Cryptographic Applications of T-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 in which the i-th bit of the output can depend only on bits $0, 1, \ldots, i$ of the input. All the bitwise machine operations and most of the numeric machine operations in modern processors are T-functions, and their compositions are also T-functions. In this paper we show that T-functions can be used to construct exceptionally efficient cryptographic building blocks which can be used as nonlinear maximal length state transition functions in stream ciphers, as large S-boxes in block ciphers, and as non-algebraic multipermutations in hash functions.

1 Introduction

A function $f(x)$ from an n-bit input to an n-bit output is called a *T-function* (which is short for a *triangular function*) if it does not propagate information from left to right, that is, bit number i of the output (denoted by $[f(x)]_i$, where the LSB is defined as bit number 0 and the MSB is defined as bit number $n-1$) can depend only on bits $j = 0, \ldots, i$ of the input. This definition can be naturally extended to functions that map several n-bit inputs to several n-bit outputs. The special class of T-functions $f(x)$ for which the i-th bit of the output can depend only on bits $j = 0, \ldots, i-1$ of the input are called *parameters* and denoted by α, β, γ, et cetera. We can thus consider each T-function f as a parameterized collection of bit slice mappings from $[x]_i$ to $[f(x)]_i$, in which the parameters describe the dependence of the output bit on the previous input bits. The simplest example of such a representation is the addition of two variables x and y: The i-th bit of the output is the XOR of $[x]_i$, $[y]_i$, and a parameter which describes the carry from the addition of all the previous bit slices. Note that the mappings $x \to 2x \bmod 2^n$ and $x \to x^2 \bmod 2^n$ are T-functions which are also parameters.[1] If $f_0(x), \ldots, f_k(x)$ are parameters then $g(x) = \Psi(f_0(x), \ldots, f_k(x))$ is also a parameter for an arbitrary T-function Ψ.

Simple examples of T-functions are the following machine operations, which are available on almost any contemporary microprocessor with n-bit words: addition ($a+b \bmod 2^n$), subtraction ($a-b \bmod 2^n$), negation ($-a \bmod 2^n$), multiplication ($a \cdot b \bmod 2^n$), bit-wise complementation ($\overline{a}$), xor ($a \oplus b$), and ($a \wedge b$), or

[1] It can be shown that a T-function $f(x)$ is a parameter if and only if $\forall x : 0 \leq x < 2^n \;\; f(x) = f(x + 2^n) \pmod{2^{n+1}}$ and it is easy to check that $2x = 2(x + 2^n) \pmod{2^{n+1}}$ and $x^2 = (x + 2^n)^2 \pmod{2^{n+1}}$ (except for $n = 0$).

$(a \vee b)$. We call these eight machine instructions *primitive operations*. Any univariate or multivariate composition of primitive operations is a T-function, and in particular all the polynomials modulo 2^n are T-functions. Note that left shift could be added to the list since it is equivalent to multiplication by two modulo 2^n, but right shift and circular shift cannot be added to the list since they are not T-functions. The main purpose of this paper is to study the properties of cryptographic building blocks defined by T-functions, and in particular the properties of mappings defined as a composition of a small number of primitive operations which mix boolean and arithmetic operations over 32 or 64 bit words, and contain nonlinear subexpressions such as squaring. Such mappings have very efficient software implementations, and they are more likely to resist cryptanalysis, even though each one of them has to be tested carefully over a long period of time before it can be used as a secure building block in an actual cryptographic design.

2 Invertible T-Functions

The round function of a block cipher has to be invertible, since we have to undo its effect in order to decrypt the ciphertext. Even when we do not wish to apply the inverse operation, it is often desirable to use invertible building blocks (e.g., to avoid collisions in hash functions or to increase the expected cycle length in stream ciphers). In [3] it was shown that it is easy to test the invertibility of mappings which are based on arbitrary compositions of primitive operations. For example, the following three T-functions

$$\begin{aligned} x &\to x + 2x^2 \\ x &\to x + (x^2 \vee 1) \\ x &\to x \oplus (x^2 \vee 1) \end{aligned}$$

require only three primitive operations over n-bit words, and they are invertible for any n. To prove this result, it is necessary and sufficient to prove that for all i the bit slice mappings $[x]_i \to [f(x)]_i$ are invertible, where all the bits of x are fixed except $[x]_i$. This condition can be easily checked by noting that x^2 is a parameter except in its the least significant bit, and the LSB problem can be fixed by either shifting it or "OR"ing it with 1. A simple generalization of this technique can show that multivariate mappings such as

$$\begin{aligned} x &\to x \oplus 2(x \vee y), \\ y &\to (y + 3x^3) \oplus x \end{aligned}$$

are also invertible for any word size n.

3 Invertible T-Functions with a Single Cycle

The problem considered in this section is how to characterize those T-functions which are permutations with a single cycle of length 2^n for any word size n.

The cycle structure of an invertible T-function is particularly important when the function is used as a state transition function in a stream cipher, since in this application we iterate the function a large number of times and do not want the sequence of generated states to be trapped in a short cycle. The fact that we can easily test whether a linear feedback shift register (LFSR) has maximal length is one of the main reasons they are so popular in stream cipher designs, in spite of the cryptographic weaknesses implied by the linearity of the transition function. If a similar theory can be developed for compositions of primitive operations, we can use more complicated (nonlinear, and even non-algebraic) transition functions which are very efficient in software and which are guaranteed to have a maximal cycle length.

One added benefit is that in LFSRs, the existence of one fixed point (the all zero state) is unavoidable, and thus we have to make sure that we do not accidentally derive such an initial state from the random key (e.g., by forcing some of the bits to 1). This is cumbersome, and in some well known stream ciphers (e.g., A5/2) such a protective mechanism even leads to devastating attacks. With appropriately chosen T-functions, we can guarantee that all the states are on a single cycle of length 2^n, and thus there are no bad states which should be avoided during the initialization of the stream cipher.

Not every invertible T-function has a single cycle. Consider, for example, the RC6 mapping $x \to x + 2x^2 \pmod{2^n}$. It is easy to show that it has a very bad cycle structure since any one of the $2^{n/2}$ x values whose bottom half is zero is a fixed point of the mapping. The mapping $x \to x + (x^2 \vee 1)$ is better, but it does not have a single cycle. In [3], we used ad-hoc techniques to show that the mapping $x \to x + (x^2 \vee 5)$ (which also requires only three primitive operations on n-bit words) has a single cycle which contains all the 2^n possible values. In this section we develop a general technique for proving such results for other T-functions.

The cycle structure of T-functions was recently studied by Anashin [1], who used p-adic analysis and infinite power series to prove[2] (in theorem 2.7) that a T-function is invertible if and only if it can be represented in the form $c+x+2v(x)$, and that an invertible T-function defines a single cycle if and only if it can be represented in the form $1 + x + 2(v(x + 1) - v(x))$, where c is some constant and $v(x)$ is some T-function. The sufficiency can be used to construct new single cycle permutations by using a $v(x)$ which is defined by a small number of primitive functions,[3] but the necessity is less useful since in many cases (e.g. the mapping $x + (x^2 \vee 5)$) we know from Anashin's theorem that $v(x)$ exists but it is not clear how to define it in order to show by his method that the mapping has a single cycle.

[2] He uses a completely different terminology of compatible, ergodic and measure preserving functions, so we translate his result to our terminology.

[3] Note, however, that the construction requires two evaluations of $v(x)$ and five additional operations, so even if $v(x) = x^2$, which requires a single operation, the constructed function requires seven primitive operations which is rather inefficient.

In the univariate case, there are only four possible bit slice mappings from one bit to one bit: 0, 1, x, $x \oplus 1$. Since only the last two are invertible and the choice between them can be a parameter depending on the previous bits, a T-function $f(x)$ is invertible if and only if it can be represented in the following form: $[f(x)]_i = [x]_i \oplus \alpha$, where α is a parameter. It is easy to show that the only possible lengths of cycles of a T-function are powers of two, so the sequence $\{x_{i+1} = f(x_i) \bmod 2^n\}$ has period 2^n if and only if $x_i \neq x_{i+2^{n-1}} \pmod{2^n}$. Since for any T-function $x_i = x_{i+2^{n-1}} \pmod{2^{n-1}}$, we can conclude that $f(x)$ defines a single cycle if and only if $[x_i]_{n-1} \neq [x_{i+2^{n-1}}]_{n-1}$, and thus $\bigoplus_{j=0}^{2^{n-1}} \alpha(x_{i+j}) = 1$. The function $\alpha(x)$ is a parameter, that is $\alpha(x + 2^{n-1}) = \alpha(x) \pmod{2^n}$. Moreover, $\{x_i \bmod 2^{n-1}\}$ has period 2^{n-1}, so $\{\alpha(x_{i+j})\}_{j=0}^{2^{n-1}}$ as a set is the same as $\{\alpha(j)\}_{j=0}^{2^{n-1}}$. This proves the following simple characterization:

Theorem 1. *The T-function $f(x)$ defines a single cycle modulo 2^n if and only if $[f(x)]_{n-1} = [x]_{n-1} \oplus \alpha([x]_{0,\ldots,n-2})$ and $\bigoplus_{x=0}^{2^{n-1}-1} \alpha(x) = 1$.*

Once again, this theorem completely characterizes those T-functions whose iteration defines a single cycle, but it is difficult to use it even in the simplest example of the mapping $x \to x + C \pmod{2^n}$, which is known to have a single cycle if and only if C is odd.

Since the general theorem is not very helpful in practice, let us consider an interesting special case of it. 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

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

Let us introduce the notion of *even* and *odd* parameters according to the 0/1 value of B. Note that in the general case B is a function of n, and thus the parameter can be neither even nor odd if

$$\forall N\ \exists n_o > N, n_e > N : \qquad B[r, n_o] = 1,\ B[r, n_e] = 0.$$

Let us give several examples of even parameters. The simplest is an arbitrary constant $r(x) = C$: $r(x) = r(x + 2^{n-1})$ and so, $b = 0$ and $B = 0$. Let $r(x) = 2x$ then $r(x + 2^{n-1}) = r(x) + 2^n \pmod{2^{n+1}}$, so $b(x) = 1$ and B is even as long as 2^{n-1} is even, that is $n \geq 2$. If $r(x) = x^2$ then $r(x+2^{n-1}) = r(x)+2^n x+2^{2(n-1)}$, so $b(x) = [x]_0$ and B is even for $n \geq 3$.

The following function is an example of an odd parameter: $[r(x)]_i = [x]_0 \wedge [x]_1 \wedge \cdots \wedge [x]_{i-1}$. Bit number i of $r(x)$ depends only on bits up to $i - 1$ of x, so it is a T-function and a parameter. Clearly $b(i) \neq 0$ if and only if $i = 1 \ldots 1$, so $B[r, n] = 1$ for $n \geq 1$.

Note that if a and b are even parameters then $a + b$, $a - b$, $a \oplus b$ are also even parameters. Suppose that we know the value of $B[r]$ and let us calculate $B[r \vee C]$, where C is a constant. If $[C]_i = 0$ then $[r(x) \vee C]_i = [r(x)]_i$, so $B[r, i] =$

$B[r \vee C, i]$. If $[C]_i = 1$ then $[r(x) \vee C]_i = 1$, so $b(x) = 0$ and $B[r \vee C, i] = 0$. Combining both cases we could say that if $r(x)$ is an even parameter then $r(x) \vee C$ is also an even parameter for an arbitrary constant C.

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

Proof. Let us prove the theorem by induction. For $n \le N_0$ the mapping defines a single cycle. Suppose that it is true for n (and $n-1$). Consider the sequence $\{x_i\}_{i=0}^{2^{n+1}-1}$, where $x_0 = 0$ and $x_i = x_{i-1} + r(x_{i-1}) \bmod 2^{n+1}$. In particular,

$$x_{2^n} = x_{2^n-1} + r(x_{2^n-1}) = \ldots = x_0 + \sum_{i=0}^{2^n-1} r(x_i) = \sum_{i=0}^{2^n-1} r(x_i) \pmod{2^{n+1}}. \quad (2)$$

From the inductive hypothesis it follows that all the x_i $(i = 0, \ldots, 2^n - 1)$ are different modulo 2^n, so as a set $\{x_i \bmod 2^n\}_{i=0}^{2^n-1} = \{i\}_{i=0}^{2^n-1}$. Using also the fact that $r(x) = r(x + 2^n) \pmod{2^{n+1}}$, we can rewrite the sum as follows:

$$\begin{aligned}\sum_{i=0}^{2^n-1} r(x_i) &= \sum_{i=0}^{2^n-1} r(x_i \bmod 2^n) = \sum_{i=0}^{2^n-1} r(i) \\ &= \sum_{i=0}^{2^{n-1}-1} r(i) + \sum_{i=0}^{2^{n-1}-1} r(i + 2^{n-1}) \pmod{2^{n+1}}.\end{aligned}$$

Let us denote the first sum as S and the second as S'. From the fact that $x_0 \ne x_{2^{n-1}} \pmod{2^n}$, but $x_0 = x_{2^{n-1}} \pmod{2^{n-1}}$ it follows that $x_{2^{n-1}} = 2^{n-1} \pmod{2^n}$. Using also (2) (with n replaced by $n-1$), we can write

$$S = \sum_{i=0}^{2^{n-1}-1} r(i) = \sum_{i=0}^{2^{n-1}-1} r(x_i) = x_{2^{n-1}} = 2^{n-1} \pmod{2^n}.$$

So, there is a constant A, such that $S = A2^n + 2^{n-1} \pmod{2^{n+1}}$. From the fact that $r(x)$ is an even parameter it follows that $S' = S \pmod{2^{n+1}}$, so, $x_{2^n} = S + S' = 2(A2^n + 2^{n-1}) = 2^n \pmod{2^{n+1}}$. This means that $x_0 \ne x_{2^n} \pmod{2^{n+1}}$, and thus the mapping $x \rightarrow x + r(x) \bmod 2^{n+1}$ defines a single cycle for $n+1$.

The mapping $x \rightarrow x + C$, where C is an odd constant defines a single cycle. So, the condition "$x \rightarrow x + r(x) \bmod 2^{N_0}$ defines a single cycle" always holds if $r(x) \bmod 2^{N_0}$ is an odd constant. This is always true for $r(x) = r'(x) \vee C$, where $C = \underbrace{1\ldots1}_{N_0}{}_2$, although other constants are also possible.

We already know that for $n \ge 3$, x^2 is an even parameter. So, $x \rightarrow x + (x^2 \vee 111_2)$ defines a single cycle modulo 2^n for arbitrary n. It turns out that $\forall x$, $\left[x^2\right]_1 = 0$, so $x^2 \vee 101_2 \pmod{2^3}$ is also a constant. So, $x \rightarrow x + (x^2 \vee C)$

defines a single cycle if and only if $C = \star \cdots \star 1 \star 1_2$, where $\star$ denotes 0 or 1. (It is easy to check by direct calculation modulo 2^3 that it is also a necessary condition.) This shows that our theorem 2 is a generalization of the theorem which was proven in [3].

A further generalization of this result, which considers the cyclic use of several constants C, is

Theorem 3. *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 for any $k = 0, \ldots, m-1$ C_k is some constant. Then the sequence of pairs (x_i, k_i) has a maximal period ($m2^n$) if and only if m is odd, and for all k, $[C_k]_0 = 1$ and $\bigoplus_{k=0}^{m-1} [C_k]_2 = 1$.

The proof of this result is quite complicated, and is omitted due to space limitations.

4 Latin Squares

Let X be a finite set of values. A function $f : X \times X \to X$ is called a *latin square* of *order* $K = |X|$ if both $f(x, \cdot)$ and $f(\cdot, x)$ are invertible. Latin squares have a rich mathematical structure, and they were shown to be useful building blocks in a variety of cryptographic applications (especially in block ciphers and hash functions, see e.g. [6]). In order to check that a function $f(\cdot, \cdot)$ which is a composition of primitive operations is a latin square it is necessary and sufficient to check, using machinery developed in [3], that $f(x, C)$ and $f(C, x)$ are invertible for every value of C.

Let us prove, for example, that $(x \oplus y)(A(x \wedge y) + B)$ is a latin square, where A is any even number and B is any odd number (i.e., $[A]_0 = 0$ and $[B]_0 = 1$): $[(x \oplus C)(A(x \wedge C) + B)]_0 = ([x]_0 \oplus [C]_0)([A]_0 [x]_0 [C]_0 \oplus [B]_0) = [x]_0 \oplus [C]_0$, and $[(x \oplus C)(A(x \wedge C) + B)]_i = [x \oplus C]_i [(A(x \wedge C) + B)]_0 \oplus [x \oplus C]_0 [A(x \wedge C) + B]_i \oplus \alpha_1 = [x \oplus C]_i \oplus [x \oplus C]_0 ([A]_i [x]_0 [C]_0 \oplus [A]_0 [x]_i [C]_i \oplus [B]_i \oplus \alpha_2) \oplus \alpha_1 = [x]_i \oplus \alpha_3$.

A pair of latin squares (f, g) is called *orthogonal*[4] if

$$\forall (x, y) \neq (x', y'),\ (f(x, y), g(x, y)) \neq (f(x', y'), g(x', y')). \tag{3}$$

Such mappings are also known as multipermutations, and were used in several cryptographic constructions. They are known to exist for all orders except two and six. There is an easy construction of orthogonal latin squares of odd order K: $(f, g) = (x + y, 2x + y) \bmod K$. We will concentrate on $K = 2^n$. Let $\mathbb{F}$ be a finite

[4] As Euler used Greek and Latin letters to denote values of f and g respectively, they got also known as *Graeco-Latin* squares.

field with 2^n elements. If $\alpha, \beta \in \mathbb{F}$, $\alpha, \beta \neq 0$ and $\alpha \neq \beta$ then $(f, g) = (\alpha x + y, \beta x + y)$ are orthogonal. Actually, this is an abuse of the notation, since $f(x, y) = \phi^{-1}(\alpha\phi(x) + \phi(y))$, where $\phi : \{0 \ldots 2^n - 1\} \to \mathbb{F}$ is a one-to-one mapping between integers and the field elements. Unfortunately, most processors do not have built-in operations which deal with finite fields, and thus the mapping is quite slow when implemented in software. In addition, the resulting transformation is linear, and thus it is a very weak cryptographic building block.

In [5] it was shown that there are no orthogonal latin squares formed by a pair of polynomials modulo 2^n. Let us prove that this is true not only for polynomials modulo 2^n but also for the wider class of all the possible T-functions. Consider the latin square of order 2^n formed by a T-function $f(x, y)$. Given arbitrary $x_0, y_0 < 2^{n-1}$ let $x'_0 = x_0 + 2^{n-1}$ and $y'_0 = y_0 + 2^{n-1}$. Since f is a T-function and $(x_0, y_0) = (x'_0, y'_0) \pmod{2^{n-1}}$ it follows that $f(x_0, y_0) = f(x_0, y'_0) = f(x'_0, y'_0) \pmod{2^{n-1}}$. The function $f(x_0, y)$ is invertible modulo 2^n, so $f(x_0, y'_0) = f(x_0, y_0) + 2^{n-1} \pmod{2^n}$. For the same reason $f(x'_0, y'_0) = f(x_0, y'_0) + 2^{n-1} \pmod{2^n}$, so $f(x_0, y_0) = f(x'_0, y'_0) \pmod{2^n}$. Suppose that a pair (f, g), where f and g are T-functions is a pair of orthogonal latin squares. Similar to the above $g(x_0, y_0) = g(x'_0, y'_0) \pmod{2^n}$, so $(f(x_0, y_0), g(x_0, y_0)) = (f(x'_0, y'_0), g(x'_0, y'_0)) \pmod{2^n}$ which contradicts (3).

In spite of this negative result, we can construct a pair of orthogonal latin squares using T-functions, if we consider the top and bottom half of each variable as a separate entity. This is a natural operation in many processors, which allows us to consider each machine word either as one word of length $2n$ or as two consecutive words of length n. Every $0 \leq x < 2^{2n}$ could be represented as $x = x_u 2^n + x_v$, where $0 \leq x_u, x_v < 2^n$. There exist T-functions $f_u(x_u, x_v, y_v, y_v)$, $f_v(x_u, x_v, y_v, y_v)$, $g_u(x_u, x_v, y_v, y_v)$ and $g_v(x_u, x_v, y_v, y_v)$, such that $(f, g) = (f_u 2^n + f_v, g_u 2^n + g_v)$ is a pair of orthogonal latin squares of order 2^{2n}. Let us construct them.

Table 1. A pair of orthogonal latin squares of order four

0,0	1,1	2,2	3,3
1,2	0,3	3,0	2,1
2,3	3,2	0,1	1,0
3,1	2,0	1,3	0,2

Table 1 shows a pair of orthogonal latin squares of order four ($n = 1$). Columns correspond to x and rows to y, for example, $f(0, 1) = 1$ and $g(0, 1) = 2$ or more explicitly for arguments $(x_u = 0, x_v = 0, y_u = 0, y_v = 1)$ $f_u = 0$, $f_v = 1$, $g_u = 1$ and $g_v = 0$. Each of x_u, x_v, y_v and y_v consists of only a single bit, so f_u, f_v, g_u and g_v are clearly T-functions. Suppose we have a pair (f, g) of orthogonal latin squares of order 2^{2n}, where f_u, f_v, g_u and g_v are T-functions. To construct a pair (f', g') of orthogonal latin squares of order $2^{2(n+1)}$ we do the following. Let $[f'_u]_i \equiv [f_u]_i$, $[f'_v]_i \equiv [f_v]_i$, $[g'_u]_i \equiv [g_u]_i$ and $[g'_v]_i \equiv [g_v]_i$ for

$i = 0, \dots, n-1$. This way bit number i of f' depends on the same bits of the arguments as f, which is a T-function and so, f' is also a T-function. The same holds for f_v, g_u and g_v. For every fixed $(x_u, x_v, y_u, y_v) \pmod{2^n}$ let us define (f', g') in such a way that $[f', g']_n$ forms a pair of orthogonal latin squares. Let us prove that f' and g' are latin squares. Suppose that there exists a row y_0 such that $\exists x_0, x_1 : f'(x_0, y_0) = f'(x_1, y_0)$. There are two possibilities: either $x_0 \neq x_1 \pmod{2^n}$, so, $f'(x_0, y_0) = f(x_0, y_0) \neq f(x_1, y_0) = f'(x_1, y_0) \pmod{2^n}$ or $x_0 = x_1 \pmod{2^n}$, but in this case $[f'(x_0, y_0)]_n \neq [f'(x_1, y_0)]_n$, because for any fixed n the least significant bits of arguments $[f']_n$ forms a latin square. The same proof applies to the columns of f', to g' and to the orthogonality of f' and g'.

Let us show some examples of orthogonal latin squares. First we need to encode table 1:[5]

$$\begin{aligned}
f_u^\star &= \texttt{0011001111001100} = x_v \oplus y_v, \\
f_v^\star &= \texttt{0101101001011010} = x_u \oplus y_u, \\
g_u^\star &= \texttt{0011110011000011} = x_v \oplus y_u \oplus y_v, \\
g_v^\star &= \texttt{0101010110101010} = x_u \oplus y_v.
\end{aligned}$$

The next question is how to define (f, g) in such a way that $[(f, g)]_n$ forms a pair of orthogonal latin squares for every fixed $(x_u, x_v, y_u, y_v) \bmod 2^n$. The simplest way is to use for $[f, g]_n$ the same functions $(f^\star, g^\star)$, that is $f = (x_v \oplus y_v, x_u \oplus y_u)$, $g = (x_v \oplus y_u \oplus y_v, x_u \oplus y_v)$. Note that if (f, g) is a pair of orthogonal latin squares and ϕ is an invertible mapping then $(\phi(f), g)$ is also a pair of orthogonal latin squares. The mapping $\phi(x) = x \oplus C$ is invertible, thus the bit trace $[f]_n = (f_u^\star \oplus \alpha, f_v^\star \oplus \beta)$, where α and β are parameters, is also admissible. So, for example, $f = (x_v \oplus y_v - (x_u^2 \vee 1), 7x_u + y_u)$ and $g = (x_v - (y_u \oplus y_v) + (x_u^2 \vee 1), x_u \oplus y_v)$ are nonlinear and non-algebraic orthogonal latin squares.

We are not limited to latin squares of size $2^{2n} \times 2^{2n}$, since using similar construction we can build a pair of orthogonal latin squares of size $2^{mn} \times 2^{mn}$ for any m. For example, if $m = 4$ and we represent the inputs as $x = 2^{3n}x_s + 2^{2n}x_t + 2^n x_u + x_v$ and $y = 2^{3n}y_s + 2^{2n}y_t + 2^n y_u + y_v$, then

$$\begin{aligned}
f &= (x_s \oplus y_s + 2x_v y_u, \quad x_t + y_t, \quad x_u \oplus y_u + 2x_s x_v, \quad x_v + y_v + 2x_s y_t), \\
g &= (x_s + y_t \oplus 2x_u y_v, \quad x_t \oplus y_u, \quad x_u + y_v \oplus y_s, \quad x_v \oplus y_s \oplus (x_s^2 \vee 1))
\end{aligned}$$

is an example of a nonlinear pair of orthogonal latin squares of size $2^{4n} \times 2^{4n}$.

5 The Minimality of Mappings

The mapping $x \to x + (x^2 \vee C)$, where $C = \star \cdots \star 1 \star 1_2$ is a nonlinear invertible function with a single cycle which requires only three primitive operations. The

[5] It is not a coincident that the functions are linear. The pair in table 1 was constructed by $(f, g) = (\alpha x + y, \beta x + y)$, where $\mathbb{F}$ is $GF_2[t]/(t^2 + t + 1)$, $\alpha = 1$, $\beta = t$. Addition in the field is just bitwise-xor — this completely explains $f^\star$, and multiplication by t is one bit left shift and xor with 11_2 in case of “carry”.

natural question is whether there are any other mappings with these properties which are compositions of one, two, or three primitive operations. In the full version of the paper we describe the search technique we used to show that there are no nonlinear mappings with one or two primitive operations which have this property. In fact, the term "*number of operations*" has two possible interpretations: we can count the number of internal nodes in the parse tree of an expression, or we can count the number of instructions in a straight-line program which evaluates this expression. In the first case the only other nonlinear mapping with three operations which has a single cycle is $x \to x - (x^2 \vee C)$, where $C = \star \cdots \star 1 \star 1_2$ (note that if $x' = x + (x^2 \vee C)$ then $-x' = -x - ((-x)^2 \vee C)$ and thus each one of the sequences is the negation (mod 2^n) of the other sequence). This is true for any word size n and for any choice of n-bit constants in the mapping, and thus it is impossible to prove such a result by exhaustively searching the infinite space of all the possible mappings. In the second model there are exactly two additional families of single cycle mappings: $x \to x \pm (x \vee C)^2$, where C is any odd constant, which can be calculated with three machine operations: $r_1 = x \vee C$, $r_2 = r_1 \times r_1$, $r_3 = x \pm r_2$. However, since the squaring is not technically considered as a primitive operation, the parse tree of the expression is based on $x \pm (x \vee C)(x \vee C)$ and thus according to the previous model it has four operations. The latter model depends on the instruction set of the target CPU. For example, the `SPARC` instruction set has a command to evaluate $r_1 = x \vee C$, but in the `x86` instruction set the result in most operations is stored in place of the first operand, and thus to evaluate $r_1 = x \vee C$ we need to execute *two* operations: $r_1 = x$ and $r_1 = r_1 \vee C$. Because of this architecture dependency we use the former model in the rest of the section.

By using similar techniques, we can find the smallest number of primitive operations required to define a nonlinear latin square $f(x, y)$, where *nonlinear* means that there is a multiplication operation in which both sides depend on x and a multiplication operation in which both sides depend on y. One example is $C_0 + ((x + y) * (C_1 \vee (x + y)))$, where C_0 is odd and $C_1 = 1 \cdots 11_2$, which is actually equivalent to the linear mapping $C_0 + C_1(x + y)$. To avoid such cases we add another constraint: for any subexpression g (except C) there should be (x, y) and (x', y') such that $g(x, y) \neq g(x', y')$. Experiment shows that there are no expressions with fewer than five primitive operations satisfying these constrains and there are 188 such expressions with five primitive operations. All of them have the form $\beta * \gamma$ with four possible β and 47 possible γ.

6 Combinations of Permutations

Let $f(x)$ and $g(x)$ be arbitrary permutations over n-bit words. Their sum can never be a permutation, and their XOR is extremely unlikely to be a permutation. The problem of constructing "random looking" $f(x)$ and $g(x)$ such that $f(x)$, $g(x)$, and $f(x) \oplus g(x)$ are all permutations seems to be difficult, but we can easily construct such T-functions by analyzing their bit-slice mappings. In fact, we can choose $g(x)$ as the identity, and thus generate nonlinear and non-algebraic mappings for which both $f(x)$ and $f(x) \oplus x$ are permutations. We can easily generalize the technique and generate k permutations with the property that the XOR of any pair of them is also a permutation, or generate a single permutation $f(x)$ with the property that various polynomials in f (such as $f(f(f(x))) \oplus f(x) \oplus x$) are also permutations. Due to space limitations, we can only describe the basic idea of the construction, which is to split x into multiple variables $x_0, \dots, x_{m-1}$, and to define $f_0, \dots, f_{m-1}$ so that its bit-slice has 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 matrix, such that both C and $C \oplus I$ (or both C and $C^3 \oplus C \oplus I$) are invertible.

7 Fast Iteration and Inversion of T-Functions

Consider the sequence $\{x_i\}$, where $x_{i+1} = f(x_i) \bmod 2^n$. The problem considered in this section is how to efficiently calculate x_k given x_0 and k without going through all the intermediate values. This problem has at least two cryptographic applications. If $f(x)$ is used as the state transition function in a stream cipher, the solution allows us to jump into an arbitrary position of the stream without calculating all the intermediate states. If $f(x)$ is used as the round function of a block cipher, then in order to decrypt the ciphertext we have to run $f(x)$ backwards (i.e., calculate x_{-1} in the above sequence) more efficiently than in the bit-by-bit manner considered so far. The length of each cycle of a T-function is 2^k for some $k \leq n$, and thus $x_{2^n} = x_0$ which implies that $f^{-1}(x) = x_{2^n-1}$.

Suppose that $f(x)$ is a polynomial $\hat{p}(x)$ of degree d, then the iterated expression $x_m = \hat{p}^{(m)}(x_0)$ is also some polynomial, but unfortunately its degree is d^m. Let k and l be constants such that $kl \geq n$. Any x could be represented in the following form: $x = 2^k y + r$, where $r = x \bmod 2^k$. For any fixed r there exists a polynomial p, such that $\hat{p}^{(m)}(x) = \hat{p}^{(m)}(2^k y + r) = p(2^k y)$. Although the degree of p could be as high as the degree of $p^{(m)}$, modulo 2^n all its coefficients except the first l are equal to zero: $a_i (2^k y)^i = a_i y^i 2^{ki} = 0 \pmod{2^n}$ for $i \geq l$.[6] There are 2^k possible values of r, so if we prepare a lookup table with the coefficients of p for each possible value of r then we could calculate x_k using $l-1$ multiplications and $l-1$ additions modulo 2^n.[7] Consider, for example, the RC6 function $p(x) = x + 2x^2$, and let $n = 64$, $k = 16$ and $l = 4$. The straightforward calculation of its inverse takes on average[8] $\approx \frac{n}{2} = 32$ calculations of

[6] Note that in order to calculate p modulo 2^n we do not need all the bits of the coefficients a_i, but only $a_i \bmod 2^{k(l-i)}$. So, in order to store the coefficients we need only $k(l + (l-1) + \cdots + 1) = \frac{kl(l+1)}{2}$ bits of memory.

[7] An interesting consequence of this is that most T-functions over n-bit words are not representable as a polynomial modulo 2^n, since in order to encode a general T-function we need $2^n + 2^{n-1} + \cdots + 2^1 = 2^{n+1} - 1$ bits of memory which is greater than the size of the table. This is in contrast to the fact that over a finite filed such as $\mathbb{Z}_p$ any function can be represented as a (very high degree) polynomial.

[8] Note that if we calculate $[f(x)]_i$ and find that $[x]_i = 0$ than we do not need to recalculate $f(x)$ in order to find $[f(x)]_{i+1}$.

$p(x)$, that is about 32 multiplications and 32 additions. Our method requires only three multiplications and three additions which is only three times slower than the forward calculation. The drawback is that our method needs a fixed precomputed table of $2^{k-1}kl(l+1) \approx 2^{23}$ bits ≈ 1 megabyte of memory, whereas the slow method does not need precomputations and memory. However, a one megabyte table is not a problem in many PC-based cryptographic applications.

To construct the lookup table we use the following property: $p^{(a+b)}(x) = p^{(a)}(p^{(b)}(x))$, so in order to calculate $p^{(2^n)}(x)$ we need n "symbolic" evaluations of polynomial of polynomial. Care must be taken to use $p[r]$ for the right r in each step, that is in the general case $r = x \bmod 2^k$ is not equal to $r' = p^{(b)}(x) \bmod 2^k$, so we need to calculate the whole table for $p^{(2^k)}(x)$ before constructing the table for $p^{(2^{k+1})}(x)$.

Unlike the RC6 function $x + 2x^2$, the function $f(x) = x + (x^2 \vee 5)$ is not a polynomial, but if $k = 3$ it is a polynomial for every possible value of r. For example, if $r = x \bmod 2^3 = 1$ then $x^2 \bmod 2^3 = 1$, whereas $x^2 \vee 5 \bmod 2^3 = 5$, so $f(x)$ can be represented by the polynomial $x + x^2 + 4$ for each such input x.

8 Cryptanalysis of $x \to x + (x^2 \vee C) \pmod{2^n}$

Our goal in this paper is to develop new efficient building blocks for larger cryptographic schemes, and in many cases the security of the building block depends on how it is used. In particular, we would like to use the $x \to x + (x^2 \vee 5) \pmod{2^n}$ mapping as a substitute for LFSRs or linear congruential `rand()` functions rather than as a stand-alone stream cipher. However, we did try to analyze the security of the exceptionally simple stream cipher which just iterates this function and sends the m most significant bits[9] of each n-bit state to the output, and discovered that it is surprisingly strong against all the attacks we could think of — the running time of all our attacks is exponential in n. The purpose of this section is to summarize our findings, but due to space limitations we outline only some of the attacks.

Any PRNG with an n-bit state could be attacked as follows. Precompute the outputs of the generator for random 2^t states, where the length of each output should be sufficient to identify uniquely the state. Sort them in order to be able to retrieve the state given the output sequence. Given 2^d actual outputs try to locate any one of them in the table. If the parameters are such that $t + d \approx n$ we have a significant probability of success. One possible choice of parameters is $t = d = \frac{n}{2}$.[10] In this case we use $O(2^{\frac{n}{2}})$ of memory and time, and thus the effective security of any PRNG is no more than $\frac{n}{2}$ bits.

[9] Note that the i-th least significant bit in any iterated single cycle T-function repeats every 2^i steps and depends only on the first i state bits, and thus the most significant bits are much stronger than the least significant bits in this application. In addition, we assume that $m \ll n$.

[10] Usually the processing time is cheaper than memory, and access to secondary memory takes a significant amount of time, so in practice it is better to have $t > d$. We do

Let us show how to attack our generator with the same complexity $O(2^{\frac{n}{2}})$ but without the preprocessing and additional memory, besides those which are used for the output sequence itself. Consider $x_i = \underbrace{\star\dots\star}_{\frac{n}{2}}\underbrace{0\dots0}_{\frac{n}{2}}$, that is $x_i = 2^{\frac{n}{2}}y$ for some y. In this case $x_i^2 = 0 \pmod{2^n}$, so $x_{i+1} = x_i + 5 \bmod 2^n$, and thus with high probability the most significant bits of x_{i+1} are the same as those of x_i. On the other hand if x_i does not end with $\frac{n}{2}$ zeros then the probability that the m most significant bits of x_i and x_{i+1} are the same is approximately 2^{-m}. Also note that if $x_i = 2^{\frac{n}{2}}y$ then $x_{i+2^{\frac{n}{2}}} = 2^{\frac{n}{2}}y'$. So, checking a few pairs of the outputs with a $2^{\frac{n}{2}}$ shift between the pairs we could identify the point at which the least significant halves of $x_{i+l2^{\frac{n}{2}}}$ are zeros. It is also guaranteed that one of any $2^{\frac{n}{2}}$ consecutive x_i's has this property.

We now describe an improved attack in which the data, time and memory complexities are of order $2^{\frac{n}{3}}$. Let $x = 2^{\frac{2n}{3}}x_u + 2^{\frac{n}{3}}x_v + x_w$ and suppose that $x_w = 0$. In this case $(2^{\frac{2n}{3}}x_u + 2^{\frac{n}{3}}x_v + x_w) \to (2^{\frac{2n}{3}}x_u + 2^{\frac{n}{3}}x_v + x_w) + ((2^{\frac{2n}{3}}x_u + 2^{\frac{n}{3}}x_v + x_w)^2 \vee 5) = 2^{\frac{2n}{3}}(x_u + x_v^2) + 2^{\frac{n}{3}}x_v + 5 \pmod{2^n}$, and so the difference between the most significant bits of consecutive outputs is approximately x_v^2. If x_v is known (and $x_w = 0$) we can easily calculate the value of x_v after $2^{\frac{n}{3}}k$ steps for several k. If we make a lookup table of these values for all possible x_v we can find x_v after inspecting a few pairs of the outputs with a $2^{\frac{n}{3}}$ shift between them.

To reduce the amount of data needed to attack the generator, we can use a different approach. In section 7 it was shown how to represent x_i as a polynomial of degree $l-1$ in $x_0 - r$ if $r = x_0 \bmod 2^k$ was guessed correctly. Given $\{x_i = a_{i,l-1}y_0^{l-1} + \dots + a_{i,0}\}_{i=0}^{p-1}$ it is possible, as we will see below, to find $\{\beta_i \in \{-1,0,+1\}\}_{i=0}^{p-1}$ such that

$$\forall j, \sum_{i=0}^{p-1} \beta_i a_{i,j} = 0 \pmod{2^{k(l-j)}}. \quad (4)$$

If the sum is zero as a polynomial then $S = \sum \beta_i x_i = 0$ as long as the k least significant bits of x_0 are guessed correctly. On the other hand if the guess is incorrect then S is likely to be random. Provided that we could distinguish these cases we could find the k least significant bits of x_0 after about $\frac{2^k}{2}$ tests. Using the same technique while incrementing k we could find the rest of the state x_0. The only two remaining questions are:

- How to notice that $\sum \beta_i x_i = 0$ given only the m most significant bits of x_i?
- How to find those β_i?

not store all the preprocessed sequences but only those that have some distinguishable feature (like starting with l zeros), thus reducing memory and postprocessing time by factor of 2^l.

Let us denote by u_i the m most significant bits of x_i, that is $x_i = 2^{n-m}u_i + v_i$, where $v_i < 2^{n-m}$. This way

$$S = \sum_{i=0}^{p-1} \beta_i x_i = 2^{n-m} \sum_{i=0}^{p-1} \beta_i u_i + \sum_{i=0}^{p-1} \beta_i v_i = 2^{n-m} A \ + B \pmod{2^n}.$$

If the guess is correct, then $S = 0 \pmod{2^n}$, and thus $B = S = 0 \pmod{2^{n-m}}$. Let us denote $B' = \frac{B}{2^{n-m}}$. In the general case $B' = q2^m - A$, where $q \in \mathbb{Z}$ and $A \in [1, 2^m]$. If $p < 2^m$ then $B' < 2^m$ so an attacker could calculate $B' = -A \bmod 2^m$ and it should behave approximately as the sum of p uniformly distributed in $[0, 1]$ random variables which is approximately normal with mean $\frac{p}{2}$. On the other hand if the guess is wrong A should be uniformly distributed in $[1, 2^m]$. An attacker can easily tell apart the cases by inspecting the expectation (average) of A for several different tuples $\{\beta_i\}$. If $p > 2^m$ the test of the expectation is not sufficient because B' is not guaranteed to be less than 2^m and thus q is not always 1. But if $p < 2^{2m}$ then $\mathrm{Var}(B') \approx \frac{p}{12}$ so the standard deviation is smaller than 2^{m-1} and thus with high probability $q = 1 + \lfloor \frac{p}{2^{m+1}} \rfloor$, so $A = q2^m - B'$ is normal and $\mathrm{Var}(A) \approx \frac{p}{12}$. On the other hand if the guess is incorrect $\mathrm{Var} A \approx \frac{2^{2m}}{12}$, so an attacker again could tell if his guess is correct.

The second question is more difficult. The probability that a random polynomial conform to (4) is 2^{-t}, where $t = \frac{kl(l+1)}{2}$. There are 3^p ways to assign $\{-1, 0, +1\}$ to $\{\beta_i\}_{i=0}^{p-1}$, so as long as $p > \tau t$, where $\tau = \frac{\log 3}{\log 2} \approx 0.63$ such tuple exists with high probability. For example if $k = 16$ and $l = 4$ then t= 160 and $p > 101$. Unfortunately, exhaustive search would take $O(t)$ time which is not practical even for a precomputation step. There are several *practical* approaches to the problem.

We could relax the problem of finding $\beta_i \in \{-1, 0, +1\}$ conforming to (4) to finding β_i which are small integers. The new problem could be solved by LLL reduction of the following (row) lattice:

$$\begin{pmatrix}
1\ 0\ 0 \cdots 0 & a_{0,0}\psi & a_{0,0}\psi & \cdots & a_{0,l-1}\psi \\
0\ 1\ 0 \cdots 0 & a_{1,0}\psi & a_{1,0}\psi & \cdots & a_{1,l-1}\psi \\
0\ 0\ 1 \cdots 0 & a_{2,0}\psi & a_{2,0}\psi & \cdots & a_{2,l-1}\psi \\
\vdots \quad \ddots \ \vdots & \vdots & \vdots & \vdots & \vdots \\
0\ 0\ 0 \cdots 1 & a_{p-1,0}\psi & a_{p-1,0}\psi & \cdots & a_{p-1,l-1}\psi \\
0\ 0\ 0 \cdots 0 & 2^{kl}\psi & 0 & \cdots & 0 \\
0\ 0\ 0 \cdots 0 & 0 & 2^{k(l-1)}\psi & \cdots & 0 \\
\vdots\ \vdots\ \vdots\ \vdots\ \vdots & \vdots & \vdots & \ddots & \vdots \\
0\ 0\ 0 \cdots 0 & 0 & 0 & \cdots & 2^k\psi
\end{pmatrix},$$

where ψ is a sufficiently large constant. If $p > \tau t$ we know that the squared length of the smallest vector in the lattice is smaller than p. On the other hand LLL algorithm with $\frac{1}{4} < \delta \leq 1$ is guaranteed to return a non-zero basis vector which is no more than $\left(\frac{1}{\delta - \frac{1}{4}}\right)^{p+l-1}$ times longer then the shortest vector in the lattice. This theoretical guarantee is not acceptable but in practice LLL behaves much better. For example, if[11] $p = 160$, the LLL algorithm returns after a few seconds on a standard PC a vector such that $\sum_i |\beta_i| = 108$ and $\max_i |\beta_i| = 9$.

In order to solve efficiently the exact problem we could use a variation [2] of Wagner's algorithm [7]. Given approximately 2^ω (for our purposes $\omega^2 = t$) members $\{x_i\}$ of a group $G' = G^\omega$ ($|G| = 2^\omega$) and the target value $z \in G'$ it finds $\{\beta_i\}$ from $\{-1, 0, +1\}$ such that $\sum_i \beta_i x_i = z$ using $O(2^\omega)$ time and memory. By using this approach we are not limited to zero on the right hand side of (4), so we could find $\{\beta_i\}$ such that $\sum_i \beta_i x_i(y) = 2^m y$ and thus reveal the m most significant bits of y or, equivalently, of x_0 instead of repeating with larger k's. For example, if $n = 64$, $k = 16$, $l = \frac{n}{k} = 4$, $t = \frac{kl(l+1)}{2} = 160$, $\omega = 13$, then preprocessing and checking each of 2^k consecutive positions needs roughly $O(2^{16})$ data, memory and time. Note that this attack works only if the constant C is smaller than 2^k, so if, for example, size of C is $\frac{n}{3}$ the attack is still exponential and requires $O(2^{\frac{n}{3}})$ time.

References

[1] V. Anashin, "Uniformly Distributed Sequences of p-adic integers, II", To appear in Diskretnaya Mathematika (Russian), English translation in Diskrete Mathematics (Plenum Publ.). Available from `http://www.arxiv.org/ps/math.NT/0209407`. 250

[2] A. Joux, "Cryptanalysis of the EMD Mode of Operation", EUROCRYPT 2003. 261

[3] A. Klimov and A. Shamir, "*A New Class of Invertible Mappings*", CHES 2002. 249, 250, 253

[4] R. Rivest, M. Robshaw, R. Sidney, and Y. L. Yin, "*The RC6 block cipher*". Available from `http://www.rsa.com/rsalabs/rc6/`.

[5] R. Rivest, "Permutation Polynomials Modulo 2^ω", 1999. 254

[6] S. Vaudenay, "On the need for multipermutations: Cryptanalysis of MD4 and SAFER", FSE 1995. 253

[7] D. Wagner, "A Generalized Birthday Problem", CRYPTO 2002. 261

[11] In fact $\{\beta_i = 0\}_{i=54}^{p-1}$, so the same vector is returned if $p = 54$.