the success of algebraic attacks on streamciphers.

In the case of TSC-3, the source of randomness, i.e., the T-function, is already nonlinear. Hence algebraic attacks do not seem to be applicable.

6.4 Rotations in the filter

One of the main reasons for using rotations in the filter is to ensure that output from the same s-box, i.e., bits from the same column, do not contribute directly (as opposed to, through carry) to the same output bit. We want contributions to any single output bit to come from bits that change independently of each other. Other reasons will follow below. (See also Section 7.4.)

6.5 Guess-then-determine attack

One property of T-functions, that could be bad from the viewpoint of security, is that it can be restricted to any number of its lower columns. In other words a part of internal state can be guessed and run forward indefinitely, opening up the possibility of a guess-then-determine attack.

The rotations used in the filter eliminates this weakness. They have been chosen so that any single output bit receives direct effect of four bits that are spread widely apart within the state. So it is not possible to calculate any output bit with the information of any small number of lower column states.

Even if all modular additions in the filter were replaced with `XOR`s and even if we ignore the mixing of words in $\mathbf{y}$, in order to calculate any one of the 32 output bits continuously, one would need to guess at least 32 columns. This is equivalent to 128 bits, so no meaningful attack can be achieved through this approach.

6.6 Correlation attack

Difficulty of correlation attacks can also be obtained from the rotations in the filter. In the last step of a correlation attack, one needs to guess a part of the state and compare calculated outputs with the actual keystream, checking for the occurrence of expected correlation.

In our situation, any correlation found to exist with a single output bit will involve multiple input bits. Due to the rotations, at least one of them will come near the most significant column of state. This will force one to guess quite a large part of the state to be able to apply T-function even once.

Looking for correlation between a function of input bits and a function of output bits will also likely fail, because none of the input bits is used more than once in the filter. Trying to weave together more than one output bit will only result in having more input bits involved.

6.7 Bit-flip probability

We had chosen the s-box (6) to satisfy the following conditions.

1. At the application of S, each of the four bits has bit-flip probability of $\frac{1}{2}$.
2. The same if true for S^2, S^5, and S^6.

In more exact terms, the first condition states that for each $k = 0, 1, 2, 3$,

$$\#\{\, 0 \le t < 16 \mid \text{the } k\text{-th bit of } t \oplus S(t) \text{ is } 1\} = 8.$$

Due to this property, regardless of the behavior of the odd parameter $\mathbf{p}(\mathbf{x})$, at the action of $\mathbf{T}$, every bit in the state is guaranteed a $\frac{1}{2}$ bit-flip probability.

6.8 Bit-flip bias of multiple applications of T-function

There is a strong distinguishing attack[12, 17] applicable to a previous version[11] of this cipher. As the work describing this attack is currently under submission, we will be very brief in describing the defense introduced in the current version.

The main observation used in the attack is that even though the bit-flip probability of T-function is near $\frac{1}{2}$, this is not true for its multiple applications. This property of T-function is still present in the current design, but the bias has been reduced to some extent.

The word mixing applied to temporary variable $\mathbf{y}$ is our main defense against this property leaking out through the filter function. In a sense, we are using 16 different filters cyclically. So an attacker can only utilize bit-flip bias of $\mathbf{T}^n$ with n a multiple of 16.

We have plausible arguments that show the bit-flip bias of filter output corresponding to $\mathbf{T}^{16}$ to be much less than 2^{-50}. In order to detect this bias, data size of the order 2^{100} is needed. The distinguishing attack based on the bit-flip bias of $\mathbf{T}^{16}$ is thus no longer possible. For larger multiples of 16, we have reasons to believe the bias to be even smaller.

6.9 State initialization

We consider security issues related to key setup in this section. Our state retains almost full 160-bit entropy after state initialization, and no weak keys are present.

Entropy loss Let us consider the question of whether our state initialization process allows every possible 160-bit state to occur with equal possibility. This question is closely related to whether each step of the rekeying process is invertible. Checking all the steps of key setup and IV setup presented in Section 2.4, we can see that only Step 3 of the IV setup may be non-invertible.

It turns out that it is not bijective. We picked 500 random 160-bit states, and for each state, checked how many states would map to the chosen state under the mapping of Step 3. About 40% of the random states had a single inverse image, but about 30% had no inverse image, and the rest had more than one inverse image. The result is summarized in the following table.

# of inverses	0	1	2	3	4	≥ 5	total
# of states	166	194	99	33	8	0	500

Under a rough approximation, we can think of this mapping as being 2-to-1, and this gives us the feeling that we loose about 1-bit entropy.

Let us be more explicit, and first assume that this table reflects the general trend of the mapping given by Step 3. Notice that the number of pre-images covered in this table

$$523 = 0 \cdot 166 + 1 \cdot 194 + 2 \cdot 99 + 3 \cdot 33 + 4 \cdot 8$$

is slightly higher than 500. Normally, for any single point in the groups with, say, two inverse images, the probability of it being reached by mapping of Step 3 would be taken to be $\frac{2}{2^{160}}$. Here, we adjust this as $\frac{2}{2^{160}} \cdot \frac{500}{523}$ so that the sum of all probability becomes 1. Then, the entropy of image space can be calculated to be

$$H = -\sum p_i \log_2 p_i \cong 160 - 0.736,$$

where p_i denotes the probability of each one of the 2^{160} states being reached. We have thus confirmed our feeling that entropy of about one bit is lost.

Notice that, as Step 1 of IV setup regains the lost entropy, this entropy loss is not accumulated through the three rounds of IV setup. In fact, the lost entropy would only be of a few bits even if it were accumulated, as in the case of fixed IV use.

In a strict sense, the states produced through our key initialization process has less than 160-bit entropy. But as the difference is very small, we shall ignore this. Also, since we do not see any way to characterize the set of smaller entropy in a simple way, this does not seem to be of use in any case.

Key recovery from state Even though the arguments so far on entropy preservation focused on the good side of this property, in the course of discussion, we have shown that the state initialization is almost invertible. In particular, this mean that from a given state, if the IV is known, one may recover the master key with relatively small effort.

Equivalent keys The method for inverting the state initialization process shows one can easily find many equivalent (key, IV) pairs. Start with any key and IV, and go through the initialization process. Then invert just the IV setup process with a different IV chosen at random. If we were lucky, the final state reached would be the form of a key padded state.

Proper discussion of this process would show that if sum of key and IV size is greater than or somewhat near 160 bits, one should be able to find many equivalent (key, IV) pairs. That such pairs exist is no surprise. The point is that they can be calculated within reasonable time.

We do not view this as a threat to our cipher. In blockcipher theory, if equivalent keys can be identified, the key search space is effectively reduced and hence is a problem. But in our case, search space is reduced to only state size because IV also plays a role. This is nowhere close to our intended security level.

Statistical property For a good key initialization process, we would expect one bit difference in key or IV to result in about half the state bits changing. We did some basic experiments to verify this on our IV setup process.

Weak keys The single-cycle property of our T-function guarantees that no state analogous to the all-zero state of LFSRs is present in our cipher. But there is one case that should be considered in view of related IV attacks.

Our IV setup shows one unwelcome property when IVs of length that are multiples of 40 are used. For these IVs, unlike IVs of other lengths, difference of one bit is reflected through Step 1 of IV setup to only one column of state. So propagation of difference in IV into difference in state is slower with these IVs.

Based on our experiments, we believe it should still be safe to use these IVs, but would like to acknowledge that IVs of length 40, 80, and 120 bits are potentially weaker.

7 Informative section

This section deals with issues that would be of interest to anyone trying to understand some of the design choices. Also, information that would be of help in any future tweaks to this cipher is given.

7.1 Design objectives and rationale

Let us explain some of the basic approaches we took for development of our cipher.

Objectives As mentioned several times, TSC-3 was designed for use in constrained hardware environments.

Explicitly, we aimed to reduce gate complexity and power consumption. Even though our hardware speed is quite fast, this was one of the side-effects, not one of our major goals. We did work to increase software speed on moderate platforms, but no choice that would sacrifice hardware efficiency was made in favor of software speed. Low-end software platforms were not considered.

Target environments Let us defend our choice of objectives explained above.

In passive RFID tags, the information sent from the tag to the reader is very small. Hence the speed of encryption is not very important. But as the power used by the tag is received from the tag reader through radio signal, there is a limit to the power it can use. To be more precise, how much energy is consumed per unit time is more important than the total amount of energy needed per encryption. Of course, the common solution of reducing clock speed to meet power constraints also slows down the cipher, so power consumption is closely related to total energy need.

With semi-passive RFID tags, where calculations are done with an onboard battery and the data transmissions are done with power provided by the reader, or on any other devices powered by small batteries, the total energy would become slightly more important. But still, speed is a less important issue.

As for the gate complexity, in many cases, the space allotted to cryptographic application already limits one to only small implementations. Also, the static power dissipation due to leakage current (as opposed to the dynamic power required for charge and discharge of circuit nodes) is closely related to gate count, and adds to the list of reasons to look for small cipher implementations.

Since we are targeting hardware environments, the software speed that matter would be on those platforms that are communicating with small hardware devices. These will usually have better resources than the small hardware clients they are communicating with. Also, as the hardware devices are slow, there is no need for speed. Even moderate speed could support, a small server (software) communicating with many hardware devices simultaneously.

Cipher components The main reason for the choice of T-functions was with algebraic attacks. Steering clear of this threat without increasing LFSR size did not seem easy and we wanted to limit our state size to 160-bit, to keep gate complexity down. Irregular clocking is one possible defense against algebraic attacks, but we did not want to use this in view of side channel attacks which are of bigger threat on small devices.

The second reason for using T-functions was its potential for allowing some security analysis. That is, it is possible to work out the period of a well designed T-function. And during the development of TSC-3, we found that for many filters, one could calculate period of the T-function based filter generator from properties of the T-function.

In developing the T-function itself, we had excluded the use of multiplications. For software environments, multiplications give large randomizing effects at good speed, but in hardware environments, the cost of multipliers is prohibitively high. The adders are smaller in size and was more manageable with respect to security analysis, since it could be approximated with `XOR`s.

7.2 Varying the security level

As we are targeting constrained hardware environments, the security level of 80 bits should not be too small for the near future. But if a slight increase in security level is needed, it is not difficult to adjust our cipher.

It suffices to increase the word size of our cipher internal state $\mathbf{x}$ from 40 bits to, say, 64 bits to achieve 128-bit security. For the cipher body, the only part that needs retouching is the single constant used in calculation of parameter $o(\mathbf{x})$, which should be easy to deal with. Slight changes to key setup could also be needed, as the propagation of IV difference to state difference would have to be rechecked.

Beyond word size of 64 bits, we believe the ratio of row to column size to be inappropriate. The small inter-column mixing property of our cipher is potentially dangerous. A similar design that uses more words of shorter length should be better.

Reducing security level is equally easy. Using four words of 35-bit length as state $\mathbf{x}$ should give us 70-bit security.

With words of 32-bit length, the least significant column $[\mathbf{x}]_0$ is used both in determining $\mathbf{y}$ row mixing and also as a part of $\mathbf{y}$ it self. This does result in bias showing up on some of the input bits used in the final additions, but does not seem to cause serious trouble. So four 32-bit words can be used for 64-bit security.

7.3 Constant for parameter p

There is a single constant involved in the calculation of the parameter $\mathbf{p}(\mathbf{x})$. It is found in

$$o(\mathbf{x}) = \pi(\mathbf{x}) \oplus (\pi(\mathbf{x}) + \texttt{0x4910891089}). \tag{17}$$

To obtain the single-cycle property of our T-function, which is needed for arguments concerning cipher period, this has to be an odd number.

Written in the binary form, the constant is

$$\texttt{0100 1001 0001 0000 1000 1001 0001 0000 1000 1001}. \tag{18}$$

Notice that the number of 0s between two neighboring 1s is not uniform, being 2, 3, or 4. This irregularity helps the T-function to quickly move away from the state with all columns identical, when it is encountered. No other consideration was done in choosing this constant.

7.4 Filter rotation choices

We consider some more issues, other than those explained in Section 6, related to the choice of rotations used in the filter.

Non-uniform bit-flip probability In a very early version of our cipher which was similar to TSC-2[9,11], the T-function did not exhibit bit-flip probability close to $\frac{1}{2}$. Among the bits of state consisting of four 32-bit words $(w_k)_{k=0}^{3}$, only significant $\frac{2}{3}$ of the word w_0 changed with bit-flip probability close to $\frac{1}{2}$.

We had tried to hide this property through the nonlinear filter

$$\text{output} = ((((w_0 \lll 16) + w_0) \lll 7 + w_1) \lll 7 + w_2) \lll 7 + w_3). \tag{19}$$

The argument was that through the Piling-up Lemma, the $\frac{1}{2}$ bit-flip probability of significant bits of w_0 removed the bias of other words from the output. It was quickly pointed out[13] that calculation of value

$$(\text{output}_{\lll 16}) \oplus (\text{output}) \tag{20}$$

removes the direct effect of word w_0. Some effects of w_0 remained through carry bits, but this was not enough to stop the bit-flip bias of the other three words from showing up.

This has influenced us to use an odd number of w_0 copies in the filter of TSC-2.

$$\text{output} = (w_0 \lll 11 + w_1) \lll 14 + (w_0 \lll 13 + w_2) \lll 22 + (w_0 \lll 12 + w_3). \tag{21}$$

Also, for added security against this weakness, rotations were chosen so that they are somewhat skewed from each other. This makes it harder to find rotated sum of output word copies that somehow contain more than one word boundary matchings. The effect of this skewed rotation is easier to see if we compare filter (19) with

$$\text{output} = ((((w_0 \lll 15) + w_0) \lll 7 + w_1) \lll 7 + w_2) \lll 7 + w_3) \tag{22}$$

keeping the approach of (20) in mind. No small number of rotated `XOR`s of output would remove direct effect of w_0.

A more uniform filter In the current submission, we have used a T-function with more uniform bit-flip probability, as any bias is potentially dangerous. But the skewed rotations have remained. It would be interesting to know if replacing the current filter (8) with a more uniform filter, such as

$$\text{output} = \big((y_0) \lll 4 + (y_1) \ggg 4\big)_{\ggg 4} + \big((y_2) \lll 8 + y_3\big)_{\lll 8} \tag{23}$$

would bring about any weakness into the current design.

Carry considerations With respect to what has been discussed so far, our current filter (8) is not any different from the filter

$$f'(\mathbf{y}) = \big((y_0)_{\lll 9} + y_1\big)_{\lll 6} + \big((y_2)_{\lll 7} + y_3\big)_{\ggg 9}. \tag{24}$$

If all modular additions are replaced with XORs, the two filters are identical. We shall argue that filter (8) is preferable.

Consider the overflows from the two additions appearing inside the parenthesis. These are discarded before going onto the middle addition. In filter (24), they are directly affected by MSBs of y_1 and y_3. The point to notice is that these belong to the same state column.

If we look at this from another viewpoint, one can conclude that keeping the whole state identical and changing just the most significant column will have less an effect on the output word than giving changes to other columns. This properties is closely related to how fast small changes in IV will turn into uniform change of internal state during our IV setup process. Compared to (8), using (24) will bring about a less secure state initialization process.

7.5 S-box selection

The choice or construction of our s-box is explained in this section. We first single out good candidates for a single slice of the s-box and weave them together to form the whole 4×4 s-box.

An alternative s-box that could be better in view of implementation is also presented in this section.

Single bit-slice of s-box First of all, notice that the bit-flip probability requirement of Section 6.7 placed on our s-box is actually a condition on each bit of s-box rather than on the whole s-box. Let us explain this more closely.

Consider a binary sequence $\mathfrak{s} = (s_i)_{i \in \mathbf{Z}/16\mathbf{Z}}$ of period 16, containing eight 0s and eight 1s. We can consider the bit-flip probability of this sequence. That is, one checks if bits adjacent to each other in the cyclic sequence is identical or different.

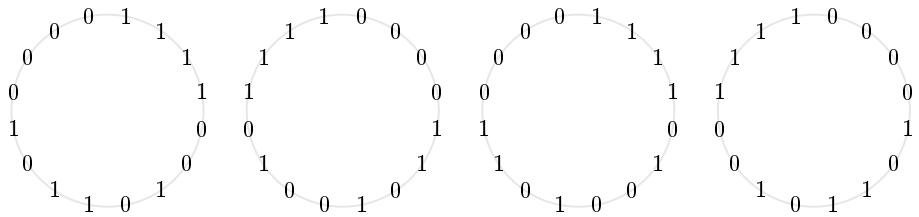
$$\text{bit-flip probability of } \mathfrak{s} = \frac{1}{16}\#\{i \in \mathbf{Z}/16\mathbf{Z} \mid s_i \oplus s_{i+1} = 1\}$$

Similarly, we can consider bit-flip probability of *powers* of $\mathfrak{s}$.

$$\text{bit-flip probability of } \mathfrak{s}^a = \frac{1}{16}\#\{i \in \mathbf{Z}/16\mathbf{Z} \mid s_i \oplus s_{i+a} = 1\}$$

We searched for all period 16 sequences $\mathfrak{s}$ with eight 0s and 1s such that the bit-flip probability of $\mathfrak{s}$, $\mathfrak{s}^2$, $\mathfrak{s}^5$, and $\mathfrak{s}^6$ were all equal to $\frac{1}{2}$. There were 448 of them.

As this was too many, we strengthened our condition to include all odd powers of $\mathfrak{s}$ and found 64 such binary sequences to exist.



As they are periodic sequences, we can gathered each in a circle, and up to rotation, the four shown above are all the sequences.

Combining the slices It now suffices to choose any four of these periodic sequences (allowing duplicates), and do a rotated combination to produce a 4×4 s-box. Up to order of the slices, there are 3840 of them. Notice that for any s-box S constructed in such a manner, the bit-flip probability of powers of S at each bit slice follow the bit-flip probability of powers of the corresponding periodic binary sequence.

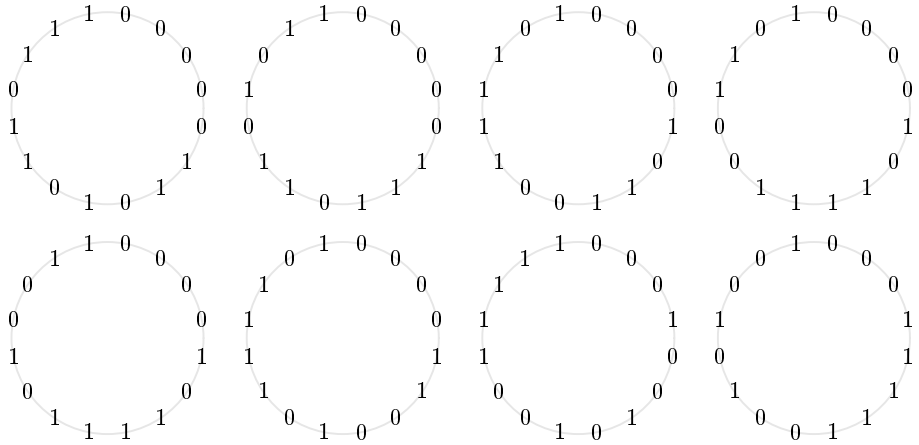
From the set of 3840 s-boxes, we chose at random and drew associated Karnaugh maps until an s-box allowing simple boolean presentation was encountered. But, in doing this, s-boxes that gave completely linear expression as one of the four output bits in either S or S^2 were rejected.

Lighter s-box The decision to remove s-boxes with linear expresions came mostly from the (unjustified) fear of algebraic attacks. When this cipher is better understood, simpler s-boxes may be used.

```
S[16] = {12,4,5,9,6,14,7,15,0,8,1,13,2,10,11,3};        (25)
```

Two output bits of this s-box are linear in the inputs, and S^2, S^5, S^6 all contain linear expressions. Using such an s-box would certainly be advantageous in view of implementation, both in hardware and software.

Stronger s-box Consider the following eight binary sequences of period 16.



These circles $\mathfrak{s}$ all have the property that powers $\mathfrak{s}$, $\mathfrak{s}^2$, $\mathfrak{s}^{13}$, and $\mathfrak{s}^{14}$ exhibit the $\frac{1}{2}$ bit-flip probability. These same powers may be used to define the T-function, if s-box constructed from these circles are used.

Such s-boxes are stronger than our current s-box in view of the attack [17], discussed in Section 6.8. As the work [17] is under submission, we shall not go into details here.

We did not use an s-box constructed from these circles, because we could not find one allowing simple boolean expression.

7.6 Time-memory tradeoff

For this section, we shall assume complete knowledge of the recent work [10] on time-memory tradeoffs. In particular, two well-known time-memory-data(TMD) tradeoffs, namely, the one based of birthday paradox that was presented independently by Babage[3] and Golić[8] (BG-tradeoff) and the tradeoff of Biryukov-Shamir[4] (BS-tradeoff) should be understood.

Over the last few years, TMD tradeoff considerations have started to push designers to use internal states of size larger than twice the key size. Arguments supporting this trend would view the maximum security level achievable by our cipher with 160-bit state to be 80 bits. We believe some of these TMD tradeoffs to be unrealistic, and discuss this matter in this section. For the moment, we do not know of any attack on our cipher of complexity reaching 2^{80}, other than TMD tradeoffs. Hence depending on how the threat of TMD is interpreted, we could view our cipher to be achieving security level greater than 80 bits.

BS-tradeoff When the anticipated security level is applied as the upper limit to the pre-computation time reachable by the attacker, one can easily see that BS-tradeoff reduces security level not to $\frac{1}{2}$, but only to $\frac{2}{3}$ of the state size.

On our cipher of 160-bit state, this translates to 106-bit security. And limiting attacker's pre-computation time to 2^{106} seems to be leaving more than enough margin at the moment.

BG-tradeoff Limiting the amount of data available to the attacker seems more appropriate for BG-tradeoffs. If data is limited to D on a cipher of state size N, BG-tradeoff achieves complexity N/D. On our cipher, limiting data to $D = 2^{80}$ would give us 80-bit security.

Since recovery of key is mostly useful because the same key would be used again with other IVs, there would be many situations where limiting data to size equivalent to IV size is reasonable. Applied to our cipher, with 64-bit IVs, the best BG-tradeoff would be of 96-bit complexity.

Summary We have argued that, with respect to TMD tradeoffs, the security level provided by our cipher with 160-bit internal state can be seen to be slightly higher than 80-bit, contrary to popular perception. This conclusion is reached, if

realistic limits are set for the attacker's ability to do pre-computation and obtain data.

7.7 T-function and conjugation

The multi-word T-function used in TSC-3 can be interpreted as a single-word T-function up to conjugation. This structure was explained to the submitters through [2], and we have summarized it in this section.

Define the 4×4 s-box C, that acts like a counter update, by setting

$$C(t) = t + 1 \pmod{16}.$$

As both S and C are single cycle permutations, in particular, of the same cycle structure, by basic group theory, there exists a permutation R for which

$$S = R^{-1} \circ C \circ R.$$

Notice that $S^a = R \circ C^a \circ R^{-1}$ for any integer a. Now, let $\mathbf{R}$ be the operator acting on the whole internal state, which applies s-box R to each column. Define a change of variable by

$$\mathbf{z} = \mathbf{R}(\mathbf{x})$$

and also define

$$\begin{aligned} \mathbf{T}'(\mathbf{z}) &= \mathbf{R} \circ \mathbf{T} \circ \mathbf{R}^{-1}(\mathbf{z}), \\ \mathbf{p}'(\mathbf{z}) &= \mathbf{p} \circ \mathbf{R}^{-1}(\mathbf{z}). \end{aligned}$$

Then we can write

$$\begin{aligned} [\mathbf{T}'(\mathbf{z})]_i = [\mathbf{R} \circ \mathbf{T} \circ \mathbf{R}^{-1}(\mathbf{z})]_i &= R([\mathbf{T}(\mathbf{x})]_i) \\ &= \begin{cases} R \circ S\ ([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 3, \\ R \circ S^2([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 2, \\ R \circ S^5([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 1, \\ R \circ S^6([\mathbf{x}]_i) & \text{if } [\mathbf{p}(\mathbf{x})]_i = 0. \end{cases} \\ &= \begin{cases} C\ ([\mathbf{z}]_i) & \text{if } [\mathbf{p}'(\mathbf{z})]_i = 3, \\ C^2([\mathbf{z}]_i) & \text{if } [\mathbf{p}'(\mathbf{z})]_i = 2, \\ C^5([\mathbf{z}]_i) & \text{if } [\mathbf{p}'(\mathbf{z})]_i = 1, \\ C^6([\mathbf{z}]_i) & \text{if } [\mathbf{p}'(\mathbf{z})]_i = 0. \end{cases} \end{aligned}$$

Notice that each C^a is the very simple operation of modular addition by a. We have shown that, up to conjugation, $\mathbf{T}$ is the simpler operator $\mathbf{T}'$, with modular addition on columns at the core.

Now, successively reading up each column while proceeding to the left, we can view $\mathbf{T}'$ as a T-function on a single 160-bit word. Hence the T-function of TSC-3 is conjugate to a single-word T-function. This shows new insight into the structure of $\mathbf{T}$, but we have not yet discovered any new security implications.

8 Conclusion

A synchronous streamcipher TSC-3 of 80-bit intended security level was presented with some security and implementation analysis. The cipher is suitable for constrained hardware environments, allowing for a wide range of implementation choices. The cipher is also highly scalable and can be adopted to serve higher security level.

References

1. NIST. A statistical test suite for random and psedorandom number generators for cryptographic applications. NIST Special Publication 800-22.
2. V. Anashin, private communication. Feb., 2005.
3. S. H. Babbage, Improved exhaustive search attacks on stream ciphers. *European Convention on Security and Detection,* IEE Conference publication No. 408, pp. 161–166, IEE, 1995.
4. A. Biryukov and A. Shamir, Cryptanalytic time/memory/data tradeoffs for stream ciphers. *Asiacrypt 2000,* LNCS 1976, pp. 1–13, Springer-Verlag, 2000.
5. M. Daum, Narrow T-functions. *FSE 2005,* to appear as an LNCS volumn.
6. C. De Cannière, J. Lano, and B. Preneel, Comments on the rediscovery of time memory data tradeoffs. A note on the ECRYPT Call for Stream Cipher Primitives page. Available from `http://www.ecrypt.eu.org/stream/TMD.pdf`, 2005.
7. M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, Strong authentication for RFID systems using the AES algorithm. *CHES 2004,* pp. 357–370, Springer-Verlag, 2004.
8. J. Dj. Golić, Cryptanalysis of alleged A5 stream cipher. *Eurocrypt'97,* LNCS 1233, pp. 239–255, Springer-Verlag, 1997.
9. J. Hong, D. H. Lee, Y. Yeom, and D. Han, A new class of single cycle T-functions and a stream cipher proposal. *SASC*(State of the Art of Stream Ciphers, Brugge, Belgium, Oct. 2004) workshop record.
10. J. Hong and P. Sarkar, Rediscovery of time memory tradeoffs. *Cryptology ePrint Archive,* Report 2005/090, 2005.
11. J. Hong, D. H. Lee, Y. Yeom, and D. Han, New class of single cycle T-functions. *FSE 2005,* to appear as an LNCS volumn.
12. P. Junod, S. Künzli, and W. Meier, Attacks on TSC. FSE 2005 rump session presentation.
13. A. Klimov, private communication. Sep., 2004.
14. A. Klimov and A. Shamir, A new class of invertible mappings. *CHES 2002*, LNCS 2523, Springer-Verlag, pp.470–483, 2003.
15. A. Klimov and A. Shamir, Cryptographic application of T-functions. *SAC 2003*, LNCS 3006, Springer-Verlag, pp.248–261, 2004.
16. A. Klimov and A. Shamir, New cryptographic primitives based on multiword T-functions. *FSE 2004*, LNCS 3017, Springer-Verlag, pp.1–15, 2004.
17. S. Künzli, P. Junod, and W. Meier, Distinguishing attacks on T-functions. Submitted to Mycrypt 2005.