ConSum v0: An Experimental Cipher

David A. Madore

April 29, 2007

CVS:
`$Id: consum.tex,v 1.22 2007-04-28 23:16:41 david Exp $`

Abstract

We present an experimental block cipher, ConSum, based on a hitherto unstudied design element: the *Conway transformation*. ConSum features an extremely simple design and the ability to operate with arbitrary key lengths, block sizes and round numbers. We study it empirically and statistically so as to illustrate how it might be secure.

Contents

1 Introduction

1.1 What is ConSum?

ConSum is an experimental symmetric block cipher whose two distinguishing characteristics are simplicity of design and flexibility. *Simplicity of design* in that it is devoid of elements as mysterious S-boxes, arbitrarily chosen primitive polynomials, random constants or any such things: the entire ConSum design is easily described and remembered. By *flexibility* we mean that ConSum can operate on any block size and any key length that is at least equal to the block size, provided both are powers of 2.

As its name indicates, the cipher we are presenting is built from two basic elements: the *Conway transformation* and addition. We chose the name "Conway transformation" because it is an instance of Conway multiplication of binary numbers, introduced in [1] and [2] (where it is called "Nim multiplication"); as far as we are concerned, the Conway transformation is a simply constructed bijective linear transformation $x \mapsto K(x)$ on $(\mathbb{F}_2)^{2^n}$ (binary sequences of length 2^n) with satisfactory "mixing" properties. The other design element of ConSum is simply addition of binary integers, $(x, y) \mapsto x \boxplus y$. Note that both of these elements are "linear" in a certain sense: the Conway transformation is linear relative to the XOR operation $(x, y) \mapsto x \oplus y$ on bit strings (which makes $(\mathbb{F}_2)^{2^n}$ into an $\mathbb{F}_2$-vector space), that is, $K(x \oplus y) = K(x) \oplus K(y)$, and it is only from the departure of $x \boxplus y$ from $x \oplus y$, viz., the carry bits, that the security of ConSum should emerge. We will give, below, an argument based on entropy computations as well as certain statistical observations, to argue why the complexity of the carry bits could suffice to account for cryptographic security.

1.2 Why another block cipher?

Simplicity in design is a double-edged sword: while it makes our cipher appealing and easy to analyze in ways that indicate security, the very same characteristic also makes is conceivable that attacks might be found which exploit this simplicity.

For this reason, ConSum is intended more as an experiment in cryptography than as a directly usable cipher: this is why emphasis is put on the word "experimental" in the title. The provoking question we wish to ask the cryptographic community by submitting this cipher to peer review is whether an extremely simple design using only "linear" building elements can nevertheless defy cryptanalysis. We feel that there is much to learn from this experiment: if ConSum holds, it shows certain new elegant ways of constructing block ciphers; if it does not, we will have gained some insight in the way the addition ($\boxplus$) and XOR ($\oplus$) operations interact on binary strings, a discovery which could be relevant in mathematics as well as cryptography. Either way, there is something to be gained by studying the construction.

1.3 Naming conventions

The version of the cipher described in this paper is "version 0". If subsequent discoveries warrant it, we might make some changes in the design structure, in which case the version number will be incremented to reflect these changes.

Since the flexible nature of ConSum does not impose a specific block size, key length, or number of rounds, we dictate that the notation “ConSum(v0, b128, k256, r12)” means “ConSum version 0 with block size 128 bits, key length 256 bits, and 12 rounds”. Omitting the key length means it is equal to block size. Omitting other parameters means they have their default values: 128 bits for block size, and 10 rounds.

2 Conventions and notations

We consider bit strings whose lengths are usually powers of 2 (which we will feel free to write in hexadecimal whenver convenient). We make the convention that if x_0 and x_1 are two bit strings (of lengths N_0 and N_1) then $x_1|x_0$ is their concatenation (of length $N = N_0 + N_1$) in which x_0 is the *low* (= least significant) part and x_1 is the *high* (= most significant) part (note that we are *not* implying anything about how these bit strings are arranged in memory: in particular, we are not saying anything about endianness, here). This will be particularly useful when x_0 and x_1 are of the same length 2^n, in which case $x_1|x_0$ is of length 2^{n+1}.

We write $x \oplus y$ for the eXclusive OR of two bit strings x and y (of equal length). Formally, this can be defined inductively as follows:

$$\texttt{0} \oplus \texttt{0} = \texttt{0}, \quad \texttt{0} \oplus \texttt{1} = \texttt{1}, \quad \texttt{1} \oplus \texttt{0} = \texttt{1}, \quad \texttt{1} \oplus \texttt{1} = \texttt{0}$$
$$(x_1|x_0) \oplus (y_1|y_0) = (x_1 \oplus y_1|x_0 \oplus y_0)$$

Similarly, we write $x \boxplus y$ for the sum of two bit strings identified as integers; for example, `de4a` $\boxplus$ `cccc` $=$ `ab16`: carries from adding bit 0 are added to bit 1, and so on. The carry $c(x, y)$ from the last (highest) bit is thrown away (i.e. we operate modulo 2^N). Here is a formal (and annoyingly long) definition of addition by induction:

$$\begin{array}{rl}
\mathtt{0} \boxplus \mathtt{0} = \mathtt{0}, \quad \mathtt{0} \boxplus \mathtt{1} = \mathtt{1}, \quad \mathtt{1} \boxplus \mathtt{0} = \mathtt{1}, \quad \mathtt{1} \boxplus \mathtt{1} = \mathtt{0} & \\
c(\mathtt{0},\mathtt{0}) = \mathtt{0}, \quad c(\mathtt{0},\mathtt{1}) = \mathtt{0}, \quad c(\mathtt{1},\mathtt{0}) = \mathtt{0}, \quad c(\mathtt{1},\mathtt{1}) = \mathtt{1} & \\
\left.\begin{array}{r} (x_1|x_0) \boxplus (y_1|y_0) = (x_1 \boxplus y_1)|(x_0 \boxplus y_0) \\ c((x_1|x_0),(y_1|y_0)) = c(x_1,y_1) \end{array}\right\} & \text{if } c(x_0,y_0) = \mathtt{0} \\
\left.\begin{array}{r} (x_1|x_0) \boxplus (y_1|y_0) = (x_1 \hat{\boxplus} y_1)|(x_0 \boxplus y_0) \\ c((x_1|x_0),(y_1|y_0)) = \hat{c}(x_1,y_1) \end{array}\right\} & \text{if } c(x_0,y_0) = \mathtt{1} \\
\mathtt{0} \hat{\boxplus} \mathtt{0} = \mathtt{1}, \quad \mathtt{0} \hat{\boxplus} \mathtt{1} = \mathtt{0}, \quad \mathtt{1} \hat{\boxplus} \mathtt{0} = \mathtt{0}, \quad \mathtt{1} \hat{\boxplus} \mathtt{1} = \mathtt{1} & \\
\hat{c}(\mathtt{0},\mathtt{0}) = \mathtt{0}, \quad \hat{c}(\mathtt{0},\mathtt{1}) = \mathtt{1}, \quad \hat{c}(\mathtt{1},\mathtt{0}) = \mathtt{1}, \quad \hat{c}(\mathtt{1},\mathtt{1}) = \mathtt{1} & \\
\left.\begin{array}{r} (x_1|x_0) \hat{\boxplus} (y_1|y_0) = (x_1 \boxplus y_1)|(x_0 \hat{\boxplus} y_0) \\ \hat{c}((x_1|x_0),(y_1|y_0)) = c(x_1,y_1) \end{array}\right\} & \text{if } \hat{c}(x_0,y_0) = \mathtt{0} \\
\left.\begin{array}{r} (x_1|x_0) \hat{\boxplus} (y_1|y_0) = (x_1 \hat{\boxplus} y_1)|(x_0 \hat{\boxplus} y_0) \\ \hat{c}((x_1|x_0),(y_1|y_0)) = \hat{c}(x_1,y_1) \end{array}\right\} & \text{if } \hat{c}(x_0,y_0) = \mathtt{1}
\end{array} \tag{1}$$

(To make sense of this, keep in mind that $x \hat{\boxplus} y$ means $x \boxplus y \boxplus \mathtt{1}$ where the $\boxplus\mathtt{1}$ comes from a carry.)

We similarly write $x \boxminus y$ for the difference of x and y (performed with borrows, with the borrow from the highest bit forgotten).

3 The Conway transformation

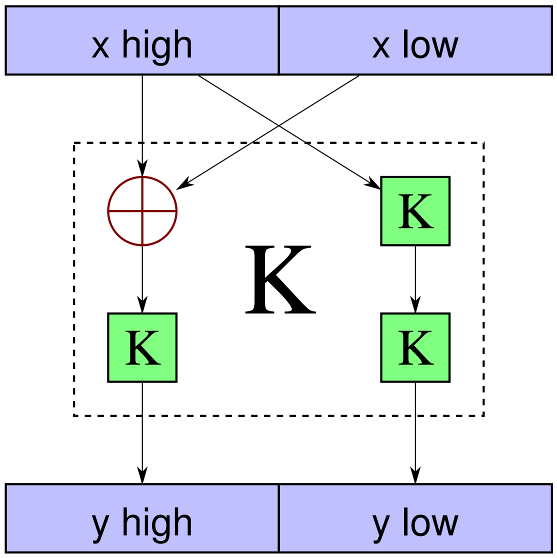
3.1 Definition and algebraic properties

We define the *Conway transformation* Kx of a bit string x of length $N = 2^n$ (necessarily a power of two) by induction on n as follows:

$$\begin{gathered} K\mathtt{0} = \mathtt{0}, \quad K\mathtt{1} = \mathtt{1} \\ K(x_1|x_0) = K(x_0 \oplus x_1) \,|\, K^2 x_1 \end{gathered} \tag{2}$$

We emphasize that in the first line of this definition, $\mathtt{0}$ and $\mathtt{1}$ are single bits, and in the second line x_0 and x_1 are of equal length, this length being a power of 2. Naturally, K^2x stands for the Conway transformation iterated twice on x.

We summarize the definition of K as follows:



The following table gives the Conway transformation on all 4-bit strings (written as hexadecimal digits):

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Kx	0	8	c	4	b	3	7	f	d	5	1	9	6	e	a	2

As an example of the Conway transformation, here is the iteration of it on 8-, 16-, 32- and 64-bit strings, starting with 1:

01	0001	00000001	0000000000000001
80	8000	80000000	8000000000000000
de	de4a	de4ae3a9	de4ae3a94a88a921
4a	e3a9	4a88a921	e3a92850a9218171
a9	a921	a9218171	a92181714b010a1a
48	0e69	22fd8f18	06e6e47169fc8502
69	8f18	69fc8502	8f189ae87791f90e
2f	a38c	4de7ee6b	9c27eb8bd1c005e0
e6	e6e4	e6e41fea	e6e41feab43c1101

Evidently, the Conway transformation is bijective; and its inverse is given by:

$$\begin{gathered} K^{-1}0 = 0, \quad K^{-1}1 = 1 \\ K^{-1}(z_1|z_0) = (K^{-2}z_0)|(K^{-2}z_0 \oplus K^{-1}z_1) \end{gathered} \tag{3}$$

Here is the iteration of K^{-1} on bit strings as previously starting with 1:

01	0001	00000001	0000000000000001
ee	8585	d963d963	ada91d55ada91d55
85	d963	ada91d55	e047faadccf408df
ba	b0fc	2cb3f272	124577333f9b3e84
d9	ada9	e047faad	8d5d71c89d1fcf59
4c	dec1	2dde49b7	4bbcd0f3b31f88e0
fc	f272	3f9b3e84	1b7d40da69970634
f8	e898	2fd98015	68ce96ea3d0605dd
ad	e047	8d5d71c8	d42ff6393766c90a

More generally, and although we will not use it as such in the ConSum cipher, we can define the *Conway product* of two bit strings of equal length 2^n, x and y, written $x \circledast y$, as follows:

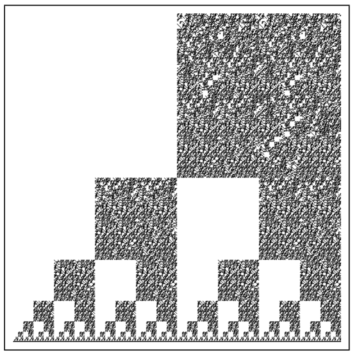
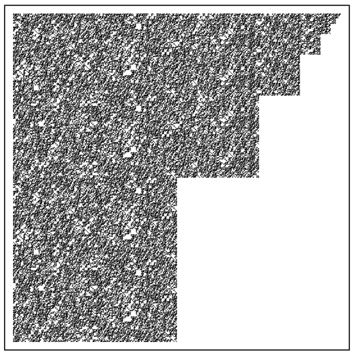
$$\begin{aligned} \mathtt{0} \circledast \mathtt{0} = \mathtt{0}, \quad \mathtt{0} \circledast \mathtt{1} = \mathtt{0}, \quad \mathtt{1} \circledast \mathtt{0} = \mathtt{0}, \quad \mathtt{1} \circledast \mathtt{1} = \mathtt{1} \\ (x_1|x_0) \circledast (y_1|y_0) = \quad & ((x_1 \circledast y_0) \oplus (x_0 \circledast y_1) \oplus (x_1 \circledast y_1)) \\ & \mid ((x_0 \circledast y_0) \oplus K(x_1 \circledast y_1)) \end{aligned}$$

The following facts hold:

Facts 1. *The set $\{\mathtt{0},\mathtt{1}\}^{2^n}$ of strings of length 2^n, with the two binary composition maps $\oplus$ (eXclusive OR) and $\circledast$ (Conway product), and unit elements 0 (zero bit string) and 1 (bit string whose least significant bit is* $\mathtt{1}$ *and all others* $\mathtt{0}$*), is a field isomorphic to $\mathbb{F}_{2^{2^n}}$ (the Galois field with 2^{2^n} elements — which is unique). The Conway transformation is then simply $Kx = x \circledast K1$.*

In particular, we emphasize that the Conway transformation K is *linear* over the field $\mathbb{F}_2$, that is, $K(x \oplus y) = Kx \oplus Ky$; in fact, K is linear over $\mathbb{F}_{2^{2^n}}$ in the sense that $K(z \circledast x) = z \circledast Kx$.

We close this section with the following diagrams, which represent the matrix of K and of K^{-1} on 512 bits over the field $\mathbb{F}_2$ with two elements (0 being represented as white and 1 as black).

3.2 The Conway transformation as a permutation

To avoid confusion, let us presently write K_n for the Conway transformation on bit strings of length $N = 2^n$. From the facts stated previously, we know that $K_n(x) = x \circledast K_n(1)$ where $(\{\mathtt{0}, \mathtt{1}\}^{2^n}, \oplus, \circledast) \cong \mathbb{F}_{2^{2^n}}$, and (since $K_n(1)$ is obviously not equal to 1, at least so long as $n > 0$) this implies that K_n is a permutation with no fixed points other than 0, all of whose orbits (other than $\{0\}$) have the same cardinality, namely the multiplicative order of the element $\kappa_n = K_n(1)$ of the field. We now try to say a little more about this order.

To make notations more readable, we temporarily replace the symbols $\oplus$ and $\circledast$ by the more usual $+$ and $\cdot$ (or an absence of symbol) and we agree to identify $\mathbb{F}_{2^{2^n}}$ with the set of bit strings of length $N = 2^n$ with these operations; we also identify $\mathbb{F}_{2^{2^{n-1}}}$ (bit strings of length $N/2 = 2^{n-1}$) with those bit strings of length N whose upper half bits are all $\mathtt{0}$.

It is easy to check that the multiplicative orders of κ_2 and κ_3 are equal to 5 and 85.

Now notice that $\kappa_n = (\kappa_{n-1}|0)$, and if we put $\bar{\kappa}_n = (\kappa_{n-1}|\kappa_{n-1})$ then we have $\kappa_n + \bar{\kappa}_n = (0|\kappa_{n-1})$ and $\kappa_n\bar{\kappa}_n = (0|\kappa_{n-1}^3)$ (where $\kappa_{n-1}^3 = K_{n-1}^2(\kappa_{n-1})$). This implies that κ_n and $\bar{\kappa}_n$ are the roots of the quadratic equation $X^2 + \kappa_{n-1}X + \kappa_{n-1}^3$ over the subfield $\mathbb{F}_{2^{2^{n-1}}}$. Since we know that the Galois group of $\mathbb{F}_{2^{2^n}}$ over $\mathbb{F}_{2^{2^{n-1}}}$ consists of the identity and the "Frobenius" map $x \mapsto x^{2^{2^{n-1}}}$, we have shown $\bar{\kappa}_n = (\kappa_n)^{2^{2^{n-1}}}$, and $(\kappa_n)^{2^{2^{n-1}}+1} = \kappa_n\bar{\kappa}_n = \kappa_{n-1}^3$.

Now let us use this to compute the order of κ_4: clearly it must divide $2^{2^4} - 1 = 65\,535 = 3 \times 5 \times 17 \times 257$. Since κ_4 does not lie in the subfield $\mathbb{F}_{2^{2^3}}$ of $\mathbb{F}_{2^{2^4}}$, its order cannot divide $255 = 3 \times 5 \times 17$, so it must be a multiple of 257. But we have also just shown that $\kappa_4^{257} = \kappa_3^3$, which is of order 85 (like κ_3 itself). Consequently, κ_4 is of order $85 \times 257 = \frac{1}{3}(2^{16} - 1)$.

Similarly, the order of κ_5 is a divisor of $2^{2^5} - 1 = 3 \times 5 \times 17 \times 257 \times 65\,537$ which is not a divisor of $65\,535$, so it must be a multiple of $65\,537$, and since $\kappa_5^{65\,537} = \kappa_4^3$, we see that the order of κ_5 is exactly $\frac{1}{3}(2^{32} - 1)$.

Concerning the order of κ_6 we encounter a difficulty: this time $2^{32} + 1$ is not prime, it is $641 \times 6\,700\,417$. So while we again have $\kappa_6^{2^{32}+1} = \kappa_5^3$ and $\kappa_6^{2^{32}-1} \neq 1$, we cannot immediately conclude concerning the order of κ_6: it divides $\frac{1}{3}(2^{64} - 1)$, but it could also be $641 \times \frac{1}{3}(2^{32} - 1)$ or $6\,700\,417 \times \frac{1}{3}(2^{32} - 1)$. However, it can be checked that $\kappa_6^{641} =$ `f79e0da1309d2d57` and $\kappa_6^{6\,700\,417} =$ `9b3606c3047014f6`: since neither is in $\mathbb{F}_{2^{2^5}}$, it follows that κ_6 is of order $\frac{1}{3}(2^{64} - 1)$ exactly.

So far we have seen that κ_n (and hence, K_n) is of order $\frac{1}{3}(2^{2^n} - 1)$ for $2 \leq n \leq 6$. It is always true that the order *divides* $\frac{1}{3}(2^{2^n} - 1)$, but we do not know whether it is always equal. It is clear, however, always because $\kappa_n^{2^{2^{n-1}}+1} = \kappa_{n-1}^3$, that the order of κ_n is always a multiple of that of κ_{n-1}, and must be greater: a lower bound is given by the order of κ_{n-1} multiplied by the smallest prime factor of $2^{2^{n-1}} + 1$.

We have not checked the order of κ_7, respectively κ_8, but by the argument above it must be a multiple of $\frac{274\,177}{3}(2^{64} - 1)$ (which is over 2^{80}), respectively over 2^{136}. It is not directly relevant to ConSum to know exactly what the order is: our point is merely to explain why the Conway transformation does not have certain undesirable properties such as fixed points or very small orbits.

3.3 Computation considerations

In hardware: Recursively applying the definition of the Conway transformation gives a circuit that computes the Conway transformation K_n of a $N = 2^n$-bit string using $\leq N^{\frac{\log 3}{\log 2}} \leq N^{1.6}$ XOR gates (with two inputs each) and $\leq N$ depth.

In software: One would typically implement the Conway transformation by a number of functions for successive bit sizes: starting with a table lookup for the Conway transformation of 8-bit strings (or perhaps 4-bit strings on certain embedded system) and implementing the 2^n-bit Conway transformation through 3 calls to the 2^{n-1}-bit function. Experiments on a Pentium® D processor (running at 3.40GHz) in 64-bit mode suggests that it can take the following number of clock cycles to perform Conway transformations of various sizes:

Bit size	Length
16 bits	$10\frac{1}{2}$ cycles
32 bits	26 cycles
64 bits	$75\frac{1}{4}$ cycles
128 bits	$226\frac{1}{2}$ cycles

Our experiments also suggest that it is not advantageous to perform the 16-bit Conway transformation by table lookup; however, it is advantageous to precompute the 8-bit Conway transformation iterated twice.

4 The ConSum cipher

4.1 Description

The ConSum cipher depends on three parameters: an integer n_{b} giving the block size $N_{\mathrm{b}} = 2^{n_{\mathrm{b}}}$, an integer n_{k} giving the key length $N_{\mathrm{k}} = 2^{n_{\mathrm{k}}}$, and an integer r which is the number of rounds. The only condition we impose is $n_{\mathrm{k}} \geq n_{\mathrm{b}}$, although, of course, r should be reasonably large if we want any kind of security. The "standard" values shall be: $n_{\mathrm{b}} = n_{\mathrm{k}} = 7$ (so we use 128-bit keys and blocks) and $r = 10$.

Suppose we are given a $2^{n_{\mathrm{b}}}$-bit string x (the plaintext) and $r + 1$ strings $t^{(0)}, \ldots, t^{(r)}$ (the subkeys) of that same length. Then we define

$$\begin{aligned} y^{(0)} &= x \boxplus t^{(0)} \\ y^{(1)} &= Ky^{(0)} \boxplus t^{(1)} \\ y^{(2)} &= Ky^{(1)} \boxplus t^{(2)} \\ &\cdots \\ y = y^{(r)} &= Ky^{(r-1)} \boxplus t^{(r)} \end{aligned}$$

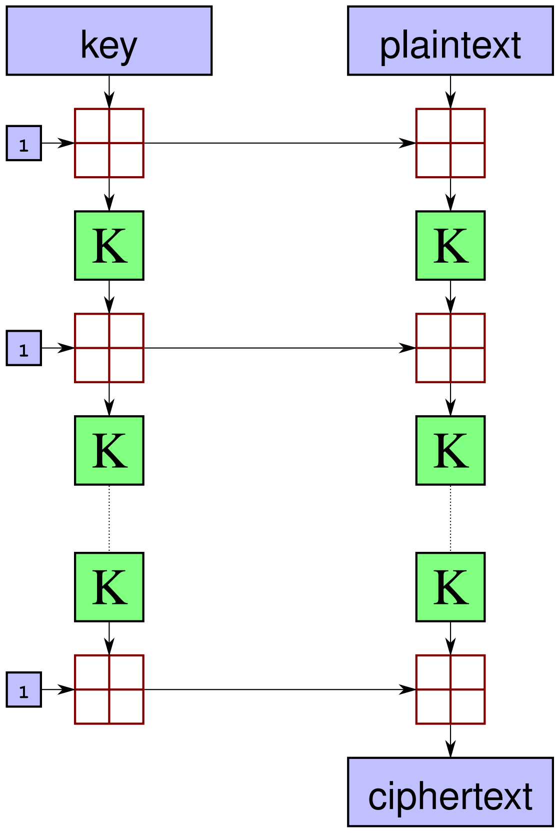
And the ciphertext is $y = y^{(r)}$. (Note that the number of rounds r is equal to the number of Conway transformations used: the number of additions is $r + 1$.)

It remains to explain the key schedule, i.e. how we define the subkeys $t^{(0)}, \ldots, t^{(r)}$ from the key z. This is also easy: it follows *precisely the same* formula, with z instead of x, $t^{(i)}$ instead of $y^{(i)}$ and 1 instead of $t^{(i)}$, that is:

$$\begin{aligned} t^{(0)} &= z \boxplus 1 \\ t^{(1)} &= Kt^{(0)} \boxplus 1 \\ t^{(2)} &= Kt^{(1)} \boxplus 1 \\ &\cdots \\ t^{(r)} &= Kt^{(r-1)} \boxplus 1 \end{aligned}$$

In other words, the subkeys are generated by taking the intermediate values of the encryption of the key with subkeys all equal to 1.

The following diagram recapitulates the overall encryption process:



Since we have allowed the key length N_{k} to be greater than the block size N_{b}, we simply reduce the subkeys (forget their $N_{\mathrm{b}} - N_{\mathrm{k}}$ most significant bits, keeping only the low end) to the desired size.

4.2 Computational considerations

Using a single core of a Pentium® D processor (running at 3.40GHz) in 64-bit mode using a straightforward implementation of ConSum(v0, b128, k256, r10) written in (portable) C, we found it to encrypt at circa $22\,\mathrm{MB}/s$ (millions of bytes per second). This is equivalent to the speed of OpenSSL's optimized implementation of Triple-DES on the same machine: so ConSum is quite slow, but not slow to the point of being unusable. Since it is meant as an experimental cipher and not immediately for production use, and since the bit size, key length and number of rounds are free (and adequate numbers have yet to be investigated), there is no sense in prematurely trying to fine-tune it for optimization.

5 Addition and XOR

5.1 The carry structure

Assume that x and y are bit strings of length N (here we do not need that N be a power of two). Define $C(x,y)$ as the "carry structure" of x and y: that is, $C(x,y)$ is an N-bit string whose k-th bit is 1 when adding the k-th bits of x and y produced a carry (to be added to the $(k+1)$-th bit), and 0 otherwise. Formally:

$$\begin{array}{c} C(\mathtt{0},\mathtt{0})=\mathtt{0},\quad C(\mathtt{0},\mathtt{1})=\mathtt{0},\quad C(\mathtt{1},\mathtt{0})=\mathtt{0},\quad C(\mathtt{1},\mathtt{1})=\mathtt{1} \\ C((x_1|x_0),(y_1|y_0))=(C(x_1,y_1)|C(x_0,y_0)) \text{ if } c(x_0,y_0)=\mathtt{0} \\ C((x_1|x_0),(y_1|y_0))=(\hat{C}(x_1,y_1)|C(x_0,y_0)) \text{ if } c(x_0,y_0)=\mathtt{1} \\ \hat{C}(\mathtt{0},\mathtt{0})=\mathtt{0},\quad \hat{C}(\mathtt{0},\mathtt{1})=\mathtt{1},\quad \hat{C}(\mathtt{1},\mathtt{0})=\mathtt{1},\quad \hat{C}(\mathtt{1},\mathtt{1})=\mathtt{1} \\ \hat{C}((x_1|x_0),(y_1|y_0))=(C(x_1,y_1)|\hat{C}(x_0,y_0)) \text{ if } \hat{c}(x_0,y_0)=\mathtt{0} \\ \hat{C}((x_1|x_0),(y_1|y_0))=(\hat{C}(x_1,y_1)|\hat{C}(x_0,y_0)) \text{ if } \hat{c}(x_0,y_0)=\mathtt{1} \end{array} \tag{4}$$

where c and $\hat{c}$ have already been defined in (1): they are simply the top bits of C and $\hat{C}$ respectively.

More usefully, the k-th bit of $C(x,y)$ is the logical AND of the k-th bit of x and y when the $(k-1)$-th bit of $C(x,y)$ is 0, the logical OR when the $(k-1)$-th bit of $C(x,y)$ is 1, with the convention that the (-1)-th bit of $C(x,y)$ is always 0; and $\hat{C}(x,y)$ is nearly the same except for this last convention which is reversed.

So we can write

$$x \boxplus y = x \oplus y \oplus (C(x,y) \ll 1) \tag{5}$$

where $\ll 1$ denotes a left bit shift by one bit (with the top bit thrown away): $x \ll 1 = x \boxplus x = x|\mathtt{0}$ (note that $C(x,x)=x$).

5.2 The addition entropy

Now fix x and let y vary. Precisely, let y be *random*, with equal probability for each of the 2^N possible bit strings. And consider the possible values of $C(x,y)$

and their different probabilities. We consider the sum of the $-p \log p$ for each such probability p (where the $\log$ is taken in base 2), and we call it the *addition entropy*, written $S(x)$, of x (expressed in logons). This is the "amount of randomness" in $C(x, y)$ if y is random (whereas x is the given string): $S(x)$ is the amount of information needed to specify $C(x, y)$ when y is chosen at random (and x is given).

Here is an example. Take $x = 1$. Then $C(x, y)$ is 0 if y ends in `0` (probability $p = \frac{1}{2}$). It is 1 if y ends in `01` (probability $p = \frac{1}{4}$). It is $3 = \texttt{11}$ if y ends in `011` (probability $p = \frac{1}{8}$) and so on. So the addition entropy of $x = 1$ is $S(1) = 1 \times \frac{1}{2} + 2 \times \frac{1}{4} + 3 \times \frac{1}{8} + \cdots + N \times \frac{1}{2^N} + N \times \frac{1}{2^N} = 2 - \frac{1}{2^{N-1}}$. Naturally, the addition entropy $S(0)$ of 0 is 0.

We can similarly define $\hat{S}(x)$ to be the entropy of $\hat{C}(x, y)$ for random y; in fact, it is easy to see that, for any x, we have $\hat{S}(x) = S(x \boxplus 1)$ (although it is, of course, *not* true that $\hat{C}(x, y) = C(x \boxplus 1, y)$ in general!).

Addition entropies are easily computed inductively. Indeed, consider an N-bit string x whose addition entropy $S(x)$ is known, and we wish to compute that of $(N + 1)$-bit strings starting with x. First consider that of $x|\texttt{0}$ (the string obtained by adding a `0` at the end of x — remember that this is the least significant bit): evidently bit 0 will never have any carry on it, so that for any bit b, $C(x|\texttt{0}, y|b) = C(x, y)|\texttt{0}$; thus, $S(x|\texttt{0}) = S(x)$. In the same way, $\hat{S}(x|\texttt{1}) = \hat{S}(x)$ because $\hat{C}(x|\texttt{1}, y|b) = \hat{C}(x, y)|\texttt{1}$. On the other hand, let us consider $S(x|\texttt{1})$. Now we have $C(x|\texttt{1}, y|\texttt{0}) = C(x, y)|\texttt{0}$ and $C(x|\texttt{1}, y|\texttt{1}) = \hat{C}(x, y)|\texttt{1}$, each case having probability $\frac{1}{2}$; and the resulting string sets are *disjoint*; this proves that $S(x|\texttt{1}) = 1 + \frac{1}{2}(S(x) + \hat{S}(x))$. And similarly $\hat{S}(x|\texttt{0}) = 1 + \frac{1}{2}(S(x) + \hat{S}(x)) = S(x|\texttt{1})$. Finally we have shown:

Proposition 2. *The addition entropy S and the addition entropy with carry $\hat{S}$ are subject to the following induction relations:*

$$\begin{array}{cc}
S(\texttt{0}) = 0 & S(\texttt{1}) = 1 \\
\hat{S}(\texttt{0}) = 1 & \hat{S}(\texttt{1}) = 0 \\
S(x|\texttt{0}) = S(x) & S(x|\texttt{1}) = 1 + \frac{1}{2}(S(x) + \hat{S}(x)) \\
\hat{S}(x|\texttt{0}) = 1 + \frac{1}{2}(S(x) + \hat{S}(x)) & \hat{S}(x|\texttt{1}) = \hat{S}(x)
\end{array}$$

Now consider the mean addition entropy S_N of all N-bit strings (that is, the average of the $S(x)$, where x ranges over all N-bit strings; or again, the expectancy of the addition entropy where x is a random N-bit string). Since $\hat{S}(x) = S(x \boxplus 1)$ as we have noted (even where x is the string -1 consisting of only 1's), and since $x \boxplus 1$ goes through every N-bit string exacly once as x does, it follows that $\hat{S}_N = S_N$ (with the obvious notation). On the other hand, the induction relations we have seen show, when averaging, that $S_1 = \frac{1}{2}$, and $S_{N+1} = \frac{1}{2} + S_N$. Thus,

Corollary 3. *The mean addition entropy S_N of all N-bit strings and the mean addition entropy with carry $\hat{S}_N$ are given by:*

$$S_N = \hat{S}_N = \frac{N}{2}$$

This says that for a given (known) but random x, one needs on the average $\frac{N}{2}$ logons of information to specify the value of $C(x, y)$ for a random y. One would like to say that because of (5) it also takes $\frac{N}{2}$ logons to specify the value of $x \boxplus y$ (together with the topmost carry) when the value of $x \oplus y$ is known, but that is nonsense: when $x \oplus y$ is known, so is y and therefore $x \boxplus y$; however, it is true (possibly with a subtlety for the topmost bit) that it takes $\frac{N}{2}$ logons to specify the value of $(x \boxplus y) \oplus z$ when the value of $x \oplus y \oplus z$ is known, for y and z both random and unknown.

Note that when x and y are both random and unknown, it takes not $\frac{N}{2}$ logons but slightly (62%) more, namely $\left(2 - \frac{3}{4}\log 3\right) N$, to give the value of $C(x, y)$ (the proof in a nutshell: giving the lowest bit of $C(x, y)$ takes $2 - \frac{3}{4}\log 3$ logons because it is the logical AND of the lowest bits of x and y; after that, each bit of $C(x, y)$ also takes $2 - \frac{3}{4}\log 3$ logons because it is either the logical AND or the logical OR, the which of which is known by the previous bit). Consequently, it takes $\left(2 - \frac{3}{4}\log 3\right) N$ logons to specify the value of $x \boxplus y$ (together with the topmost carry) when the value of $x \oplus y$ is known. However, this is of lesser interest to us.

Note also that in all this, forgetting the topmost carry bit (as we prescribed) decreases the specified entropies by *at most* one logon (since a bit cannot contain more than that much information).

5.3 Relevance to ConSum

We now explain heuristically why the previous section's considerations on addition entropies are relevant to ConSum's security. Evidently, if we replace ordinary addition (with carry) by eXclusive OR (addition without carry) in ConSum's design, we get no security at all (since K distributes over XOR), and it is in the departure of $x \boxplus y$ from $x \oplus y$ that all the security lies.

Consider the operation (left-conjugation by K^s of addition) defined by $x \boxplus^{(s)} u = K^{-s}(K^s x \boxplus u)$. In other words, make K^s act the left operand, add, and then undo the K^s. What ConSum's round function amounts to, then, is to take start with a plaintext x and calculate $(\cdots((x \boxplus t^{(0)}) \boxplus^{(1)} t^{(1)}) \cdots) \boxplus^{(r)} t^{(r)}$, where $t^{(s)}$ is a subkey. Now because of (5), we have $x \boxplus^{(s)} u = x \oplus K^{-s}u \oplus K^{-s}\tilde{C}(K^s x, u)$ (where we have set $\tilde{C}(x', u') = C(x', u') \ll 1$ for simplicity's sake). So ConSum's round function XORs x with (a) the XOR of $t^{(0)}$,..., $K^{-r}t^{(r)}$, which do not depend on x, and (b) the XOR of $C(x, t^{(0)})$, $K^{-1}C(Kx, t^{(1)})$,..., $K^{-r}C(K^r x, t^{(r)})$, which have respective entropies (for random x) of at least $S(t^{(0)}) - 1$, $S(t^{(1)}) - 1$,..., $S(t^{(r)}) - 1$ (the -1 comes from the fact that we throw away the top bit). We are in danger of attacks, therefore, at least when the sum of these entropies is less than N. But since we have proven that for a random u, $S(u)$ is on the average $\frac{N}{2}$, we are far above this figure even for small numbers of rounds.

6 Statistics and cryptanalysis of 8-bit ConSum

6.1 Bernoulli distribution of ciphertexts

In an ideal cipher, $E(z, x)$, where z is the key and x the plaintext, is a randomly and independently chosen permutation of x for each z; but if we fix the plaintext x and let the key z vary, each $E(z, x)$ should be independent: for a given plaintext x, the number of occurrences of each possible value of $E(z, x)$ when z ranges through a certain set of keys should follow a Bernoulli distribution (with average 1 if there are as many ciphertexts as keys). We emphasize this: whereas $x \mapsto E(z, x)$ (fix the key and let the plaintext vary) is a permutation, so each ciphertext occurs once and only once, $z \mapsto E(z, x)$ should not be biased in its distribution—this is an important condition in resisting attacks that try to recover the key.

We have performed the following statistical test on ConSum(b8,r4) and ConSum(b8,r5) (that is, the version of ConSum with 8-bit keys and blocks and only 4 or 5 rounds): for each given plaintext x we have computed the number of ciphertexts which are reached 0 through 7 times (more than 7 never occurs) by $E(x, z)$ when z varies through all (256) keys, then we have summed this over all (256) values of x. The result is as follows:

Count	Ideal	ConSum r4	ConSum r5
0	24 062	24 067	24 008
1	24 157	24 174	24 212
2	12 078	11 999	12 124
3	4 010	4 083	3 974
4	995	990	990
5	197	186	183
6	32	34	36
7	5	3	9

For example, the 186 in the "ConSum" column for count 5 means that there are 186 $(\text{plaintext}, \text{ciphertext})$ pairs such that the ciphertext occurred as the encoding of plaintext with 5 different keys. The "Ideal" column was obtained by rounding $65\,536 \times \frac{256!}{k!(256-k)!} \times (\frac{255}{256})^{256-k} \times (\frac{1}{256})^k$ to the closest integer (where k is the count).

The values in this table indicate that ConSum is not biased, even for a small number of rounds, toward or against producing the same ciphertext for the same plaintext when the key varies.

6.2 Bit change correlations

6.2.1 Plaintext to ciphertext

We have performed the following test on ConSum(b8,r4) and ConSum(b8,r5): for each one of 8×8 combinations of a plaintext bit position u and a ciphertext bit position v, we have examined how many out of the 256×128 $(\text{key}, \text{plaintext}_1, \text{plaintext}_2)$ triplets, where the two plaintexts differ exactly at bit position u, give rise to ciphertexts pairs $(\text{ciphertext}_1, \text{ciphertext}_2)$ in which bit position v differs. Now for each given bit v, the proportion of pairs of bytes (y_1, y_2) where $y_1 \neq y_2$ which differ in bit v is $p = \frac{128}{255}$ (this is a little over $\frac{1}{2}$ because $y_1 \neq y_2$ implies they must differ by at least one bit). So for an ideal cipher we expect in our experiment an average of p, with a variance of $p(1-p)$, for each (u, v) pair. Here are the observed results, expressed in expected standard deviations ($\sqrt{256 \times 128 \times p(1-p)} = 90.509$) relative to the expected mean ($256 \times 128 \times p = 16448.25$); we emphasize that these results are bit *change* correlations, not absolute bit correlations (so the value 2.21 in line $u = 1$ and column $v = 2$ means that two plaintexts which differ only in bit 1 have a rather high tendency, 2.21 deviations above the expected value, of producing ciphertexts in which bit 2 differs):

r4	$v=0$	$v=1$	$v=2$	$v=3$	$v=4$	$v=5$	$v=6$	$v=7$
$u=0$	1.28	0.06	−3.32	2.74	−1.48	2.05	−1.77	−0.69
$u=1$	2.05	7.91	2.21	1.90	−1.28	−1.51	−0.51	−0.14
$u=2$	2.89	−1.00	−0.53	0.26	0.39	−3.98	−1.37	0.13
$u=3$	0.39	1.99	−2.81	0.90	−0.27	0.97	0.86	0.04
$u=4$	2.43	−2.41	−1.84	−1.57	0.97	0.17	−0.91	1.43
$u=5$	−1.02	2.16	−0.29	−1.93	3.42	−0.67	0.64	−1.51
$u=6$	−0.78	2.34	2.18	0.70	−0.11	−1.31	1.77	2.34
$u=7$	−1.62	0.37	−0.44	−0.60	0.48	−0.02	2.21	−1.06

r5	$v=0$	$v=1$	$v=2$	$v=3$	$v=4$	$v=5$	$v=6$	$v=7$
$u=0$	−0.51	1.61	−0.38	1.61	0.42	−0.64	1.46	−0.38
$u=1$	0.33	−0.18	0.28	−1.24	0.28	−0.56	1.10	1.70
$u=2$	−2.48	2.14	0.66	−1.68	−3.25	−0.73	−0.98	−1.75
$u=3$	−0.07	−0.29	0.81	1.23	−1.62	1.79	0.64	−0.31
$u=4$	−0.31	0.62	0.64	−0.22	1.26	0.42	0.95	−1.11
$u=5$	1.10	1.04	0.53	1.92	−0.14	−0.93	0.26	2.23
$u=6$	1.23	0.75	0.84	0.06	−0.22	0.20	−1.06	−0.22
$u=7$	−0.00	0.48	0.51	0.13	0.64	0.39	−0.18	0.99

The first table, for ConSum(b8,r4), having sum of squares $= 228.7$, presents significant deviation from an ideal table (for 64 gaussian variables with average 0 and standard deviation 1 we expect a sum of squares of 64 ± 11.3). The second table, for ConSum(b8,r5), however, presents no noticeable anomaly, having a sum of squares $= 75.5$. This is true for all further numbers of rounds:

Rounds	$\sum(\xi^2)$	Norm.
r3	1 479.0	125.07
r4	228.7	14.56
r5	75.5	1.02
r6	50.7	−1.18
r7	67.9	0.35
r8	39.7	−2.14
r9	69.5	0.48
r10	69.8	0.51

(The “Norm.” column is simply the value of the colum $\sum(\xi^2)$ minus its expected mean 64 and divided by its expected standard deviation 11.3.)

The values in this table indicate that ConSum is not noticeably biased, from five rounds on, toward correlating bit changes in the ciphertext to single bit changes in the plaintext.

6.2.2 Key to ciphertext

The test described here is analogous to the previous one, except that instead of varying a bit in the plaintext we vary it in the key.

In other words: for each one of 8×8 combinations of a key bit position u_{k} and a ciphertext bit position v, we have examined how many out of the 256×128 $(\mathrm{key}_1, \mathrm{key}_2, \mathrm{plaintext})$ triplets, where the two keys differ exactly at bit position u_{k}, give rise to ciphertexts pairs $(\mathrm{ciphertext}_1, \mathrm{ciphertext}_2)$ in which bit position v differs. This time the ideal proportion is $p = \frac{1}{2}$ because there is no reason to exclude the two ciphertexts being equal. Here are the observed results, expressed in expected standard deviations ($\sqrt{256 \times 128 \times p(1-p)} = 90.510$) relative to the expected mean ($256 \times 128 \times p = 16384$):

r4	$v=0$	$v=1$	$v=2$	$v=3$	$v=4$	$v=5$	$v=6$	$v=7$
$u_{\mathrm{k}}=0$	-0.51	-0.46	-2.21	-2.89	1.30	0.29	-0.51	-1.46
$u_{\mathrm{k}}=1$	-1.02	-1.17	0.57	0.29	1.28	-1.02	-1.37	-0.15
$u_{\mathrm{k}}=2$	2.47	1.10	-0.75	0.64	-0.15	-0.04	-0.60	1.06
$u_{\mathrm{k}}=3$	0.62	-1.86	-0.09	0.66	-0.95	0.18	-0.04	2.67
$u_{\mathrm{k}}=4$	0.40	-0.80	0.97	0.38	-0.49	-1.83	0.42	-0.93
$u_{\mathrm{k}}=5$	1.81	0.97	1.59	-0.09	0.33	0.20	-0.53	1.61
$u_{\mathrm{k}}=6$	-2.06	0.71	2.01	0.22	-0.51	-1.90	0.33	-2.72
$u_{\mathrm{k}}=7$	0.57	0.49	0.44	1.17	-0.18	-0.38	-0.84	0.11

r5	$v=0$	$v=1$	$v=2$	$v=3$	$v=4$	$v=5$	$v=6$	$v=7$
$u_{\mathrm{k}}=0$	0.20	1.94	-1.59	0.46	1.55	1.02	-2.30	-0.09
$u_{\mathrm{k}}=1$	-1.68	0.66	0.02	-0.02	-1.48	1.04	0.69	0.15
$u_{\mathrm{k}}=2$	1.39	-1.83	0.49	-1.13	-0.42	0.71	-0.77	-0.24
$u_{\mathrm{k}}=3$	-1.06	-0.13	-1.70	-0.40	-1.41	0.69	0.84	-1.75
$u_{\mathrm{k}}=4$	-1.17	0.93	-0.40	-0.51	1.06	-1.37	-0.20	0.57
$u_{\mathrm{k}}=5$	1.30	-0.66	-2.25	-0.51	-1.66	0.71	-0.55	0.69
$u_{\mathrm{k}}=6$	-1.33	1.99	0.27	-0.75	-0.02	-1.55	-0.93	1.35
$u_{\mathrm{k}}=7$	0.24	0.09	-1.77	-1.44	-0.04	-0.33	-0.53	-0.49

Rounds	$\sum(\xi^2)$	Norm.
r3	629.3	49.96
r4	87.0	2.03
r5	75.7	1.03
r6	47.8	−1.44
r7	68.0	0.36
r8	53.7	−0.91
r9	63.9	−0.01
r10	45.9	−1.60

The values in this table indicate that ConSum is not noticeably biased, from four rounds on, toward correlating bit changes in the ciphertext to single bit changes in the key.

6.3 Correlation of $\boxplus$-differentials

6.3.1 Plaintext to ciphertext

We now consider a table analogous to the one described in section 6.2.1, except that it is a 255×255 table which, for each possible non-zero differential u (between 1 and 255) in the plaintext and each possible non-zero differential v (between 1 and 255) in the ciphertext, records how many out of the 256×256 $(\text{key}, \text{plaintext}_1, \text{plaintext}_2)$ triplets where $\text{plaintext}_2 = \text{plaintext}_1 \boxplus u$ have $\text{ciphertext}_2 = \text{ciphertext}_1 \boxplus v$.

In other words, the table counts how often a certain $\boxplus$-differential u in the plaintext gives rise to a certain $\boxplus$-differential v in the ciphertext. For an ideal cipher, we expect a mean count of $256 \times 256 \times p = 257.00$ (with $p = \frac{1}{255}$) for each entry in the table, with standard deviation $\sqrt{256 \times 256 \times p(1-p)} = 16.000$, so we renormalize the table by these values. Naturally, given its size, we will not give the entire table, but as previously we tabulate its sum of squares, and the same normalized to account for an expected $65\,025 \pm 360.6$, for various round values:

Rounds	$\sum(\xi^2)$	Norm.
r3	550 027	1345
r4	100 153	97.41
r5	68 875	10.68
r6	67 116	5.80
r7	65 980	2.65
r8	65 495	1.30
r9	65 060	0.10
r10	64 384	−1.78

The results may appear bad as we have detected deviations from a statistically ideal cipher up to six rounds of ConSum. However, this test is extremely stringent since it examins *all* plaintext differentials versus *all* ciphertext differentials (something not at all realizable for larger block sizes): so it is not surprising to detect very subtle levels of correlation.

6.3.2 Key to ciphertext

This time the table is 255×256, because for each of the 255 non-zero key differentials u_{k} any one of the 256 ciphertext differentials v is possible and should be equally likely: the table counts how often a certain $\boxplus$-differential u_{k} in the key gives rise to a certain $\boxplus$-differential v in the ciphertext.

Rounds	$\sum(\xi^2)$	Norm.
r3	76 861	32.05
r4	65 990	1.97
r5	66 825	4.28
r6	65 360	0.22
r7	65 996	1.98
r8	64 879	−1.11
r9	65 312	0.09
r10	65 137	−0.40

Here we find no meaningful correlation from six rounds on (again, this is a very stringent test, as it examines all possible key differentials).

6.4 Correlation of $\oplus$-differentials

We now proceed with $\oplus$-differentials exactly as we did with $\boxplus$-differentials. Only this time we must remember that $E(z, x \oplus u) = E(z, x) \oplus v$ is tantamount to

$E(z, x' \oplus u) = E(z, x') \oplus v$ where $x' = x \oplus u$ (for any function E): hence if we count (for each given pair (u, v)) every pair (z, x) such that $E(z, x \oplus u) = E(z, x) \oplus v$ we expect 257.00 ± 22.627 (not 257.00 ± 16.000 as for $\boxplus$-differentials).

Plaintext to ciphertext:

Rounds	$\sum(\xi^2)$	Norm.
r3	186 980	338
r4	79 678	40.63
r5	68 716	10.23
r6	65 170	0.40
r7	65 869	2.34
r8	65 227	0.56
r9	65 726	1.95
r10	64 796	-0.64

Key to ciphertext:

Rounds	$\sum(\xi^2)$	Norm.
r3	75 866	29.30
r4	64 891	-1.08
r5	65 606	0.90
r6	64 994	-0.79
r7	65 591	0.87
r8	65 263	-0.05
r9	66 081	2.22
r10	65 008	-0.75

6.5 Why analyze only 8-bit ConSum?

We anticipate the following objection: while we have given reasons to think that 8-bit ConSum behaves essentially like an ideal (i.e., true random) cipher at least from seven rounds on, this does not portend much for 64-bit, 128-bit or 256-bit ConSum, the sizes one is generally interested in. In response to this, we argue that, in ConSum, there is no particular reason to increase the number of rounds as the key length, or block size, goes up. Indeed, this is a precautionary measure un ciphers for which the round function does not perform a full "mixing" of the bits, so time must be allowed for every bit to interact with every other one. But in the case of ConSum, the mixing is provided by the Conway transformation and by the carry bits: and certainly the Conway transformation "mixes" on the whole width of the bit strings (this is intuitive in the diagrams showing its matrix, and the fact that it has no small periods also goes in this direction); as for carry bits, we have argued in section 5 that the amount of entropy they bear is proportional to the string length. So there is no particular reason to increase the number of rounds when we have more bits of input (basically, the Conway operation just becomes more complicated).

On the other hand, 8-bit ConSum provides us with a toy-model for the larger ConSum sizes, on which we can perform statistical tests of some significance (4 bits is too small, 16 bits is already too large).

7 Tables and test vectors

7.1 Eight-bit Conway transformation

The following table lists the value of Kx (the Conway transformation of x) for all 8-bit values x; here, the line determines the more significant hexadecimal digit of x, and the column the less significant digit.

	$x_1$0	$x_1$1	$x_1$2	$x_1$3	$x_1$4	$x_1$5	$x_1$6	$x_1$7	$x_1$8	$x_1$9	x_1a	x_1b	x_1c	x_1d	x_1e	x_1f
0x_0	00	80	c0	40	b0	30	70	f0	d0	50	10	90	60	e0	a0	20
1x_0	8d	0d	4d	cd	3d	bd	fd	7d	5d	dd	9d	1d	ed	6d	2d	ad
2x_0	c6	46	06	86	76	f6	b6	36	16	96	d6	56	a6	26	66	e6
3x_0	4b	cb	8b	0b	fb	7b	3b	bb	9b	1b	5b	db	2b	ab	eb	6b
4x_0	b9	39	79	f9	09	89	c9	49	69	e9	a9	29	d9	59	19	99
5x_0	34	b4	f4	74	84	04	44	c4	e4	64	24	a4	54	d4	94	14
6x_0	7f	ff	bf	3f	cf	4f	0f	8f	af	2f	6f	ef	1f	9f	df	5f
7x_0	f2	72	32	b2	42	c2	82	02	22	a2	e2	62	92	12	52	d2
8x_0	de	5e	1e	9e	6e	ee	ae	2e	0e	8e	ce	4e	be	3e	7e	fe
9x_0	53	d3	93	13	e3	63	23	a3	83	03	43	c3	33	b3	f3	73
ax_0	18	98	d8	58	a8	28	68	e8	c8	48	08	88	78	f8	b8	38
bx_0	95	15	55	d5	25	a5	e5	65	45	c5	85	05	f5	75	35	b5
cx_0	67	e7	a7	27	d7	57	17	97	b7	37	77	f7	07	87	c7	47
dx_0	ea	6a	2a	aa	5a	da	9a	1a	3a	ba	fa	7a	8a	0a	4a	ca
ex_0	a1	21	61	e1	11	91	d1	51	71	f1	b1	31	c1	41	01	81
fx_0	2c	ac	ec	6c	9c	1c	5c	dc	fc	7c	3c	bc	4c	cc	8c	0c

7.2 Step-by-step encryption

We perform the encryption of plaintext 0000...0000 (128 bits) with key 0000...0000 in ConSum(v0, b128, k256, r10).

Key schedule:

$t^{(0)}$ =0000000000000000 0000000000000000 0000000000000000 0000000000000001

$t^{(1)}$ =8000000000000000 0000000000000000 0000000000000000 0000000000000001

$t^{(2)}$ =5e4ae3a94a88a921 e3a92850a9218171 4a88a921e2208b6b a92181714b010a1b

$t^{(3)}$ =bde3cbf9e3a92850 c376d7b8fc4df767 e3a92850a9218171 20dfffe8556c7617

$t^{(4)}$ =14c24a88a8a8224a 4a88a921e2208b6b 9f8c3840a5211092 8a8065f6702581f4

$t^{(5)}$ =679c50b6fe2b4f6b 038bd29a4c2a85ac f670bd42668cf89d afc7650022fd8f19
$t^{(6)}$ =6a68e79f65cd77d3 863e9d9e461d73f4 d0a9e13bd61aed60 05a6f4aee59e1cf1
$t^{(7)}$ =33da041e149339c0 3a912275b41863b6 048c255511b09e07 77941d605d6e83a7
$t^{(8)}$ =3949da428b1f529b 7614ac31c747e15f ef39d4887a6e57a9 d27e22db0c2f58db
$t^{(9)}$ =21797023c351872d 85c639150ca2602a 98c8a0e75c8172d0 e9ec6b6796c93d6f
$t^{(10)}$ =5a2bb3684ab96ebf 12c0944f82dcf0e5 c1a17b575e9e7491 b2e70aa12d9738db

Encryption:

$y^{(0)}$ = 0000000000000000 0000000000000001
$y^{(1)}$ = 8000000000000000 0000000000000001
$y^{(2)}$ = a8d38ccb2ca9348d 8ccaa9c1f4228b8c
$y^{(3)}$ = efc998c83a0383f8 dc99505629e6427a
$y^{(4)}$ = 7daf7aac099df886 d1ec1c967687cd52
$y^{(5)}$ = b1bad4d984789a50 e4d667f5e49be585
$y^{(6)}$ = cf3c323d27b3e9f0 03733b882cc77667
$y^{(7)}$ = 6db1a70fffa763cb 23ac09438b54c3b8
$y^{(8)}$ = 80f660ff6fc880da f48eb29892864c6c
$y^{(9)}$ = 8b21980fac55fae4 f7b7173c1527ac40
$y^{(10)}$ = 20e2998f0041115a 3b6bfb505e881ef3

The last line (20e2998f0041115a 3b6bfb505e881ef3) is the ciphertext.

7.3 Repeated encryption

We start with $a_0 = a_1 = a_2 = 00...00$ as 128-bit strings and repeatedly define a_{n+1} as the encryption of a_n with key $a_{n-2}|a_{n-1}$ (that is a_{n-2} as high-order bits and a_{n-1} as low-order bits) using ConSum(v0, b128, k256, r10):

a_3 = 20e2998f0041115a 3b6bfb505e881ef3
a_4 = 407f607817a169b9 114de8c86eae011d
a_5 = 328a1865d48c4500 1c83a1464557a5d0
a_6 = ddf08de099b9f063 21d6321a2f2568d9
a_7 = 26533d04205d5b68 fb130bc2fa055e2f
a_8 = 626e73beafd865ab e5c28b8100fd9f3c
a_9 = 39b78910c3a3704b d88f424023d1875e
a_{10} = 1018adf304fe111c ec7cc35878e98eb2
a_{100} = 21108083fcaeeaf9 76b39f6d3e01ad18
a_{1000} = 9ebcfb6ead11f411 9bef167eb2e44c67
a_{10000} = a00920187851a27d 1bc5268df24a5960
a_{100000} = 4ebf22fec1d25af3 f9e99376a5f670bc
$a_{1000000}$ = 0feaa1d911ba7250 255f1a827e7c938d

7.4 Variation on parameters

We constantly use a key and plaintext of `00...00` (various lengths):
ConSum(v0, b128, k256, r10): `20e2998f0041115a 3b6bfb505e881ef3`
ConSum(v0, b128, k256, r9): `8b21980fac55fae4 f7b7173c1527ac40`
ConSum(v0, b128, k256, r11): `b77cac9ee4e251a1 0d2695d0b093a24d`
ConSum(v0, b128, k256, r12): `b27f1601a8370bed 44b4a89e35826818`
ConSum(v0, b256, k256, r10): `b47258422ff4d2ad a2e286117feecfed 07d77b55f150266a ed944303c92194d0`
ConSum(v0, b256, k512, r10): `72fb12624775873c 9426a190d2bd7053 e9cd0a14ecbbc4d9 95a2467f42def11c`
ConSum(v0, b64, k128, r10): `57fff41ae07852f4`
ConSum(v0, b64, k64, r10): `e8a28731578224ca`
ConSum(v0, b64, k256, r10): `1333b3dbf1b2d9b2`
ConSum(v0, b32, k32, r10): `be92e11f`
ConSum(v0, b16, k16, r10): `8a22`
ConSum(v0, b8, k8, r10): `99`
ConSum(v0, b4, k4, r10): `e`

References

[1] E. Berlekamp, J. H. Conway & R. Guy, *Winning Ways*, Academic Press.

[2] J. H. Conway, *On Numbers and Games*, Academic Press.