New Applications of T-functions in Block Ciphers and Hash Functions

Alexander Klimov and Adi Shamir

Computer Science department,
The Weizmann Institute of Science,
Rehovot 76100, Israel
{ask,shamir}@wisdom.weizmann.ac.il

Abstract. A *T-function* is a mapping from n-bit words to n-bit words in which for each $0 \leq i < n$, bit i of any output word can depend only on bits $0, 1, \ldots, i$ of any input word. All the boolean operations and most of the numeric operations in modern processors are T-functions, and all their compositions are also T-functions. Our earlier papers on the subject dealt with "crazy" T-functions which are invertible mappings (including Latin squares and multipermutations) or single cycle permutations (which can be used as state update functions in stream ciphers). In this paper we use the theory of T-functions to construct new types of primitives, such as MDS mappings (which can be used as the diffusion layers in substitution/permutation block ciphers), and self-synchronizing hash functions (which can be used in self-synchronizing stream ciphers or in "fuzzy" string matching applications).

1 Introduction

There are two basic approaches to the design of secret key cryptographic schemes, which we can call "tame" and "wild". In the tame approach we try to use only simple primitives (such as linear mappings or LFSRs) with well understood behavior, and try to prove mathematical theorems about their cryptographic properties. Unfortunately, the clean mathematical structure of such schemes can also help the cryptanalyst in his attempt to find an attack which is faster than exhaustive search. In the wild approach we use crazy compositions of operations (which mix a variety of domains in a non-linear and non-algebraic way), hoping that neither the designer nor the attacker will be able to analyze the mathematical behavior of the scheme. The first approach is typically preferred in textbooks and toy schemes, but real world designs often use the second approach.

In several papers published in the last few years [4–6] we tried to bridge this gap by considering "semi-wild" constructions, which look like crazy combinations of boolean and arithmetic operations, but have many analyzable mathematical properties. In particular, in these papers we defined the class of T-functions which contains arbitrary compositions of *plus, minus, times, and, or,* and *xor* operations on n-bit words, and showed that it is easy to analyze their invertibility and cycle structure for arbitrary word sizes. This led to the efficient construction of multipermutations and stream ciphers. In this paper we explore additional applications of the theory of T-functions, which do not depend on their invertibility or cycle structure. In particular, we develop new classes of MDS mappings for block ciphers and hash functions, self-synchronizing stream ciphers, and self-synchronizing hash functions which can be used in "fuzzy" string matching to compare strings with a relatively large edit distance.

2 Basic definitions

Let us first introduce our notation. We denote the set $\{0,1\}$ by $\mathbb{B}$. We denote by $[x]_i$ bit number i of word x (with $[x]_0$ being the least significant bit). We use the same symbol x to denote the n-bit vector $([x]_{n-1},\ldots,[x]_0)\in\mathbb{B}^n$ and an integer modulo 2^n, with the usual conversion rule: $x \longleftrightarrow \sum_{i=0}^{n-1} 2^i [x]_i$.

A collection of m n-bit numbers is described either as a column vector of values or as an $m \times n$ bit matrix x. We start numbering its rows and columns from zero, and refer to its i-th row $x_{i-1,\star}$ as x_{i-1} and to its j-th column $x_{\star,j-1}$ as $[x]_{j-1}$.

The basic operations we allow in our mappings are the following *primitive operations*: "+", "−", "×" (*addition*, *subtraction*, and *multiplication* modulo 2^n), "$\vee$", "$\wedge$", and "$\oplus$" (bitwise *or*, *and*, and *xor* on n-bit words). Note that left shift is allowed (since it is equivalent to multiplication by a power of two), but right shift and circular rotations are not included in this definition, even though they are available as basic machine instructions in most microprocessors. It does not mean that we exclude them from further consideration, we just want to use them in a more restricted way.

Definition 1 (T-function). *A function f from $\mathbb{B}^{m\times n}$ to $\mathbb{B}^{l\times n}$ is called a* T-function *if the k-th column of the output $[f(x)]_{k-1}$ depends only on the first k columns of the input $[x]_{k-1},\ldots,[x]_0$:*

$$\begin{aligned}
[f(x)]_0 &= f_0([x]_0),\\
[f(x)]_1 &= f_1([x]_0,[x]_1),\\
[f(x)]_2 &= f_2([x]_0,[x]_1,[x]_2),\\
&\vdots\\
[f(x)]_{n-1} &= f_{n-1}([x]_0,\ldots,[x]_{n-2},[x]_{n-1}).
\end{aligned} \tag{1}$$

The name is due to the Triangular form of (1). It turns out that T-functions are very common since any combination of constants, variables, and primitive operations is a T-function.

Definition 2. *A T-function is called a* parameter *(and denoted by a Greek letter such as α) if each* bit-slice function *f_i does not depend on $[x]_i$.*

If T-functions can be viewed as triangular matrices, then parameters can be viewed as triangular matrices with zeroes on the diagonal (note that these functions are typically non-linear, and thus the matrix form is a visualization metaphor rather than an actual definition). The name "parameter" usually refers to some unspecified constant in an expression, and in this context we use it to denote that in many applications it suffices to analyze the dependence of a bit-slice of the output $[f(x)]_i$ only on the current bit-slice of the input $[x]_i$, and to consider the effect of all the previous bit-slices of the input (e.g., in the form of addition or multiplication carries) as unspecified values.

Given an arbitrary expression with primitive operations, we can recursively apply the following rules to produce a simple representation of its bit-slice mappings using such unspecified parameters. Note that in this representation we only have to distinguish between the least significant bit-slice and all the other bit-slices, regardless of the word length n:

Theorem 1. *For $i > 0$ the following equalities hold*

$$\begin{array}{ll} [x \times y]_0 = [x]_0 \wedge [y]_0, & [x \times y]_i = [x]_i \alpha_{[y]_0} \oplus \alpha_{[x]_0} [y]_i \oplus \alpha_{xy}, \\ [x \pm y]_0 = [x]_0 \oplus [y]_0, & [x \pm y]_i = [x]_i \oplus [y]_i \oplus \alpha_{x \pm y}, \\ [x \oplus y]_0 = [x]_0 \oplus [y]_0, & [x \oplus y]_i = [x]_i \oplus [y]_i, \\ [x \overset{\wedge}{\vee} y]_0 = [x]_0 \overset{\wedge}{\vee} [y]_0, & [x \overset{\wedge}{\vee} y]_i = [x]_i \overset{\wedge}{\vee} [y]_i, \end{array} \qquad (2)$$

where the unspecified parameters α's denote the dependence of the subscripted operation on previous bit-slices.

Consider, for example, the following mapping: $x \rightarrow x + (x^2 \vee 5)$.

$$\left[x + (x^2 \vee 5)\right]_0 = [x]_0 \oplus \left[x^2 \vee 5\right]_0 = [x]_0 \oplus \left(\left[x^2\right]_0 \vee [5]_0\right) = [x]_0 \oplus 1$$

and, for $i > 0$,

$$\begin{aligned} \left[x + (x^2 \vee 5)\right]_i = [x]_i \oplus \left[x^2 \vee 5\right]_i \oplus \alpha_{x+(x^2 \vee 5)} &= [x]_i \oplus \left(\left[x^2\right]_i \vee [5]_i\right) \oplus \alpha_{x+(x^2 \vee 5)} \\ = [x]_i \oplus ([x]_i \alpha_{[x]_0} \oplus \alpha_{[x]_0} [x]_i \oplus \alpha_{x^2}) \vee [5]_i \oplus \alpha_{x+(x^2 \vee 5)} & \\ = [x]_i \oplus \alpha_{x^2} \vee [5]_i \oplus \alpha_{x+(x^2 \vee 5)} &= [x]_i \oplus \alpha. \end{aligned}$$

This mapping is clearly invertible since we can uniquely recover the consecutive bit-slices of the input (from LSB to MSB) from the given bit-slices of the output. A summary of the simplest recursive constructions of parameters can be found in Figure 1 at the end of the paper.

3 A New Class of MDS mappings

In this section we consider the efficient construction of new types of MDS mappings, which are a fundamental building block in the construction of many block ciphers. Unlike all the previous constructions, our mappings are non-linear and non-algebraic, and thus they can provide enhanced protection against differential and linear attacks.

Let X be a finite set and ϕ be an invertible mapping on m-tuples of values from X ($\phi : X^m \rightarrow X^m$). Let $y = \phi(x)$ and $y' = \phi(x')$, where $x =$

$(x_0, \ldots, x_{m-1})^T$, $y = (y_0, \ldots, y_{m-1})^T$, and, similarly, $x' = (x'_0, \ldots, x'_{m-1})^T$, $y' = (y'_0, \ldots, y'_{m-1})^T$, and $x \neq x'$. Let d_x be the number of i's such that $x_i \neq x'_i$, and, similarly, let d_y be the number of differences between y and y'. Let $D_\phi = min_{x,x'}(d_x + d_y)$. Since $d_y \leq m$ and d_x can be equal to 1 it follows that $D_\phi \leq m + 1$ for arbitrary ϕ.

Definition 3. *A mapping ϕ is called* Maximum Distance Separable (MDS) *if* $D_\phi = m + 1$.[1]

If we use ϕ as a diffusion layer in a Substitution Permutation[2] encryption Network (SPN),[3] then every differential [1] or linear [7] characteristic has at least D_ϕ active S-boxes in each pair of consecutive layers of the network. Using this property we can demonstrate resistance to differential and linear cryptanalysis, because in combination with the probability bounds on a single S-box it provides an upper bound on the probability of any differential or linear characteristic. Consequently, MDS mappings are used in many modern block cipher designs including AES [3].

Common constructions of MDS mappings use linear algebra over the finite field $GF(2^n)$. This makes the analysis easier, but has the undesirable side effect that a linear diffusion layer by itself is "transparent" (i.e., has transition probability of 1) to differential and linear attacks. If we could use a "non-transparent" MDS diffusion layer we would simultaneously achieve two goals by spending the same computational effort—forcing many S-boxes to be active and further reducing the upper bound on the probability of characteristics in each diffusion layer.

One way to construct a linear MDS mapping over a finite field is to use the following method. Let

$$\mathcal{W}(a_0, \ldots, a_{m-1}) = \begin{pmatrix} 1 & a_0 & a_0^2 & \ldots & a_0^{m-1} \\ 1 & a_1 & a_1^2 & \ldots & a_1^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{m-1} & a_{m-1}^2 & \ldots & a_{m-1}^{m-1} \end{pmatrix}.$$

It is known that if all the a_i are distinct then this matrix is non-singular. Consider the following mapping

$$x \to y = \mathcal{W}(a_0, \ldots, a_{m-1})\mathcal{W}^{-1}(a_m, \ldots, a_{2m-1})x.$$

[1] In our definition ϕ can be an arbitrary mapping, even though the name MDS usually relates to *linear* mappings or error correcting codes. The alternative definition which counts the number of non-zero entries in a single input/output pair is applicable only to linear codes.

[2] Note that the name "permutation" here is due to the historical tradition since modern designs use for diffusion not a bit permutation (as, e.g., in DES) but a general linear or affine transformation (as, e.g., in AES).

[3] Alternatively ϕ can be used as a diffusion layer in a Feistel construction. Note that in this case ϕ need not to be calculated backwards even during decryption.

Let us show that if all the a_i's are distinct then this mapping is MDS. Let

$$p = \mathcal{W}^{-1}(a_m, \ldots, a_{2m-1})x.$$

If we consider p as the vector of the coefficients of a polynomial then

$$\begin{aligned} x_i &= p_0 + p_1 a_{m+i} + p_2 a_{m+i}^2 + \cdots + p_{m-1} a_{m+i}^{m-1} = p(a_{i+m}), \\ y_i &= p_0 + p_1 a_i \quad\;\; + p_2 a_i^2 \quad\;\; + \ldots + p_{m-1} a_i^{m-1} = p(a_i). \end{aligned}$$

The number of common values c of two distinct polynomials (p and p', defined by the two sets of input/output values) of degree $m-1$ is at most $m-1$ and thus the number of unequal pairs of primed and non-primed values among all the inputs and outputs satisfies

$$d = d_x + d_y = 2m - c \geq m + 1.$$

Consider an example in $\mathbb{F}_{2^3}$ (modulo $\mathtt{b} = 1011_2 = t^3 + t + 1$):

$$\mathcal{W}(\mathtt{1},\mathtt{2},\mathtt{3}) \times \mathcal{W}^{-1}(\mathtt{4},\mathtt{5},\mathtt{6}) = \begin{pmatrix} \mathtt{1\,1\,1} \\ \mathtt{1\,2\,4} \\ \mathtt{1\,3\,5} \end{pmatrix} \times \begin{pmatrix} \mathtt{4\,3\,6} \\ \mathtt{4\,7\,3} \\ \mathtt{5\,6\,3} \end{pmatrix} = \begin{pmatrix} \mathtt{5\,2\,6} \\ \mathtt{5\,3\,7} \\ \mathtt{4\,2\,7} \end{pmatrix}. \tag{3}$$

Notice that this mapping uses multiplication in a finite field. We prefer to use arithmetic modulo 2^n, which is much more efficient in software implementations on modern microprocessors, and would also like to mix arithmetic and boolean operations in order to make cryptanalysis harder. The general T-function methodology in such cases can be summarized as follows:

1. Find a skeleton bitwise mapping from 1-bit inputs to 1-bit outputs which has the desired property (e.g., invertibility).
2. Extend the mapping in a natural way to n-bit words.
3. Add some parameters in order to obtain a larger class of mappings with the same bit-slice properties, and to provide some inter–bit-slice mixing.
4. Change some $\oplus$ operations to *plus* or *minus*, using the fact that they have the same bit-slice mappings (up to the exact definition of some parameters).

Let us apply this T-function methodology to the construction of an MDS mapping with $m = 3$ input words. First of all, we have to represent x_i as a bit vector $(x_{i,u}, x_{i,v}, x_{i,w})$ and represent (3) as a mapping of bits:

$$\begin{aligned}
y_{0,u} &= (x_{0,w}) \oplus (x_{1,v}) \oplus (x_{2,v} \oplus x_{2,u} \oplus x_{2,w}), \\
y_{0,v} &= (x_{0,u}) \oplus (x_{1,u} \oplus x_{1,w}) \oplus (x_{2,w} \oplus x_{2,v}), \\
y_{0,w} &= (x_{0,w} \oplus x_{0,v}) \oplus (x_{1,u}) \oplus (x_{2,u} \oplus x_{2,v}), \\
y_{1,u} &= (x_{0,w}) \oplus (x_{1,u} \oplus x_{1,v}) \oplus (x_{2,v} \oplus x_{2,w}), \\
y_{1,v} &= (x_{0,u}) \oplus (x_{1,v} \oplus x_{1,u} \oplus x_{1,w}) \oplus (x_{2,w}), \\
y_{1,w} &= (x_{0,w} \oplus x_{0,v}) \oplus (x_{1,w} \oplus x_{1,u}) \oplus (x_{2,w} \oplus x_{2,u} \oplus x_{2,v}), \\
y_{2,u} &= (x_{0,u} \oplus x_{0,w}) \oplus (x_{1,v}) \oplus (x_{2,v} \oplus x_{2,w}), \\
y_{2,v} &= (x_{0,u} \oplus x_{0,v}) \oplus (x_{1,u} \oplus x_{1,w}) \oplus (x_{2,w}), \\
y_{2,w} &= (x_{0,v}) \oplus (x_{1,u}) \oplus (x_{2,w} \oplus x_{2,u} \oplus x_{2,v}).
\end{aligned}$$

Note that multiplication in our field works as follows:

$$\begin{aligned}
(x_{i,u}, x_{i,v}, x_{i,w}) \times \mathtt{1} &= (x_{i,u}, & x_{i,v}, & & x_{i,w}\ ),\\
(x_{i,u}, x_{i,v}, x_{i,w}) \times \mathtt{2} &= (x_{i,v}, & x_{i,w} \oplus x_{i,u}, & & x_{i,u}\ ),\\
(x_{i,u}, x_{i,v}, x_{i,w}) \times \mathtt{4} &= (x_{i,w} \oplus x_{i,u}, & x_{i,u} \oplus x_{i,v}, & & x_{i,v}\ ).
\end{aligned}$$

So, for example, to get the topmost-leftmost block we calculate

$$\mathtt{5}x_0 = \mathtt{1}x_0 \oplus \mathtt{4}x_0 = \begin{pmatrix} x_{0,u} \oplus (x_{0,w} \oplus x_{0,u}) \\ x_{0,v} \oplus (x_{0,u} \oplus x_{0,v}) \\ x_{0,w} \oplus \quad x_{0,v} \end{pmatrix} = \begin{pmatrix} x_{0,w} \\ x_{0,u} \\ x_{0,w} \oplus x_{0,v} \end{pmatrix}.$$

Let us now consider each $x_{i,\cdot}$ and $y_{i,\cdot}$ not as a single-bit variable but as a whole n-bit word, so each x_i and y_i is now of length $3n$. Suppose that for $(x_0, \ldots, x_m)^T$ and $(x'_0, \ldots, x'_m)^T$ the number of differences $d_x = D > 0$, that is there are D values of i such that $x_i \neq x'_i$. It follows that in each bit-slice $d_{[x]_j} \leq D$ and so, since at least one bit-slice was changed and the bit-slice mapping

$$\begin{pmatrix} [x_0]_j \\ \vdots \\ [x_{m-1}]_j \end{pmatrix} \to \begin{pmatrix} [y_0]_j \\ \vdots \\ [y_{m-1}]_j \end{pmatrix}$$

is MDS, it follows that $d_{[y]_j} \geq m + 1 - D$, and thus $d_y \geq m + 1 - D$, that is $d_x + d_y \geq m + 1$, and thus the whole mapping $x \to y$ is also MDS.

Our next goal is to introduce arbitrary parameters in order to define a much larger class of mappings. Note that if

$$\begin{pmatrix} x_0 \\ \vdots \\ x_{m-1} \end{pmatrix} \to \begin{pmatrix} y_0 \\ \vdots \\ y_{m-1} \end{pmatrix} \quad \text{is MDS then} \quad \begin{pmatrix} x_0 \\ \vdots \\ x_{m-1} \end{pmatrix} \to \begin{pmatrix} \phi_0(y_0) \\ \vdots \\ \phi_{m-1}(y_{m-1}) \end{pmatrix},$$

where the ϕ_i's are any invertible mappings, is also MDS. Since $\phi : x \to x \oplus \alpha$ is an invertible mapping it follows that the introduction of *additive* parameters preserves the MDS property of bit-slices. Consequently, we can replace some "$\oplus$"s with "+"s or "−"s, and add arbitrary parameters in order to obtain the following "crazier" mapping which is also provably MDS:

$$\begin{aligned}
y_{0,u} &= x_{0,w} - (x_{1,v} \oplus x_{2,v}) + (x_{2,u} \oplus x_{2,w}) && \oplus\, 2x_{0,u}x_{1,v},\\
y_{0,v} &= (x_{0,u} + x_{1,u} - (x_{1,w} \oplus x_{2,w})) \oplus x_{2,v} && \oplus\, 2x_{0,v}x_{1,w},\\
y_{0,w} &= x_{0,w} \oplus (x_{0,v} + x_{1,u}) \oplus x_{2,u} \oplus x_{2,v} && \oplus\, 2x_{0,w}x_{2,u},\\
y_{1,u} &= x_{0,w} + (x_{1,u} \oplus x_{1,v} \oplus x_{2,v}) + x_{2,w} && \oplus\, 2x_{1,u}x_{2,v},\\
y_{1,v} &= x_{0,u} \oplus (x_{1,v} + x_{1,u} - x_{1,w}) \oplus x_{2,w} && \oplus\, 2x_{1,v}x_{2,w},\\
y_{1,w} &= (x_{0,w} - x_{0,v} - x_{1,w}) \oplus x_{1,u} \oplus (x_{2,w} - (x_{2,u} \oplus x_{2,v})) && \oplus\, 2x_{1,w}x_{0,u},\\
y_{2,u} &= x_{0,u} \oplus (x_{0,w} + x_{1,v} + x_{2,v}) \oplus x_{2,w} && \oplus\, 2x_{2,u}x_{0,v},\\
y_{2,v} &= x_{0,u} - x_{0,v} + (x_{1,u} \oplus x_{1,w} \oplus x_{2,w}) && \oplus\, 2x_{2,v}x_{0,w},\\
y_{2,w} &= (x_{0,v} + x_{1,u} \oplus x_{2,w}) - (x_{2,u} \oplus x_{2,v}) && \oplus\, 2x_{2,w}x_{1,u}.
\end{aligned}$$

This mapping allows us to intermix 3 S-boxes of $3n$ bits each. It is possible to construct a similar mapping which allows us to intermix m S-boxes of ln bits each as long as $2m \leq 2^l$, since in this case $\mathbb{F}_{2^l}$ contains $2m$ different elements. In the above example $m = l = 3$, and so the block size is $mln = 576$ bits for $n = 64$, and the size of each S-box is $ln = 192$ bits. Although in some applications (e.g., hash functions or stream ciphers) this is not a limitation, in others (e.g., block ciphers) such long blocks can be a problem. Note that for embedded low-end devices $n = 8$ and so the above example is too small ($mln = 72$ and $ln = 24$), but if we use larger parameters, such as $l = 4$ and $m \leq 8$ in a 128-bit block cipher, we can intermix, for example, the outputs of four 32-bit S-boxes by an MDS mapping.

4 Simpler mappings which are almost MDS

The constructions in the previous section were somewhat complicated and did not have ideal parameter sizes (even if we take into account a slight improvement described in the appendix). The source of the problem was that a non-trivial linear mapping cannot be MDS modulo 2 and thus it is provably impossible to have $l = 1$ and $m > 1$. Fortunately, we can use much simpler functions which are almost MDS, and which are almost as useful as actual MDS functions in cryptographic applications.

Define a mapping as an *almost MDS mapping* if $d_x + d_y \geq m$. Such a diffusion layer guarantees that at least m (instead of $m + 1$) S-boxes are active in any pair of consecutive layers in a substitution permutation network, and thus in many block cipher designs they provide almost the same upper bound on the probability of the cipher's differential and linear characteristics.

Let us construct an almost MDS mapping with conveniently sized parameters. As usual we start with a skeleton of the bit-slices. In this case we use the simple skeleton:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0\,1\,1\,1 \\ 1\,0\,1\,1 \\ 1\,1\,0\,1 \\ 1\,1\,1\,0 \end{pmatrix} \times \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix},$$

where the α_i are arbitrary constants. It is easy to check that $d_x + d_y \geq 4$. Using this skeleton and replacing the constants α_i by simple parameters we can construct, for example, the following non-linear almost MDS mapping:

$$\begin{aligned} y_0 &= x_1 + (x_2 \oplus x_3)(2x_0 + 1), \\ y_1 &= x_2 + (x_3 \oplus x_0)(2x_1 + 1), \\ y_2 &= x_3 + (x_0 \oplus x_1)(2x_2 + 1), \\ y_3 &= x_0 + (x_1 \oplus x_2)(2x_3 + 1). \end{aligned}$$

This mapping can diffuse the outputs of four S-boxes of arbitrary sizes (e.g., 32-bit to 32-bit computed S-boxes in a 128-bit block cipher) in a way that guarantees that at least 4 S-boxes are active in any pair of consecutive layers. A similar construction can be used to diffuse a higher number of smaller (computed or stored) S-boxes in other designs.

An interesting application of this mapping is to enhance the security of SHA-1 against the recently announced collision attacks. The first step in SHA-1 is to linearly expand the 16 input words into 80 output words. The linearity of this process makes it easy to search for low Hamming weight differential patterns in the output words without committing to actual input values. We propose to replace the linear expansion by the following process: Arrange the 16 input words in a 4×4 array, and alternately apply our 4-word nonlinear mapping to its rows and columns. This is similar to the AES encryption process, but without keys and S-boxes, and using 32-bit words rather than bytes as array elements. Each application produces a new batch of 16 output words, and thus two row mixings and two column mixings suffice to expand the 16 input words into 80 output words. To enhance the mixing of the weak LSBs in T-functions, we propose to cyclically rotate each generated word by a variable number of bits. The nonlinearity of the mapping makes it difficult to predict the evolution of differential patterns, the MDS property provides lower bounds on their Hamming weights, and thus the modified SHA-1 offers greatly enhanced protection against the new attacks.

5 Self-synchronizing functions

In this section we propose several novel applications of T-functions which are based on the observation that the iterated application of a parameter slowly "forgets" its distant history in the same way that a triangular matrix with zeroes on the diagonal converges to the zero matrix when raised to a sufficiently high power. Let us start with the following definition:

Definition 4 (SSF). *Let* $\left\{c^{(i)}\right\}_{i=0,\dots}$ *and* $\left\{\hat{c}^{(i)}\right\}_{i=0,\dots}$ *be two input sequences, let* $s^{(0)}$ *and* $\hat{s}^{(0)}$ *be two initial states, and let* K *be a common key. Assume that the function* $\mathcal{U}$ *is used to update the state based on the current input and the key:* $s^{(i+1)} = \mathcal{U}(s^{(i)}, c^{(i)}, K)$ *and* $\hat{s}^{(i+1)} = \mathcal{U}(\hat{s}^{(i)}, \hat{c}^{(i)}, K)$*. The function* $\mathcal{U}$ *is called a* self-synchronizing function (SSF) *if equality of any* k *consecutive inputs implies the equality of the next state, where* k *is some integer:*

$$c^{(i)} = \hat{c}^{(i)}, \quad \dots, \quad c^{(i+k-1)} = \hat{c}^{(i+k-1)} \qquad \Longrightarrow \qquad s^{(i+k)} = \hat{s}^{(i+k)}.$$

Let us now show why the T-function methodology is ideally suited to the construction of SSFs:

Theorem 2. *Let the T-function* $\mathcal{U}(s, c, K)$ *be a parameter with respect to the state* s *and an arbitrary function of the input* c *and the key* K*:*

$$[\mathcal{U}(s, c, K)]_i = f_i([s]_{0,\dots,i-1}, [c]_{0,\dots,n-1}, K),$$

then $\mathcal{U}$ *is an SSF.*

Proof. To prove the theorem it is enough to note that bit number i of $s^{(t)}$ can depend only on

$$(\left[s^{(t-1)}\right]_{0,\ldots,i-1}, c^{(t-1)}, K),$$

that, in turn, can depend only on

$$(\left[s^{(t-2)}\right]_{0,\ldots,i-2}, c^{(t-1)}, c^{(t-2)}, K),$$

... that, in turn, can depend only on

$$(\left[s^{(t-i)}\right]_{0}, c^{(t-1)}, \ldots, c^{(t-i)}, K),$$

and, finally, this can depend only on

$$(c^{(t-1)}, \ldots, c^{(t-i-1)}, K).$$

So if all the calculations are done modulo 2^n, $s^{(t)}$ can depend on $c^{(t-1)}, \ldots, c^{(t-n)}$, but cannot depend on any earlier input.

By using the parameters described in Figure 1 it is easy to construct a large variety of SSFs, for example, $\mathcal{U}(s, c, K) = 2s \oplus cK$ or $\mathcal{U}(s, c, K) = ((s \oplus K)^2 \vee 1) + c$. In different applications it may be important to have different k values (representing the number of steps needed to resynchronize). Our constructions seem to be limited to $k = n$ steps (using n^2 input bits), where n is the word-size of the processor (usually, $n = 32$ or $n = 64$). However, it is easy to enlarge or shrink the size of the *effective region* by adjusting the size of c used in each iteration. For example, on a 64-bit processor the above construction has an effective region of size 2^{12} bits, using one byte of c in each iteration we can reduce it down to 2^9, or enlarge it up to 2^{15} if we use eight 64-bit words at a time. Alternatively, to avoid performance penalties, we can use $\mathcal{U}(s, c, K)$ which is a *multiple parameter* with respect to s:

$$[\mathcal{U}(s, c, K)]_i = f_i([s]_{0,\ldots,i-p}, [c]_{0,\ldots,n-1}),$$

where p is some integer. For such function $s^{(t)}$ depends only on $c^{(t-1)}, \ldots, c^{(t-\frac{n}{p})}$. For example, $\mathcal{U}(s, c, K) = ((s \Lsh 8) \oplus c) \times (c \vee 1)$, where $a \Lsh b$ denotes left shift of a by b bit positions, depends only on eight ($\frac{64}{8} = 8$) previous c's.

SSFs have many applications including cryptography, fast methods for bringing remote files into sync, finding duplications on the web, et cetera. Let us now describe those applications in more detail.

Self-synchronizing stream ciphers allow parties to continue their communication even if they temporarily lose synchronization: after processing k additional ciphertexts the receiver automatically resynchronizes it state with the sender. The standard way to achieve this is to create a state which is the concatenation of the last k ciphertext symbols, and to compute the next pseudo random value as the keyed hash of this state. However, in stream cipher construction speed is extremely important, and thus the active maintenance of such a concatenation (adding the newest input, deleting the oldest input, and shifting the other inputs) wastes precious cycles. In addition, the opponent always knows and can sometimes manipulate this state, and thus the hash function has to be relatively strong (and thus relatively slow) in order to withstand cryptanalysis. We propose to combine the state maintenance and the hash operation (and thus eliminate the computational overhead of the state maintenance) by applying a mapping which is a parameter with respect to the state and an arbitrary function with respect to the ciphertext and secret key. This keeps the current state secret, and allows us to use a potentially weaker hash function to produce the next output. More formally, let $\{p^{(i)}\}_{i=0,\dots}$ denote the plaintext, K denote the secret key, $\{s^{(i)}\}_{i=0,\dots}$ denote the internal state, and $\mathcal{I}$ denote the initialization function. The state is initially set to $s^{(0)} = \mathcal{I}(K)$, and then it evolves over time by an SSF update operation $\mathcal{U}$: $s^{(i+1)} = \mathcal{U}(s^{(i)}, c^{(i)}, K)$, where $\{c^{(i)}\}_{i=0,\dots}$ denotes the ciphertext which is produced using an output function $\mathcal{O}$: $c^{(i)} = p^{(i)} \oplus \mathcal{O}(s^{(i)}, K)$.

The actual construction of a *secure* self-synchronizing stream cipher requires great care. Unlike PRNG, where the known-plaintext attack is usually the only one to be considered, there are many reasonable attacks on a self-synchronizing stream cipher:

- known plaintext attack,
- chosen plaintext attack,
- chosen ciphertext attack, and probably even
- related key attack.

To avoid some of these attacks, it is recommended to use a *nonce* in the initialization process to make sure that the opponent cannot restart the stream cipher in the same state.

In this paper we propose a general methodology for the construction of cryptographic primitives rather than fully specified schemes, but let us give one concrete example in order to demonstrate our ideas and encourage further research on the new approach. Let the state s consist of three 64-bit words: $s = (s_0, s_1, s_2)^T$. At each iteration, we would like to output a 64-bit pseudo random value which can be xored with the next plaintext to produce the next ciphertext. Since in the T-function–based constructions the LSBs are usually weaker than the MSBs, the proposed output function swaps the high and low halves:

$$\mathcal{O}(s_0, s_1, s_2) = ((s_0 \oplus s_2 \oplus K_{\mathcal{O}}) \circlearrowleft 32) \times ((s_1 \oplus K'_{\mathcal{O}}) \circlearrowleft 32) \vee 1),$$

where $a \circlearrowleft b$ denotes circular left shift of a by b bit positions. The state is updated by the following function:

$$\begin{pmatrix} s_0 \\ s_1 \\ s_2 \end{pmatrix} \to \begin{pmatrix} (((s'_1 \oplus s'_2) \vee 1) \oplus K_0)^2 \\ (((s'_0 \oplus s'_2) \vee 1) \oplus K_1)^2 \\ (((s'_0 \oplus s'_1) \vee 1) \oplus K_2)^2 \end{pmatrix},$$

where $s'_0 = s_0 + c$, $s'_1 = s_1 - (c \circlearrowleft 21)$, and $s'_2 = s_2 \oplus (c \circlearrowright 21)$. The best attack we are aware of against this particular example requires $O(2^{96})$ time.

Let us now consider some non-cryptographic applications of self-synchronizing functions. Suppose that we want to update a file on one machine (receiver) to be identical to a similar file on another machine (sender) and we assume that the two machines are connected by a low-bandwidth high-latency bidirectional communications link. The simplest solution is to break the file into blocks, and to send across the hashed value of each block in order to identify (and then correct) mismatches. However, if one of the two files has a single missing (or added) character, then all the blocks from that point onwards will have different hash values due to framing errors. The *rsync* algorithm [8] allows two parties to find and recover from such framing errors by asking one party to send the hash values of all the non-overlapping k-symbol blocks, and asking the other party to compare them to the locally computed hash values of all the possible k-symbol blocks (at any offset, not just at locations which are multiples of k). To get the fastest possible speed in such a comparison, we can again eliminate the block maintenance overhead by using a single pass SSF computation which repeatedly updates its internal state with the next character and compares the result to the hash values received from the other party. Such an incremental hash computation can overcome framing errors by automatically forgetting its distant history.

Self-synchronizing functions are also useful in "fuzzy" string matching applications, in which we would like to determine if two documents are sufficiently similar, even though they can have a relatively large edit distance (of changed, added, deleted or rearranged words). Computing the edit distance between two documents is very expensive, and finding a pair of similar documents in a large collection is even harder. To overcome this difficulty, Broder et al. [2] introduced the following notion of *resemblance* of two documents A and B:

$$r(A, B) = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|},$$

where $S(\cdot)$ denotes the set of all the overlapping word k-grams (called *shingles*) of a document, and $|\cdot|$ denotes the size of a set. It was suggested [2] that a good estimate of this resemblance can be obtained by computing the set of hash values of all the shingles in each document, reducing each set into a small number of representative values (such as the hash values which are closest to several dozen particular target values), and then computing the similarity expression above for just the representative values. Since each document can be independently summarized, we get a linear rather than a quadratic algorithm for finding similar pairs of documents in a large collection. In web applications, this makes it possible to analyze the structure of the web, to reduce the size of its cached copies, to find popular documents, or to identify copyright violations.

Notice that the notion of the adversary in the non-cryptographic applications of SSFs is rather limited, and thus we are not bothered by some of the inherent limitations of any such similarity checking procedure for web documents. For example, there are many ways to trick a web crawler into "thinking" that your arbitrary document is similar to a totally different document, or that very similar documents are quite different. The techniques range from checking the IP address of the tester (returning to the crawler a different page than to other users), to creating web pages in such a way that after the execution of `JavaScript` it displays the text on the screen in a completely different way than the raw text which is "seen" by a crawler. Thus it seems that we should not over-design the application to withstand sophisticated attacks against it, and just make sure that the hashed values are random looking and reasonably unpredictable when we use a secret key K in the initialization and state update functions.

References

1. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," CRYPTO 1990.
2. A. Broder, S. Glassman, M. Manasse, and G. Zweig, "Syntactic Clustering of the Web." Available from http://decweb.ethz.ch/WWW6/Technical/Paper205/Paper205.html
3. J. Daemen, V. Rijmen, "AES Proposal: Rijndael," version 2, 1999.
4. A. Klimov and A. Shamir, "A New Class of Invertible Mappings," Workshop on Cryptographic Hardware and Embedded Systems (CHES), 2002.
5. A. Klimov and A. Shamir, "Cryptographic Applications of T-functions," Selected Areas in Cryptography (SAC), 2003.
6. A. Klimov and A. Shamir, "New Cryptographic Primitives Based on Multiword T-functions," Fast Software Encryption Workshop (FSE), 2004.
7. M. Matsui, "Linear Cryptanalysis Method for DES Cipher," EUROCRYPT 1993.
8. A. Tridgell and P. Mackerras, "The rsync algorithm." Available from http://rsync.samba.org/tech_report/

Smaller MDS mappings

In Section 3 we considered MDS mappings which allow us to intermix m S-boxes of ln bits each as long as $2m \leq 2^l$. In order to study if this inequality is an essential condition or just an artifact of that skeleton construction method let us first consider the following question: is it possible to construct an MDS T-function such that $l = 1$ and $m > 1$. The following reasoning shows that it is impossible. Suppose that we constructed such an MDS T-function θ. Let

$$\begin{aligned} x &= (0, \quad 0, \quad 0, \ldots 0)^T, \\ x' &= (2^{n-1}, 0, \quad 0, \ldots 0)^T, \\ x'' &= (0, \quad 2^{n-1}, 0, \ldots 0)^T, \end{aligned}$$

and θ maps them into y, y', and y'' respectively. Since $x = x' = x'' \pmod{2^{n-1}}$ and θ is a T-function it follows that $y = y' = y'' \pmod{2^{n-1}}$. So the only difference between y, y', and y'' is in the most significant bit. Our mapping θ is an MDS and $d_{x,x'} = 1$ so $d_{y,y'} \geq m$ and thus the most significant bit in y'_i is the inverse of y_i:

$$\forall i, [y'_i]_{m-1} = \overline{[y_i]_{m-1}}.$$

For the same reason

$$\forall i, [y''_i]_{m-1} = \overline{[y_i]_{m-1}},$$

and so $y' = y''$ which is a contradiction because any MDS mapping has to be invertible.

Although the algorithm which uses finite field arithmetic does not allow us to construct a mapping intermixing three S-boxes such that each one of them consists of fewer than three words ($m = 3$ and $l < 3$), it is possible to construct such a mapping using a different algorithm. Since we already know that the case of $m = 3$ and $l = 1$ is impossible let us try to construct an MDS mapping with $m = 3$ and $l = 2$. To do it we need a skeleton for bit-slices that is an MDS mapping

$$\psi : \mathbb{B}^2 \times \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2 \times \mathbb{B}^2 \times \mathbb{B}^2.$$

Using a computer search we found the following mapping:[4]

00⋆ → 000 111 222 333	01⋆ → 123 032 301 210	02⋆ → 231 320 013 102	03⋆ → 312 203 130 021
10⋆ → 132 023 310 201	11⋆ → 011 100 233 322	12⋆ → 303 212 121 030	13⋆ → 220 331 002 113
20⋆ → 213 302 031 120	21⋆ → 330 221 112 003	22⋆ → 022 133 200 311	23⋆ → 101 010 323 232
30⋆ → 321 230 103 012	31⋆ → 202 313 020 131	32⋆ → 110 001 332 223	33⋆ → 033 122 211 300

Interestingly, this mapping is linear:

$$\begin{pmatrix} x_{0,u} \\ x_{0,v} \\ x_{1,u} \\ x_{1,v} \\ x_{2,u} \\ x_{2,v} \end{pmatrix} \to \begin{pmatrix} 1\,0 & 1\,0 & 1\,0 \\ 0\,1 & 0\,1 & 0\,1 \\ 0\,1 & 1\,1 & 1\,0 \\ 1\,1 & 1\,0 & 0\,1 \\ 1\,1 & 0\,1 & 1\,0 \\ 1\,0 & 1\,1 & 0\,1 \end{pmatrix} \begin{pmatrix} x_{0,u} \\ x_{0,v} \\ x_{1,u} \\ x_{1,v} \\ x_{2,u} \\ x_{2,v} \end{pmatrix}.$$

Using it as a skeleton we can construct, for example, the following "crazy" mapping:

$$\begin{pmatrix} x_{0,u} \\ x_{0,v} \\ x_{1,u} \\ x_{1,v} \\ x_{2,u} \\ x_{2,v} \end{pmatrix} \to \begin{pmatrix} x_{0,u} + (x_{1,u} \oplus x_{2,u})(2x_{0,v}x_{1,v} + 1) \\ x_{0,v} - x_{1,v} + x_{2,v} \oplus ((x_{0,u} \oplus x_{1,u})^2 \vee 1) \\ (x_{0,v} \oplus x_{1,u})(2x_{0,v}x_{2,v} - 1) - x_{1,v} + x_{2,u} \\ (x_{0,u} + x_{0,v}) \oplus (x_{1,u} - x_{2,v})((x_{1,v} \oplus x_{2,u})^2 \vee 1) \\ (x_{0,u} - x_{0,v})(2x_{1,u}x_{2,v} + 1) + x_{1,v} \oplus x_{2,u} \\ (x_{0,u} \oplus x_{1,u}) - (x_{1,v} \oplus x_{2,v}) + ((x_{0,v} \oplus x_{2,u})^2 \vee 1) \end{pmatrix}.$$

[4] This notation means that

$$\begin{array}{ccc} (x_0 = 0, x_1 = 0, x_2 = 0) & \mapsto & (y_0 = 0, y_1 = 0, y_2 = 0), \\ & \vdots & \\ (x_0 = 3, x_1 = 3, x_2 = 3) & \mapsto & (y_0 = 3, y_1 = 0, y_2 = 0). \end{array}$$

Formal representation	Examples
$\mathcal{P}$:	
$\mathcal{C}$	3
$\mathcal{E} \times \mathcal{E}$	$(x+5)^2$
$\mathcal{P}_0$	$2x$
$(\mathcal{P} \pm \mathcal{E}) \oplus \mathcal{E}$	$(5+x) \oplus x$
$\mathcal{P}' \circ \mathcal{P}''$	$x^2 + 2x$
$\mathcal{P}_0$:	
$\mathcal{C}_0$	2
$\mathcal{C}_0 \times \mathcal{E}$	$2(x^3 \wedge x)$
$\mathcal{P} \wedge \mathcal{C}_0$	$x^2 \wedge 1 \ldots 10_2$
$\mathcal{P}_0 \times \mathcal{P}$	$(x^2 \wedge 1 \ldots 10_2)((5+x) \oplus x)$
$\mathcal{P}_0' \circ \mathcal{P}_0''$	$2x - (x^2 \wedge 1 \ldots 10_2)$

$\mathcal{C}$	constant
$\mathcal{P}$	parameter
$\mathcal{E}$	expression
$\mathcal{C}_0$	constant with 0 in the least significant bit
$\mathcal{P}_0$	parameter with 0 in the least significant bit
$\circ$	primitive operation

This table summarize techniques most commonly used to construct parameters. Note that in a single line the same symbol denotes the same expression (e.g., $\mathcal{E} \times \mathcal{E}$ denotes squaring). Keep in mind that expressions obtained by means of this table are not necessary parameters in the least significant bit-slice (clearly, $\mathcal{P}_0$ are parameters everywhere).

Fig. 1. Common parameter construction techniques.