New Cryptographic Primitives Based on Multiword T-functions

Alexander Klimov and Adi Shamir

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

Abstract. A *T-function* is a mapping from n-bit words to n-bit words in which for each $0 \le i < n$ bit i of the output can depend only on bits $0, 1, \ldots, i$ of the input. All the boolean operations and most of the numeric operations in modern processors are T-functions, and their compositions are also T-functions. In earlier papers we considered 'crazy' T-functions such as $f(x) = x + (x^2 \vee 5)$, proved that they are invertible mappings which contain all the 2^n possible states on a single cycle for any word size n, and proposed to use them as primitive building blocks in a new class of software-oriented cryptographic schemes. The main practical drawback of this approach is that most processors have either 32 or 64 bit words, and thus even a maximal length cycle (of size 2^{32} or 2^{64}) may be too short. In this paper we develop new ways to construct invertible T-functions on multiword states whose iteration is guaranteed to yield a single cycle of arbitrary length (say, 2^{256}). Such mappings can lead to stream ciphers whose software implementation on a standard Pentium 4 processor can encrypt more than 5 gigabits of data per second, which is an order of magnitude faster than previous designs such as RC4.

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 feedback shift registers) with well understood behaviour, 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 nonlinear and nonalgebraic way), hoping that neither the designer nor the attacker will be able to analyse the mathematical behaviour 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 [5, 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, we defined the class of T-functions which contains arbitrary compositions of plus, minus, times, or, and, xor operations on n-bit words, and showed that it is easy to analyse their invertibility and cycle structure for arbitrary word sizes. Such constructions can replace LFSRs and linear congruential mappings (which are vulnerable to correlation and algebraic attacks) in a new class of stream ciphers and pseudo random generators.

The paper is organized in the following way. In section 2 we recall the basic definitions from [5] for single word mappings, and consider several ways in which they can be extended to the multiword case. In section 3 we extend our bit-slice technique to analyse the invertibility of multiword T-functions. In section 4 we extend our technique from [6] to analyse the cycle structure of multiword T-functions. Finally, in section 5 we provide experimental data on the speed of several possible implementations of our functions on a PC.

2 Multiword T-functions

Invertible mappings with a single cycle have many cryptographic applications. The main context in which we study them in this paper is pseudo random generation and stream ciphers. Modern microprocessors can directly operate on up to 64-bit words in a single clock cycle, and thus a univariate mapping can go through at most 2^{64} different states before entering a cycle. In some cryptographic applications this cycle length may be too short, and in addition the cryptanalyst can guess a 64 bit state in a feasible computation. A common way to increase the size of the state and extend the period of a generator is to run in parallel and combine the outputs of several generators with different periods. The overall period is determined by the least common multiple of their individual periods. This works well with LFSRs, whose periods $2^{n_1}-1, 2^{n_2}-1, \dots$ can be relatively prime, and thus the overall period can be their product. However, our univariate mappings have periods of $2^{n_1}, 2^{n_2}, \dots$ whose least common multiple is just $2^{\max(n_1, n_2, \dots)}$.

A partial solution to this problem is to cyclically use a large number of different state update functions, starting from a secret state and a secret index. For example, we can use 64-bit words and $2^{16}-1$ different constants C_k to get a guaranteed cycle length of almost 2^{80} from the following simple generator:

Theorem 1. *Consider the sequence* $\{(x_i, k_i)\}$ *defined by iterating*

$$x_{i+1} = x_i + (x_i^2 \vee C_{k_i}) \bmod 2^n,$$
$$k_{i+1} = k_i + 1 \bmod m,$$

where each x *is an* n*-bit word and* C_k *is some* n*-bit constant for each* $k = 0, \dots, m-1$*. Then the sequence of pairs* (x_i, k_i) *has a maximal period (of size* $m2^n$*) if and only if* m *is odd, and for all* k*,* $[C_k]_0 = 1$ *and* $\bigoplus_{k=0}^{m-1} [C_k]_2 = 1$*.*

A special case of this theorem for $m = 1$ is that the function $f(x) = x + (x^2 \vee C)$ is invertible with a single cycle if and only if both the least significant bit and the third least significant bit in C are 1, and the smallest such C is 5.

Unfortunately, the cyclic change of state update functions is inconvenient, and it cannot yield really large cycles (e.g., of 2^{256} possible states). We can try to solve the problem by using a single high precision variable x (say, with 256 bits), but the multiplication of such long variables can become prohibitively expensive. What we would like to do is to define the mapping by operating separately on the various input words, without trying to interpret the result as a natural mathematical operation on multi-precision words.

Let us first review the definitions from [5] in the case of univariate mappings. Let x be an n-bit word. We can view x as a vector of bits denoted by $([x]_{n-1}, \ldots, [x]_0)$, where the least significant bit has number 0. In this bit notation, the univariate function $f(x) = x + 1 \pmod{2^n}$ can be expressed in the following way:

$$\begin{array}{lll} [f(x)]_0 & = f_0([x]_0) & = [x]_0 \oplus 1 \\ [f(x)]_1 & = f_1([x]_1 ; [x]_0) & = [x]_1 \oplus \alpha_1([x]_0) \\ [f(x)]_2 & = f_2([x]_2 ; [x]_1 , [x]_0) & = [x]_2 \oplus \alpha_2([x]_1 , [x]_0) \\ & \vdots & \vdots \\ [f(x)]_{n-1} & = f_{n-1}([x]_{n-1} ; [x]_{n-2} , \ldots , [x]_0) & = [x]_{n-1} \oplus \alpha_{n-1}([x]_{n-2} , \ldots , [x]_0), \end{array} \tag{1}$$

where each α_i denotes one of the carry bits. Note that for any bit position i, $[f(x)]_i$ depends only on $[x]_i, \ldots, [x]_0$ and does not depend on $[x]_{n-1}, \ldots, [x]_{i+1}$. We call any univariate function f which has this property a *T-function* (where 'T' is short for *triangular*). Note further that each carry bit α_i depends only on strictly earlier input bits $[x]_{i-1}, \ldots, [x]_0$ but not on $[x]_i$. This is a special type of a T-function, which we call a *parameter*. To provide some intuition from the theory of linear transformations on n-dimensional spaces, we can say that T-functions roughly correspond to lower triangular matrices, parameters roughly correspond to lower triangular matrices with zeroes on the diagonal, and a T-function can be roughly represented as a diagonal matrix plus a parameter.

Let us now define these notions for functions which map several input words into one output word. The natural extension of the notion of a T-function in this case is to allow bit i of the output to depend only on bits 0 to i of each one of the inputs. The observation which makes this notion interesting is that all the boolean operations and most of the arithmetic operations available on modern processors are T-functions. In particular, addition ('$+$'), subtraction ('binary $-$'), negation ('unary $-$'), multiplication ('$*$'), or ('$\vee$'), and ('$\wedge$'), exclusive or ('$\oplus$'), and complementation ('$\neg$') (where the boolean operations are performed on all the n bits in parallel and the arithmetic operations are performed modulo 2^n) are T-functions with one or two inputs. We call these eight functions *primitive operations*. Note that circular rotations and right shifts are not T-functions, but left shifts can be expressed as multiplication by a power of 2 and thus they are T-functions. Since the composition of T-functions is also a T-function, any 'crazy' function which contains arbitrarily many primitive operations is always a T-function.

In order to define multiword mappings f which can be iterated, we have to further extend the notion to functions with the same number m of input and output words. We can represent the multiword input as the following $n \times m$ bit matrix $B^{n\times m}$:

$$x=\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} [x]_{0,n-1} & \cdots & [x]_{0,1} & [x]_{0,0} \\ [x]_{1,n-1} & \cdots & [x]_{1,1} & [x]_{1,0} \\ \vdots & \ddots & \vdots & \vdots \\ [x]_{m-1,n-1} & \cdots & [x]_{m-1,1} & [x]_{m-1,0} \end{pmatrix}. \tag{2}$$

We can now consider the columns of the bit matrix as parallel bit slices with no internal bit order, and say that a multiword mapping is a T-function if all the bits in column i of the matrix of output words can depend only on bits in columns 0 to i of the matrix of input words. In this interpretation it is still true that any composition of primitive operations is a multiword T-function, but some of the proven properties of univariate T-functions (e.g., that all the cycle lengths are powers of 2) are no longer true.

An alternative definition of multiword T-functions is to concatenate all the input words into one long word, to concatenate all the output words into one long word, and then to use the standard univariate definition of a T-function in order to limit which input bits can affect which output bits. If we denote the l input words by $x_u, x_v, \ldots$, then we define the single logical variable x by

$$x=(x_u,\ldots,x_w)=([x]_{n(l-1)+(n-1)},\ldots,[x]_{n(l-1)},\ldots [x]_{n-1},\ldots,[x]_0). \tag{3}$$

Note that in this interpretation $f(x)=(f_u,f_v)=(x_u+x_v,x_v)$ is a T-function, but the very similar $f(x)=(f_u,f_v)=(x_u,x_u+x_v)$ is not a T-function, and thus we cannot compose primitive operations in an arbitrary way. On the other hand, we can obtain many new types of T-functions in which low-order words can be manipulated by non-primitive operations (such as cyclic rotation) before we use them to compute higher order output words.

Our actual definition of multiword T-functions combines and generalizes these two possible interpretations. Let x be an $nl\times m$ bit matrix ($\mathbb{B}^{nl\times m}$):

$$\begin{pmatrix} [x]_{0,n(l-1)+(n-1)} & \cdots & [x]_{0,n(l-1)+1} & [x]_{0,n(l-1)} & \cdots & [x]_{0,n-1} & \cdots & [x]_{0,0} \\ [x]_{1,n(l-1)+(n-1)} & \cdots & [x]_{1,n(l-1)+1} & [x]_{1,n(l-1)} & \cdots & [x]_{1,n-1} & \cdots & [x]_{1,0} \\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ [x]_{m-1,n(l-1)+(n-1)} & \cdots & [x]_{m-1,n(l-1)+1} & [x]_{m-1,n(l-1)} & \cdots & [x]_{m-1,n-1} & \cdots & [x]_{m-1,0} \end{pmatrix}. \tag{4}$$

We consider it as an $m\times l$ matrix of n bits words:

$$x=\begin{pmatrix} x_{0,u} & \cdots & x_{0,w} \\ \vdots & \ddots & \vdots \\ x_{m-1,u} & \cdots & x_{m-1,w} \end{pmatrix}.$$

We concatenate the l words in each row into a single logical variable, and then consider the collection of the m long variables as the inputs to the T-function. Finally, we allow the bits in column i of the output matrix to depend only on bits in columns $0,\ldots,i$ in the input matrix.

To demonstrate this notion, consider the following mapping over 4-tuples of words:

$$f(x) = \begin{pmatrix} x_{0,u} + x_{1,u} x_{0,v} & x_{0,v} + x_{1,v} \\ x_{0,u} - x_{1,u}(x_{1,v} \Rsh 1) & x_{0,v} \oplus x_{1,v} \end{pmatrix}.$$

This is a valid T-function under our general multiword definition even though it contains the non-primitive right shift operation $\Rsh 1$.

3 Bit slice analysis and invertibility

The main tool we use in order to study the invertibility of T-functions is bit slice analysis. Its basic idea is to define the mapping from $[x]_i$ to $[f(x)]_i$ by abstracting out the complicated dependency on $[x]_{0\ldots i-1}$ via the notion of parameters. For example, the size of the explicit description of the mapping $[f(x)]_i = \phi([x]_0, \ldots, [x]_i)$ in the function $f(x) = x + (x^2 \vee 5)$ grows exponentially with i, but it can be written as $[f(x)]_i = [x]_i \oplus \alpha_i$, where α_i is some function of $[x]_0, \ldots, [x]_{i-1}$ (that is, a parameter). By using this parametric representation we can easily prove the invertibility of the mapping by induction on i, since if we already know bits 0 to $i-1$ of the input x and bit i of the output $f(x)$, we can (in principle) calculate the value of α_i and thus derive in a unique way bit i of the input. Intuitively, this is the same technique we use in order to solve a triangular system of linear equations, except that in our case the explicit description of α_i can be extremely complicated, and thus we do not use this technique as a real inversion algorithm for $f(x)$, but only in order to prove that this inverse is uniquely defined.

The main observation in [5] was that such an abstract parametric representation can be easily derived for any composition of primitive operations by the following recursive definition, in which i can be any bit position except zero:

$$\begin{aligned}
[xy]_0 &= [x]_0 \wedge [y]_0 \\
\left[x \underset{\oplus}{\overset{+}{-}} y\right]_0 &= [x]_0 \oplus [y]_0 \\
\left[x \overset{\wedge}{\vee} y\right]_0 &= [x]_0 \overset{\wedge}{\vee} [y]_0 \\
[xy]_i &= [x]_i \, \alpha_{[y]_0} \oplus \alpha_{[x]_0} [y]_i \oplus \alpha_{xy} \\
\left[x \overset{+}{-} y\right]_i &= [x]_i \oplus [y]_i \oplus \alpha_{x \pm y} \\
\left[x \overset{\oplus}{\underset{\vee}{\wedge}} y\right]_i &= [x]_i \overset{\oplus}{\underset{\vee}{\wedge}} [y]_i
\end{aligned} \tag{5}$$

To demonstrate this technique, consider our running example: $\left[x + (x^2 \vee 5)\right]_0 = [x]_0 \oplus \left[x^2 \vee 5\right]_0 = [x]_0 \oplus 1$ and, for $i > 0$, $\left[x + (x^2 \vee 5)\right]_i = [x]_i \oplus (\left[x^2\right]_i \vee [5]_i) \oplus \alpha_{x+(x^2 \vee 5)} = [x]_i \oplus (([x]_i \, \alpha_{[x]_0} \oplus \alpha_{[x]_0} [x]_i \oplus \alpha_{x^2}) \vee [5]_i) \oplus \alpha_{x+(x^2 \vee 5)} = [x]_i \oplus \alpha$.

This invertibility test can be easily generalized to the multivariate case (2). Let us show an example of such a construction. We start from an arbitrary non singular matrix which denotes a possible bit slice mapping, such as:

$$\begin{matrix} 1 & \alpha \\ 0 & 1 \end{matrix}$$

We can add to this linear mapping an affine part (where α, β, and γ are arbitrary parameters) and get the following bit slice structure:

$$\begin{pmatrix} [x_0]_i \\ [x_1]_i \end{pmatrix} \to \begin{pmatrix} [x_0]_i \oplus \alpha\,[x_1]_i \oplus \beta \\ [x_1]_i \oplus \gamma. \end{pmatrix} \tag{6}$$

It is easy to check that for $i > 0$ the i-th bit slice of the following mapping matches (6):

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \to \begin{pmatrix} x_0 + (x_0^2 \wedge x_1) \\ x_1 + x_0^2. \end{pmatrix}$$

Unfortunately, the least significant bit slice of this mapping is not invertible:

$$\begin{pmatrix} [x_0]_0 \\ [x_1]_0 \end{pmatrix} \to \begin{pmatrix} [x_0]_0 \oplus [x_0]_0\,[x_1]_0 \\ [x_1]_0 \oplus [x_0]_0\,. \end{pmatrix}$$

So we have to apply a little tweak to fix it:

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \to \begin{pmatrix} x_0 + ((x_0^2 \wedge x_1) \vee 1) \\ x_1 + x_0^2. \end{pmatrix}$$

The reader may get the impression that the bit slice mappings of invertible functions are always linear. From (5) it is easy to see that every expression which uses only $\oplus$, $+$, $-$ and $\times$ has linear i-th bit slice, but in general this is not true.

4 The single cycle property

A T-function has the single cycle property if its repeated application to any initial state goes through all the possible states. Let us recall the basic results from [6] in the univariate case. Invertibility is a prerequisite of the single cycle property. If a T-function has a single cycle modulo 2^k then it has a single cycle modulo 2^{k-1}. If a T-function has a cycle of length l modulo 2^{k-1} then modulo 2^k it has either a cycle of length $2l$ or two cycles of length l. Taking into account the fact that modulo 2^1 a function has either one cycle of length two or two cycles of length one we can conclude that the size of any cycle of a T-function is always a power of 2.

In the univariate case a bit slice of an invertible T-function has the form $[f(x)]_i = [x]_i \oplus \alpha$. From (5) it follows that $f(x)$ has one of the following forms: $f_1(x) = x \oplus r_1(x)$, $f_2(x) = x + r_2(x)$ or $f_3(x) = xr_3(x)$, where the r_i are parameters (in the case of multiplication additionally we need $[r_3]_0 = 1$). It is easy to see that $[f_3(x)]_0 = [x]_0\,[r_3(x)]_0 = [x]_0$, that is it has two cycles modulo 2 and so it can not form a single cycle modulo 2^n. So, a single cycle function has either[1] the first or the second form. In order to analyse the cycle structure of these forms the following definitions of *even* and *odd* parameters were introduced.

[1] Note that there is no *exclusive* or here since every function can be represented in both forms, for example $x + 1 = x \oplus (x \oplus (x + 1))$.

Suppose that $r(x)$ is a parameter, that is $r(x) = r(x + 2^{n-1}) \pmod{2^n}$. So, $r(x) = r(x + 2^{n-1}) + 2^n b(x) \pmod{2^{n+1}}$. Consider

$$B[r, n] = 2^{-n} \sum_{i=0}^{2^{n-1}-1} (r(i + 2^{n-1}) - r(i)) \pmod 2 = \bigoplus_{i=0}^{2^{n-1}-1} b(i). \qquad (7)$$

The parameter is called *even* if $B[r, n]$ is always zero, and *odd* if $B[r, n]$ is always one[2].

Let us give several examples of even parameters:

- $r(x) = C$, where C is an arbitrary constant ($r(x) = r(x + 2^{n-1})$ and so, $b = 0$ and $B = 0$);
- $r(x) = 2x$ ($r(x + 2^{n-1}) = r(x) + 2^n \pmod{2^{n+1}}$, so $b(x) = 1$ and B is even as long as 2^{n-1} is even, that is for $n \geq 2$);
- $r(x) = x^2$ ($r(x + 2^{n-1}) = r(x) + 2^n x + 2^{2(n-1)}$, so $b(x) = [x]_0$ and B is even for $n \geq 3$);
- $r(x) = 4g(x)$, where $g(x)$ is an arbitrary T-function ($r(x + 2^{n-1}) - r(x) = 4(g(x + 2^{n-1}) - g(x)) = 0 \pmod{2^n}$);
- $r(x) = r'(x) \overset{+}{\underset{\oplus}{-}} r''(x)$, where r' and r'' are simultaneously even or odd parameters ($B = B' \oplus B''$).
- $r(x) = r'(x) \vee C$, where C is an arbitrary constant and $r'(x)$ is an even parameter (if $[C]_i = 0$ then $[r(x)]_i = [r'(x)]_i$, and if $[C]_i = 1$ then $[r(x)]_i = [C]_i$, so in both cases $[r(x)]_i$ is the same as for some even parameter.)

The following theorem was proved in [6]:

Theorem 2. *Let N_0 be such that $x \to x + r(x) \bmod 2^{N_0}$ defines a single cycle and for $n > N_0$ the function $r(x)$ is an* even *parameter. Then the mapping $x \to x + r(x) \bmod 2^n$ defines a single cycle for all n.*

We can use our running example of $f(x) = x + (x^2 \vee C)$ to demonstrate this theorem. If the binary form of C ends with $\ldots 1{}^0_1 1$, then $C = 5, 7 \pmod 8$, and $x^2 \vee C = C \pmod 8$ is an odd constant modulo 2^3 so $x + C$ has a single cycle modulo 2^3. In addition, x^2 is an even parameter for $n \geq 3$, and this is not affected by 'or'ing it with an arbitrary constant. In [6] it was shown that $x \to x + (x^2 \vee C)$ is the smallest nonlinear expression which defines a single cycle, in other words there is no nonlinear expression which defines a single cycle and consists of less than three operations.

Another important class of single cycle mappings is $f(x) = 1 + x + 4g(x)$ for an arbitrary T-function $g(x)$. It turns out that `x86` microprocessors have an instruction which allows us to calculate $1 + x + 4y$ with a single instruction[3] and

[2] Note that in the general case B is a function of n, and thus the parameter can be neither even nor odd. We often relax these definitions by allowing exceptions for small n such as 1 or 2.

[3] The `lea` (load effective address) instruction makes it possible to calculate any expression of the form $C + R_1 + kR_2$, where C is a constant, R_1 and R_2 are registers and k is a power of two. Its original purpose was to simplify the calculation of addresses in arrays.

thus the single cycle mapping $f(x) = 1 + x + 4x^2$ can be calculated using only two instructions.

Odd parameters are less common and harder to construct. Their main application is in mappings of the form $x \oplus r(x)$:

Theorem 3. *Let N_0 be such that $x \to x \oplus r(x) \bmod 2^{N_0}$ defines a single cycle and for $n > N_0$ the function $r(x)$ is an* odd *parameter. Then the mapping $x \to x \oplus r(x) \bmod 2^n$ defines a single cycle for all n.*

Proof. Let us prove this by induction: suppose that the mapping defines a single cycle modulo 2^n and we are going to prove that it defines a single cycle modulo 2^{n+1}. Since there are only two possible cycle structures modulo 2^{n+1} (a single cycle of size 2^{n+1} or two cycles of size 2^n) we will prove that the second case is impossible, that is $\left[x^{(0)}\right]_n \neq \left[x^{(2^n)}\right]_n$ at least for one x. Recall that $[x]_n$ is the most significant bit of x modulo 2^{n+1}. Let $x^{(0)} = 0$, since r is a parameter it follows that $r(i) = r(i + 2^n) \pmod{2^{n+1}}$, and so $\left[x^{(2^n)}\right]_n = \left[r(x^{(0)}) \oplus \ldots \oplus r(x^{(2^n-1)})\right]_n = \bigoplus_{i=0}^{2^n-1} [r(i)]_n =$

$\bigoplus_{i=0}^{2^{n-1}-1}(\left[r(i + 2^{n-1})\right]_n \oplus [r(i)]_n) = \bigoplus_{i=0}^{2^{n-1}-1}(\left[r(i + 2^{n-1}) - r(i)\right]_n) = 1.$

Here we use the fact that $\left[r(i + 2^{n-1})\right]_n - [r(i)]_n = \left[r(i + 2^{n-1}) - r(i)\right]_n$.

Recently the related notions of measure preservation and ergodicity of compatible functions over p-adic numbers were independently studied by Anashin [1].[4] His motivation was mathematical rather than cryptographic, and he used different techniques. In order to study if a T-function is invertible (respectively, has a single cycle property) he tried to represent it as $f(x) = d + cx + pv(x)$ (respectively, $f(x) = c + rx + p(v(x+1) - v(x))$), or to represent it as Mahler interpolation series, or to use the notion of uniform differentiability. The first characterization is the most general (that is $v(x)$ can be any T-function) and complete (he proved that every invertible (respectively, a single cycle function) can be represented in this form. For example, it follows that there exists $v_f(x)$, such that $f(x) = x + (x^2 \vee 5) = 1 + x + 2(v_f(x+1) - v_f(x))$ and in order to prove that $f(x)$ defines a single cycle it is enough to find such a function $v_f(x)$. This example shows that this criterion is not that good in practice in checking properties of a given function but it allows us to construct arbitrary complex functions with needed properties. For the second approach any function f can be represented as a Mahler interpolation series $\sum_{i=0}^{\infty} a_i \left(\frac{x(x-1)\cdots(x-i+1)}{i!}\right)$. It turns out, for example, that a T-function is invertible if and only if $\|a_1\|_2 = 1$ and $\|a_i\|_2 \leq 2^{-\lfloor \log_2 i \rfloor - 1}$, for $i = 2, 3, \ldots$. This is used to prove theoretical results similar to the previous one, but once again for practical purposes it is usually hard to represent a given function as a Mahler series. The uniformly differentiable[5] func-

[4] This could be translated to our terminology as follows: *measure preservation* — invertibility, *ergodicity* — a single cycle property, *compatible* — T-function. To simplify reading we will continue to use our terminology, but an interested reader who will refer to his paper should keep in mind this “dictionary”.

[5] It is exactly the same concept as the usual notion of uniform differentiability of real functions, but with respect to the p-adic distance.

tions allow us to use Hensel lifting. However, there are several obstacles to the application of this theorem in practice. First of all, it is not always easy to tell if a given function is uniformly differentiable. Consider, for example, $x + (x^2 \vee 5)$: $\vee$ is not a differentiable operation but according to [2] in this particular case $(x+(x^2 \vee 5))' = (x+x^2+5-(x^2 \wedge 5))' = 1+2x+2x(u \wedge 5)|'_{u=x^2}$, but $(u \wedge 5)' = 0$ for $\|h\|_2 \leq \frac{1}{8}$, and so the whole expression is uniformly differentiable. Unfortunately, this trick does not work for the general case $x+(x^2 \vee C)$. Moreover, there are invertible mappings which are not uniformly differentiable. The second obstacle is how to find N_0. In the very restricted case of polynomials (expressions which use only $+$, $-$ and $\times$) it is possible to calculate this number in advance, but this is not possible even if we add only $\oplus$: we found a family of functions $f_i(x)$ (which use only $+$, $-$, and $\oplus$) such that for every N there is $N_0 \geq N$, i_{N_0} such that $f_{i_{N_0}}$ has a single cycle modulo 2^{N_0} but not modulo 2^{N_0+1}. In particular, if $f_0 = x - 1$, $f_i = ((f(i-1, x) \oplus x) + x) \oplus (x + x)$ then $N_0 = i + 2$, $i = 2^r$.

Both Anashin's techniques and our techniques can be used to completely characterize all the univariate polynomial mappings modulo 2^n which are invertible with a single cycle:

Theorem 4. *A polynomial $P(x) = \sum_{i=0}^{d} a_i x^i$ is invertible modulo any 2^n if and only if it is invertible modulo 4, and it has a single cycle modulo any 2^n if and only if it has a single cycle modulo 8.*

Proof. From (5) it follows[6] that $[P(x)]_0 = [a_0]_0 + [(a_1 + \cdots + a_d)]_0 [x]_0$ and $[P(x)]_i = [a_1]_0 [x]_i \oplus \bigoplus_{\text{odd } k \geq 3} [a_k]_0 [x]_0 [x]_i \oplus \alpha$. Since a T-function is invertible if and only if each bit slice is invertible, the following conditions are necessary and sufficient for the invertibility of a polynomial: $[(a_1 + \cdots + a_d)]_0 = 1$, $[a_1]_0 = 1$ and $\bigoplus_{\text{odd } k \geq 3} [a_k]_0 = 0$. In order to prove the single cycle property let us represent the polynomial in the following form: $P(x) = x+P'(x)$. Since in order to generate a single cycle $P(x)$ should be invertible it follows that $[(a'_1 + \cdots + a'_d)]_0 = 0$, $[a'_1]_0 = 0$ and $\bigoplus_{\text{odd } k \geq 3} [a'_k]_0 = 0$. Now we need to prove that $P'(x)$ is an even parameter, that is, that $\bigoplus_{i=0}^{2^{n-1}-1} b(i) = 2^{-n} \sum_{i=0}^{2^{n-1}-1} (P'(i + 2^{n-1}) - P'(i)) \pmod 2 = 0$. Let us do it separately for different parts of P'. It is easy to show that a_0, $a_1 x$ with even a_1 and $a_k x^k$ for even k are even parameters for $n \geq 3$. Let us prove that the sum of odd powers is also an even parameter if $\sum_k [a_k]_0 = 0 \bmod 2$ and $n \geq 3$: $2^{-n} \sum_{i=0}^{2^{n-1}-1} \sum_{\text{odd } k \geq 3} (a_k (i + 2^{n-1})^k - a_k i^k) = 2^{-n} \sum_i \sum_k a_k k i^{k-1} 2^{n-1} + \sum_i \sum_k a_k (\frac{k(k-1)}{2} i^{k-2} 2^{2(n-1)-n} + \cdots) = 0 \pmod 2$. So, if $P(x)$ defines a single cycle modulo 8 then it defines a single cycle modulo any 2^n. On the other hand if $P(x)$ does not define a single cycle modulo 8 then it does not define a single cycle modulo any $2^n \geq 8$.

It is easy to verify that exactly 1/8 of all the polynomials are invertible, and 1/64 of all the polynomials have a single cycle, and thus it is easy to pick random polynomials with these properties. Typical examples of quadratic single cycle polynomials are $f(x) = (x + 1)(2x + 1)$ and $f(x) = 6x^2 - x + 1$.

[6] See [5] for details

Our goal now is to construct invertible multiword T-functions whose iteration defines a single cycle. Let us start with T-functions of type (2), which map m parallel input words to m parallel output words. We would like to construct a mapping

$$\begin{pmatrix} x_0 \\ \vdots \\ x_{m-1} \end{pmatrix} \to \begin{pmatrix} f_0(x_0, \ldots, x_{m-1}) \\ \vdots \\ f_{m-1}(x_0, \ldots, x_{m-1}) \end{pmatrix}$$

which is invertible and has the maximum possible period of 2^{mn}. Several simple constructions can be easily shown to be impossible. For example:

Theorem 5. *No T-mapping of the form* $(x_0, x_1) \to (f_0(x_0), f_1(x_0, x_1))$ *can have a period of* 2^{2n}.

Proof. Suppose there is a mapping $(x_0, x_1) \to (f_0(x_0), f_1(x_0, x_1))$, where f_0 and f_1 are T-functions, such that it has a period of size $P = 2^{2i}$ modulo 2^i. This means that, for example, $f^{(P)}(0,0) = (0,0)$ and $\forall p < P$, $f^{(p)}(0,0) \neq (0,0)$. Let $p = 2^{2(i-1)}$. Since the period of the mapping modulo 2^{i-1} is at most $2^{2(i-1)}$ it follows that $f^{(p)}(0,0) \neq (0,0)$ if and only if the same holds for the most significant bits, but since f_0 depends only on x_0 modulo 2^i the period of f_0 is at most 2^i and so (for sufficiently large i) the most significant bit of $f_0^{(p)} = f_0^{(2p)} = f_0^{(3p)}$ and since the most significant bit of f_1 can assume only two possible values it follows that either $f^{(p)} = f^{(2p)}$ or $f^{(p)} = f^{(3p)}$ which is a contradiction.

Also it can be shown that it is impossible to obtain a single cycle function of this type from slice-linear mappings, which are T-functions which have the following form: $[f_j]_i = \bigoplus_{k=0}^{m-1} [C_{j,k}]_i \, [x_k]_i \oplus \alpha_{j,i}$, where $(C_{i,j})$ is a constant invertible matrix and $m > 2$.

Note that the construction is trivial in the two extreme cases of $m = 1$ and $n = 1$. In the univariate case ($m = 1$) the answer is given by theorem 3: $x \to x \oplus \alpha$, where α is an odd parameter. The case of $n = 1$ is also simple because every function is then a T-function, and it is easy to define a counting transformation such as $f_i = x_i \oplus (x_0 \wedge \cdots \wedge x_{i-1})$ which goes through all the states in the natural order $0 \ldots 00 \to 0 \ldots 01 \to 0 \ldots 10 \to \cdots \to 1 \ldots 11 \to 0 \ldots 00$. Let us combine these two cases:

$$f_i(x_0, \ldots, x_{m-1}) = x_i \oplus (\alpha_i(x_0, \ldots, x_{m-1}) \wedge x_0 \wedge \cdots \wedge x_{i-1}), \tag{8}$$

where each α_i is an odd parameter, that is

$$\bigoplus_{(x_0,\ldots,x_{m-1})=(0,\ldots,0)}^{(2^n-1,\ldots,2^n-1)} [\alpha_i(x_0, \ldots, x_{m-1})]_n = 1$$

and $[\alpha_i]_0 = 1$. We can now prove:

Theorem 6. *The mapping defined by (8) defines a single cycle of length* 2^{mn}.

Proof. Let us prove it by induction. For $n = 1$ we know that it is true. Suppose that it is a single cycle modulo 2^n and let us prove this for 2^{n+1}. Suppose without loss of generality that $(x_0, \ldots, x_{m-1})^{(0)} = (0, \ldots, 0)$. From the induction hypothesis it follows that $\{(x_0, \ldots, x_{m-1})^{(i)} \bmod 2^n\}_{i=0}^{2^{mn}}$ contains all the possible tuples, so $\left[x_0^{(2^{mn})}\right]_n = x_0^{(0)} \oplus \bigoplus \alpha_0 = 1$, more generally, $\left[x_0^{(l+2^{mn})}\right]_n = \left[x_0^{(l)}\right]_n \oplus 1$, so

$$\bigoplus_{(x_0,\ldots,x_{m-1})=(0,\ldots,0)}^{(2^{n+1}-1,2^n-1,\ldots,2^n-1)} [\alpha_1(x_0, \ldots, x_{m-1})]_n \wedge [x_0]_n = \bigoplus_{(x_0,\ldots,x_{m-1})=(0,\ldots,0)}^{(2^n-1,2^n-1,\ldots,2^n-1)} [\alpha_1(x_0, \ldots, x_{m-1})]_n .$$

So, if we consider the next variable x_1: $\left[x_1^{(2^{mn+1})}\right]_n = x_1^{(0)} \oplus \bigoplus \alpha_1 \wedge x_0 = x_1^{(0)} \oplus 1$. Using similar arguments, we can prove that $\left[x_{m-1}^{(2^{mn+(m-1)})}\right]_n = \left[x_{m-1}^0\right]_n \oplus 1$, that is modulo 2^{n+1} the period is $2^{mn+m} = 2^{m(n+1)}$.

In order to use this theorem we need an odd parameter. We know that $f(x) = x + r(x)$ defines a single cycle if $r(x)$ is an even parameter. We also know that every such function can be represented as $f(x) = x \oplus s(x)$, where $s(x)$ is an odd parameter. So, for any even parameter $r(x)$ the following expression $s(x) = (x + r(x)) \oplus x$ is an odd parameter. For example, if $r(x) = x^2 \vee 5$, then $s(x) = (x + (x^2 \vee 5)) \oplus x$ is an odd parameter. Note that if $f(x)$ is invertible then $[r(x)]_0 = 1$ and so $[s(x)]_0 = [r(x)]_0 = 1$. To construct an odd parameter with m variables we need the following lemma:

Lemma 1.

$$\bigoplus_{t=0}^{2^n} \alpha(t) = \bigoplus_{(x_0,\ldots,x_{m-1})=(0,\ldots,0)}^{(2^n-1,\ldots,2^n-1)} \alpha(x_0 \wedge \cdots \wedge x_{m-1})$$

Proof. In order to prove the lemma it is sufficient to prove that for every t there is an *odd* number of tuples $(x_0, \ldots, x_{m-1})$ such that $x_0 \wedge \cdots \wedge x_{m-1} = t$. In fact, we can directly calculate this number: if t contains k zeros then there are $(2^m - 1)^k$ such tuples, which is an odd number.

For example, the following mapping defines a single cycle for any m and n:

$$f_i(x_0, \ldots, x_{m-1}) = x_i \oplus (s(x_0 \wedge \cdots \wedge x_{m-1}) \wedge x_0 \wedge \cdots \wedge x_{i-1}), \tag{9}$$

where $s(t) = (t + (t^2 \vee 5)) \oplus t$. Note that to evaluate this mapping for each particular $(x_0, \ldots, x_{m-1})$ we need one squaring, one addition and $3m$ bitwise operations, or $3m + 2$ operations in total. Unfortunately, this mapping has the property that during update each bit of x_{m-1} is changed with probability $2^{-(m+1)}$ instead of $\frac{1}{2}$ as expected for a random mapping. To avoid this we can add any even parameter[7] with zero in the least significant bit, simplify $s(t)$ and obtain, for example,

[7] Note that in the multivariate case every parameter which does not use all the variables is even.

the following mapping:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \to \begin{pmatrix} x_0 \oplus s & \oplus (x_1^2 \wedge M) \\ x_1 \oplus (s \wedge a_0) & \oplus (x_2^2 \wedge M) \\ x_2 \oplus (s \wedge a_1) & \oplus (x_3^2 \wedge M) \\ x_3 \oplus (s \wedge a_2) & \oplus (x_0^2 \wedge M) \end{pmatrix}, \tag{10}$$

where $a_0 = x_0$, $a_1 = a_0 \wedge x_1$, $a_2 = a_1 \wedge x_2$, $a_3 = a_2 \wedge x_3$, $s = (a_3 + C) \oplus a_3$, C is any odd constant and $M = 1 \ldots 1110_2$ or we can enhance the inter-variable mixing with the following mapping:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \to \begin{pmatrix} x_0 \oplus s & \oplus 2x_1x_2 \\ x_1 \oplus (s \wedge a_0) & \oplus 2x_2x_3 \\ x_2 \oplus (s \wedge a_1) & \oplus 2x_3x_0 \\ x_3 \oplus (s \wedge a_2) & \oplus 2x_0x_1 \end{pmatrix} \tag{11}$$

During the iteration of these mappings each bit is changed in approximately one half of the cases. Each mapping requires 24 operations to obtain a cycle of size 2^{256}.

We next analyse mappings of type (3), which consider multiple words as concatenated parts of a single multi-precision logical variable x, for example, $x = 2^{3n}x_a + 2^{2n}x_b + 2^n x_c + x_d$. In this case our running example $x \to x + (x^2 \vee C)$ can be represented as $(x_a, x_b, x_c, x_d) \to (f_a(x_a; x_b, x_c, x_d), \ldots, f_d(x_d))$ with appropriate f_a, f_b, f_c and f_d, but the squaring of a $4n$-bit word would be rather inefficient on an n-bit processor. Note that there is no requirement that the whole mapping $x \to f(x)$ has a simple interpretation as a mapping of a $4n$ bit word although we need this interpretation to prove the single cycle property.

Let us start with the simplest single cycle mapping $x \to x + 1$. It can be implemented as follows: $(x_a, x_b, x_c, x_d) \to (x_a + \kappa_b, x_b + \kappa_c, x_c + \kappa_d, x_d + 1)$, where κ_d is the carry (overflow) from x_d, κ_c is the carry from x_c, et cetera. Many contemporary processors (including x86 and SPARC) have an operation `adc` (addition with carry) $(x, y, c) \to (z, c')$, where $z = (x + y + c) \bmod 2^n$ and $c' = (x + y + c \geq 2^n)$, so the addition of κ can be done at no additional cost[8] if the mapping has the following form: $(x_a, x_b, x_c, x_d) \to (x_a + f_a + \kappa_b, x_b + f_b + \kappa_c, x_c + f_c + \kappa_d, x_d + f_d + 1)$. The only restriction on each f is that it should be an even parameter. For example,

$$\begin{pmatrix} x_d \\ x_c \\ x_b \\ x_a \end{pmatrix} \to \begin{pmatrix} x_d + (s_d^2 \vee C_d) \\ x_c + (s_c^2 \vee C_c) + \kappa_d \\ x_b + (s_b^2 \vee C_b) + \kappa_c \\ x_a + (s_a^2 \vee C_a) + \kappa_b, \end{pmatrix} \tag{12}$$

[8] This is not always true. For example, on `Intel Pentium 4` processor the ordinary `add` instruction has the latency (the number of clock cycles that are required for the execution core to complete the execution of all of the μops that form a IA-32 instruction) 0.5 and the throughput (the number of clock cycles required to wait before the issue ports are free to accept the same instruction again) 0.5, but `adc` has the latency 8 and the throughput 3 [4].

where $s_d = x_d$, $s_c = s_d \oplus x_c$, $s_b = s_c + x_b$, $s_a = s_b \oplus x_a$, C_a, C_b, C_c are odd constants[9] and C_d ends with $\dots 1{}^0_1 1_2$.

This mapping uses 15 operations to obtain a cycle of size 2^{256}. So, it can be much faster than the previous example but only on processors that have the `adc` instruction. If this instruction has to be emulated[10] this mapping can be slower. Note that almost all high level programming languages do not have such an instruction so it has to be emulated or to be written in assembly language.

The number of operations is not usually a good predictor of the speed of the implementation, since on modern microprocessors several operations can be done in parallel, use different number of clocks, etc. We should take into account that an update function is not the complete generator. There is also an output function which produces the output bits given the internal state. Since the least significant bits of the state are repeated in small cycles the simplest output function is the one that gives away the most significant bits of the state. It seems that it is easier to implement such a function for the second mapping since x_a and x_b form the most significant part of the state, but their least significant bits are also weak since they depend only of the least significant bits of x_i and a single bit carry.[11] One solution is to give away the most significant part of x_a (32 bits) or use a more sophisticated output function and give away more bits thus raising the ratio of the output bits per operation. For example, $O_1 = ((x_a \Rsh \frac{n}{2}) \oplus x_b)(((x_c \Rsh \frac{n}{2}) \oplus x_d) \vee 1)$, where $\Rsh$ could be substituted with circular rotation if the target microprocessor supports it, uses only five operations and doubles the size of the output. It can be shown that O_1 has maximal period and produces each possible value with the same probability (since it is an invertible mapping of x_b for fixed x_a, x_c and x_d). Note that we give here only examples of the possible mappings and output functions, but in order to construct a secure stream cipher they have to be subjected to a lengthy and thorough cryptanalysis.

5 Experimental results

We tested the actual execution speed of our mappings on an IA-32 machine. We used a standard PC with a 1.7 GHz Pentium 4 processor with 256KB of cache. In each experiment we encrypted one gigabyte of data by encrypting 10^4 times a buffer which is 10^5 bytes long (this ensured that the data was prefetched into L2 cache). The tests were done on an otherwise idle Linux system. To calculate the overhead produced by memory access we "encrypted" the buffer by xoring

[9] Note that the only purpose of C_a, C_b and C_c is to make parameters out of $x_\cdot^2$, that is mask the least significant bit. The evenness of the parameter is guaranteed since n from (7) is larger than 3.

[10] There are basically two ways to emulate it: check if $a+b \geq a$ or $a+b \geq b$ for unsigned a and b or, usually faster since conditions break pipeline, to set $n = 63$ and use the most significant bit of the sum as $\kappa_{\cdot\cdot}$. The second approach adds two operation for each addition of $\kappa_\cdot$ and gives 21 operations overall with reduced word size.

[11] It is possible to change the definitions of s_a,s_b,s_c,s_d to incorporate right shifts or rotations, but this will also increase the calculation time.

it with a 32 bit constant. It took 0.42 seconds, and we did *not* subtract it from our actual results.

In this paper we propose a general approach rather than a concrete stream cipher, and thus in our performance tests we experimented with many different state sizes, update mappings and output functions. As a typical example, we used 256 bit states defined as four words of length $n = 64$ and updated them by (11). We wanted to use a simple output function which is just the top half of each x_i, but we discovered that this variant is vulnerable to a (theoretical) attack in which the attacker guesses the 17 least significant bits of each x_i (i.e., a total of 68 guessed bits) and searches in 2^{34} bytes of data for places where two of the x_i end with 17 zeroes and thus their product ends with 34 zeros. To protect against this attack, we slightly modified the state update function to:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \to \begin{pmatrix} x_0 \oplus s & \oplus \ (2(x_1 \vee C_1)x_2) \\ x_1 \oplus (s \wedge a_0) & \oplus \ (2x_2(x_3 \vee C_3)) \\ x_2 \oplus (s \wedge a_1) & \oplus \ (2(x_3 \vee C_3)x_0) \\ x_3 \oplus (s \wedge a_2) & \oplus \ (2x_0(x_1 \vee C_1)) \end{pmatrix}, \tag{13}$$

where C_1 and C_2 are constants with several ones in the least significant half, for example, $C_1 = \mathtt{12481248}_{16}$ and $C_3 = \mathtt{48124812}_{16}$. To take advantage of the SSE2 instruction set of Pentium 4 processors (which contains 128-bit integer instructions that allow us to operate on two 64-bit integers simultaneously), we run two generators ((x_0, x_1, x_2, x_3) and (x'_0, x'_1, x'_2, x'_3)) in parallel and the output is generated by $(x_0 \upharpoonright 32) \oplus x_1$, $(x_2 \upharpoonright 32) \oplus x_3$, $(x'_0 \upharpoonright 32) \oplus x'_1$ and $(x'_2 \upharpoonright 32) \oplus x'_3$. Since each command operates on a pair (x_i, x'_i) we can run two generators instead of one at no additional cost. With this optimization, the complete encryption operation required only 1.56 seconds per gigabyte, and thus the experimentally verified encryption speed was approximately 5.13 gigabits/second.

To put our results in perspective, the reader should consider the RC4 stream cipher, which is one of the fastest software oriented ciphers available today. The web site of Crypto++ contains benchmarks for highly optimized implementations of many well known cryptographic algorithms [3]. For RC4 it quotes a speed of 110 megabyte per second on a 2.1 GHz Pentium 4 processor. This can be scaled to $1000/110 \times 1.7/2.1 = 11.2$ seconds required to encrypt a gigabyte of data on a 1.7GHz processor, which is almost an order of magnitude slower than the speed we obtain with our approach.

References

1. V. Anashin, "Uniformly Distributed Sequences of p-adic integers, II". Available from `http://www.arxiv.org/ps/math.NT/0209407`.
2. V. Anashin, private communication.
3. Crypto++ 5.1 Benchmarks: `http://www.eskimo.com/~weidai/benchmarks.html`
4. "IA-32 Intel Architecture Optimization Reference Manual". Available from `http://www.intel.com/design/pentium4/manuals/248966.htm`
5. A. Klimov and A. Shamir, "*A New Class of Invertible Mappings*", CHES 2002.
6. A. Klimov and A. Shamir, "*Cryptographic Applications of T-functions*", SAC 2003.