Fast Computation of Large Distributions and Its Cryptographic Applications

Alexander Maximov and Thomas Johansson

Dept. of Information Technology, Lund University, Sweden
P.O. Box 118, 221 00 Lund, Sweden
{movax,thomas}@it.lth.se

Abstract. Let $X_1, X_2, \ldots, X_k$ be independent n bit random variables. If they have *arbitrary* distributions, we show how to compute distributions like $\Pr\{X_1 \oplus X_2 \oplus \cdots \oplus X_k\}$ and $\Pr\{X_1 \boxplus X_2 \boxplus \cdots \boxplus X_k\}$ in complexity $O(kn2^n)$. Furthermore, if $X_1, X_2, \ldots, X_k$ are *uniformly* distributed we demonstrate a large class of functions $F(X_1, X_2, \ldots, X_k)$, for which we can compute their distributions efficiently.

These results have applications in linear cryptanalysis of stream ciphers as well as block ciphers. A typical example is the approximation obtained when additions modulo 2^n are replaced by bitwise addition. The efficiency of such an approach is given by the bias of a distribution of the above kind. As an example, we give a new improved distinguishing attack on the stream cipher SNOW 2.0.

Keywords: cryptanalysis, complexity, algorithms, convolution, approximations, large distributions, pseudo-linear functions.

1 Introduction

Linear cryptanalysis is one of the most powerful techniques for cryptanalysis. It can be regarded as a generic attack. It is for example the fastest known attack on DES. More recently, we have seen that linear cryptanalysis also plays a major role in the area of stream ciphers. Many recent proposals have been analyzed through the idea of replacing nonlinear operations by linear ones, and then hoping that obtained linear equations are correct with a probability slightly larger than otherwise expected. Actually, the best known attacks on many recent stream cipher proposals are linear attacks. This includes stream ciphers like Scream [1], SNOW [2,3], SOBER [4,5], RC4 [6], A5/1 [7], and many more.

Most work in linear cryptanalysis on block ciphers are based on bitwise linear approximations. To oversimplify, we find a sum of certain plaintext bits, ciphertext bits and key bits such that this sum is zero with a probability $1/2 + \epsilon$, where ϵ is usually small. By getting access to a large number of different plaintext/ciphertext pairs we can eventually find out the value of the sum of key bits. This results in a key recovery attack.

In linear attacks on stream ciphers, it is mostly the case that a linear approximation will give us a set of keystream symbols that sum to zero with probability

$1/2+\epsilon$. Since no key bits are involved in the expression, this gives us a distinguishing attack. In some linear attacks on stream ciphers, one has moved from the binary alphabet to instead consider a sum of variables defined over a larger set. For example, we can consider a sum of different bytes from keystream sequence if it is byte oriented. Distinguishers based on symbols from a larger alphabet have been considered in for example [8–10].

It is clear that moving to a larger alphabet gives improved results. However, the computational complexity of finding the result increases. To be a bit more specific, assume for example that the operation $X_1 \boxplus X_2$ is replaced by $X_1 \oplus X_2$, where $\boxplus$ denotes mod 2^n addition. The usefulness of such an approximation is given by the distribution $\Pr\{(X_1 \boxplus X_2) \oplus (X_1 \oplus X_2) = \gamma\}$. However, the complexity of computing this distribution can be large. For example, for $n = 32$ bits a straight forward approach would require complexity 2^{64}, an impossible size to implement.

Several previous papers studied related problems. For example, in [11] differential properties of addition, such as $\mathtt{DC}^{+}(\alpha, \beta \to \gamma) := \Pr\{(x \boxplus y) \oplus ((x \oplus \alpha) \boxplus (y \oplus \beta)) = \gamma\}$, were studied in details, including different useful and efficient computational algorithms. There are a few other results where different classes of similar functions (mostly related to differential properties) were achieved, e.g., in [12–14], and others. However, these papers focus only on a small class of functions, which can be regarded as a subclass of the functions studied in this paper, refered to as pseudo-linear functions. Moreover, our main concern is the algorithms on large distribution tables, i.e., to provide a practical tool for cryptanalysis over large distributions (or a large alphabet). When, for example, the probability space is $|\Omega| = 2^{32}$, our algorithms and data structures allow us to store and perform the most common operations over such huge distributions, with a reasonable time on a usual PC.

Consider $X_1, X_2, \ldots, X_k$ to be independent n bit random variables. If they have *arbitrary* distributions, we show how to compute distributions like $\Pr\{X_1 \oplus X_2 \oplus \cdots \oplus X_k\}$ and $\Pr\{X_1 \boxplus X_2 \boxplus \cdots \boxplus X_k\}$ in complexity $O(kn2^n)$. For example, we compute the distribution $\Pr\{(X_1 \boxplus X_2) \oplus (X_1 \oplus X_2) = \gamma\}$ in complexity $2^{37} \cdot c$ for some small c. The presented algorithms makes use of techniques from Fast Fourier Transform and Fast Hadamard Transform. Although some of these techniques were also mentioned in a recent paper [15], we include the full approach for completeness. We show how they can be performed when more complicated data structures are used, introduced due to a high memory complexity.

Next, if $X_1, X_2, \ldots, X_k$ are *uniformly* distributed we demonstrate a large class of functions $F(X_1, X_2, \ldots, X_k)$, for which we can compute the distribution $\Pr\{F(X_1, X_2, \ldots, X_k) = \gamma\}$ efficiently. Here, the algorithms are based on performing a combinatorial count in a bitwise fashion, taking the "carry depth" into account. These results give us efficient methods of calculating distributions of *certain* functions $F(X_1, X_2, \ldots, X_k)$. Fortunately, this includes many functions that appear in linear analysis of ciphers.

As an example, we show an application in linear cryptanalysis of stream ciphers. A typical operation is the approximation obtained when additions modulo

2^n are replaced by bitwise addition. The efficiency of such an approach is given by the bias of a distribution of the above kind. In our example, we give a new improved distinguishing attack on the stream cipher SNOW 2.0.

In Section 2 we define a pseudo-linear class of functions and derive an algorithm to calculate their distributions. In Section 3 we show how a convolution of several distributions can be calculated efficiently. In Section 4 an application example of our approach to attack SNOW 2.0 is given. Finally, we summarize our results and make conclusions.

2 A Pseudo-Linear Function Modulo 2^n and Its Distribution

For notation purposes we denote n-bit variables by a capital letter X, and 1-bit variables by a small letter x. Individual bits of X in a vector form are represented as $X = \overline{x_{n-1} \dots x_1 x_0}$. By $X[a:b]$ we denote an integer number of the form $\overline{x_b \dots x_{a+1} x_a}$. If $Y = \overline{y_{m-1} \dots y_0}$, then $X||Y = \overline{x_{n-1} \dots x_0 y_{m-1} \dots y_0}$ is another integer number (*concatenation*). We use '$\boxplus$' and '$\boxminus$' to denote arithmetical addition and subtraction modulo 2^n, respectively. However, when the inputs to a function $F(\cdot)$ are from the ring $\mathbb{Z}_{2^n}$, we assume '$+$' to be an addition in the ring as well. Matrix multiplication is denoted as '$\times$'. When '$\cdot$' is applied to two vectors, then it denotes element-by-element multiplication of corresponding positions from the vectors.

2.1 A Pseudo-Linear Function Modulo 2^n

Let $\mathcal{X}$ be a set of k uniformly distributed n-bit (nonnegative) integer random variables $\mathcal{X} = \{X_1, \dots, X_k\}$, $X_i \in \mathbb{Z}_{2^n}$. Let $\mathcal{C}$ be a set of n-bit constants $\mathcal{C} = \{C_1, \dots, C_l\}$. Let T_i be some symbol or expression on $\mathcal{X}$ and $\mathcal{C}$. We define *arithmetic*, *Boolean*, and *simple terms* as follows.

Definition 1. *Given $\mathcal{X}$ and $\mathcal{C}$ we say that: (1) $\mathcal{A}$ is an 'arithmetic term', if it has only the arithmetic $+$ operator between the input terms (e.g., $\mathcal{A} = T_1 + T_2 + \dots$); (2) $\mathcal{B}$ is a 'Boolean term' if it contains only bitwise operators such as NOT, OR, AND, XOR, and others (e.g., $\mathcal{B} = (\overline{T_1} \oplus T_2) | T_3 \& \overline{T_4} \dots$); (3) $\mathcal{S}$ is a 'simple term' if it is a symbol either from $\mathcal{X}$ or $\mathcal{C}$ (e.g., $\mathcal{S} = X_i$).* □

Next, we define a *pseudo-linear function modulo* 2^n.

Definition 2. *$F(X_1, \dots, X_k)$ is called a 'pseudo-linear function modulo 2^n' (PLFM) on $\mathcal{X}$ if it can recursively be expressed in arithmetic ($\mathcal{A}$), Boolean ($\mathcal{B}$), and simple ($\mathcal{S}$) terms* [1]. *We also refer the number of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{S}$ terms to be a, b, and s, respectively.* □

[1] Note that a PLFM is a T-function [16], but not vice versa.

Note, if a given function contains a subtraction $\boxminus$, then it can easily be substituted by $\boxplus$ using

$$X \boxminus Y \equiv X \boxplus (\,\texttt{NOT}\,Y) \boxplus 1 \mod 2^n, \tag{1}$$

which is valid in the ring modulo 2^n. Note that the number of $\mathcal{A}$-terms does not grow during the substitution

As an example, let us consider a linear approximation of a modulo sum of the following kind '$X_1 \boxplus X_2 \boxplus X_3 \rightarrow X_1 \oplus X_2 \oplus X_3 \oplus N$', where N is the noise variable introduced due to the approximation. The expression for the noise variable is a PLFM: $N = F(X_1, X_2, X_3) = (X_1 + X_2 + X_3) \oplus X_1 \oplus X_2 \oplus X_3$.

Finding the distribution of such an approximation could be the bottleneck in cryptanalysis work. The trivial algorithm for solving this problem would be as follows.

```
Loop for all $(X_1, X_2, X_3) \in \mathbb{Z}^3_{2^n}$
    $T[(X_1 \boxplus X_2 \boxplus X_3) \oplus X_1 \oplus X_2 \oplus X_3] + +;$
```

After termination of the algorithm we have $\Pr\{N = \gamma\} = T[\gamma]/2^{3n}$. The complexity of this classical solution when the variables are 32-bits integers, is $O(2^{96})$, infeasible for a common PC. Instead, we suggest another principle to solve this problem, as follows.

```
for $\gamma = 0 \ldots 2^n - 1$
    $T[\gamma]$ = some combinatorial function.
```

In the upcoming section we show how this combinatorial function is constructed.

2.2 Algorithm for Calculating the Distribution for a PLFM

The problem we are considering in this subsection is the following. Given a PLFM $F(X_1, X_2, \ldots, X_k)$ on $\mathcal{X}$ and $\mathcal{C}$, we want to calculate the probability $\Pr\{F(X_1, X_2, \ldots, X_k) = \gamma\}$, for a fixed value γ, in an efficient way.

Let some arithmetic term $\mathcal{A}$ have k^+ operators '+', i.e., $\mathcal{A} = T_0 + T_1 + \ldots + T_k$, where T_j are some other terms, possibly $\mathcal{B}$ or $\mathcal{S}$. Then, considering 1-bit inputs, the evaluation of the $\mathcal{A}$ term can, potentially, produce the *local* maximum carry value $\omega_{\max} = \lfloor \frac{k^+ + 1}{2} \rfloor$. This carry value at some bit t can influence on the next bits of the sum at positions $t+1, t+2$, etc. Therefore, the maximum carry value $\sigma_{\max}$ at every bit t of the sum for $\mathcal{A}$ is then derived as the minimum integer solution for the equation $\sigma_{\max} = \lfloor (k^+ + 1 + \sigma_{\max})/2 \rfloor$. Thus, for every arithmetic term $\mathcal{A}_i$ the *maximum local carry value* is

$$\sigma_{i\max} = k_i^+, \tag{2}$$

where k_i^+ is the number of additions in $\mathcal{A}_i$.

For any t-bit truncated input tuple $(X_1, \ldots, X_k)$ to the function $F(\cdot)$ we can define *a tuple of local carry values* for each of the $\mathcal{A}_i$-terms, as follows:

$$\Psi|_t = (\sigma_1, \sigma_2, \ldots, \sigma_a)|_t, \tag{3}$$

where σ_i is the corresponding local carry value for the $\mathcal{A}_i$-term, when the inputs are t-bit truncated, and it can also be expressed as

$$\sigma_i|_t = \left(\sum_{j=0}^{k_i^+} (T_{i,j}(X_1, \ldots, X_k) \mod 2^t) \right) \text{ div } 2^t, \tag{4}$$

when $\mathcal{A}_i = T_{i,0} + \ldots + T_{i,k_i^+}$.

Assume there is an oracle $P_t(\Psi_0, \gamma)$ which can tell us the number of choices of the tuple $(X_1[0 : t-1], \ldots, X_k[0 : t-1])$ out of $2^{t \cdot k}$ possible combinations, such that for each choice the function F produces a required vector of local carry values $\Psi|_t = \Psi_0$, and the condition $F(X_1, \ldots, X_k) = \gamma \mod 2^t$ is satisfied, i.e. $F(X_1, \ldots, X_k)[0 : t-1] = \gamma[0 : t-1]$. The probability we are seeking can now be written as

$$\Pr\{F(X_1, \ldots, X_k) = \gamma\} = \frac{1}{2^{k \cdot n}} \sum_{\Psi} P_n(\Psi, \gamma). \tag{5}$$

It remains to show how to construct the oracles $P_t(\Psi_0, \gamma)$. Assume we know the answer $P_t(\Psi_0, \gamma)$ for every Ψ_0. When $\Psi|_t = \Psi_0$ is fixed, then, by trying all combinations for t^{th} bits of the inputs, i.e., testing each k-bit vector $(X_1[t : t], \ldots, X_k[t : t])$, we can calculate the exact value of $F(X_1, \ldots, X_k)[t : t]$, as well as the exact resulting local carries vector $\Psi|_{t+1}$. Clearly, the oracle $P_{t+1}(\Psi', \gamma)$ makes calls to $P_t(\Psi_0, \gamma)$, for various values of Ψ_0. That relation is linear, and can easily be represented in a matrix form. For this purpose, let us introduce a one-to-one *index mapping function* $\texttt{Index}(\Psi) : (\sigma_1 \times \sigma_2 \times \ldots \times \sigma_a) \to \theta \in [0 \ldots \theta_{\max} - 1]$, as follows.

$$\begin{aligned} &\texttt{Index}(\Psi) = ((\sigma_1 \cdot (\sigma_{2\max} + 1) + \sigma_2) \cdot (\sigma_{3\max} + 1) + \sigma_3) \cdot \ldots \\ &\theta_{\max} = \prod_{j=1}^{a} (\sigma_{j\max} + 1) = \prod_{j=1}^{a} (k_j^+ + 1). \end{aligned} \tag{6}$$

Now, $P_t(\Psi, \gamma)$ for all Ψ can be regarded as a vector $\Big(P_t(\texttt{Index}^{-1}(0), \gamma), \ldots, P_t(\texttt{Index}^{-1}(\theta_{\max} - 1), \gamma)\Big)$, also referred for simplicity as P_t, for all the consecutive valid tuples Ψ. The transformation from P_t to P_{t+1} is a linear function, i.e., it can be written as

$$P_{t+1} = M_{\gamma_t|t} \times P_t, \tag{7}$$

where $M_{\gamma_t|t}$ is some fixed *connection matrix* of size $(\theta_{\max} \times \theta_{\max})$, which, in general, is different for different t's. It depends on the t^{th} bits of the constants involved in $F(\cdot)$, and it also depends on the value of the t^{th} bit γ_t from the given γ, since the oracle $P_{t+1}(\Psi, \gamma)$ must satisfy γ taken modulo 2^{t+1} as well. If the input variables are 0-truncated, then the only one vector $\Psi|_0 = (0, 0, \ldots, 0)$ of local carry values is possible, i.e., $P_0 = (1 \;\; 0 \;\; \ldots \;\; 0)$. Therefore, we assign the oracle P_0 to be just a zero vector, but $P_0(0, \gamma) = 1$.

In this way, $2n$ such matrices have to be constructed. However, in most cases this number is much less. The algorithm to construct matrices from (7) and then calculate (5) is given as follows.

Theorem 1. *For a given PLFM $F(X_1, \ldots, X_k)$, and a fixed $\gamma \in \mathbb{Z}_{2^n}$, we have:*

$$\Pr\{F(X_1, \ldots, X_k) = \gamma\} = \frac{1}{2^{k \cdot n}}(1\ 1\ \ldots\ 1) \times \left(\prod_{t=n-1}^{0} M_{\gamma_t|t} \right) \times (1\ 0\ \ldots\ 0)^{\mathrm{T}}, \quad (8)$$

where $M_{\gamma_t|t}$ are connection matrices of size $(\theta_{\max} \times \theta_{\max})$, precomputed with the algorithm below.

```
Algorithm: Construction of $2n$ matrices $M_{\gamma_t|t}$.

1. Input:
      $F(X_1, \ldots, X_k)$ – a PLFM with $a$ arithmetical terms $\mathcal{A}_i$, each having
      $k_i^+$ operators '$+$', correspondingly;
2. Data structures:
      $\theta_{\max} = \prod_{i=1}^{a}(k_i^+ + 1)$;
      $M_{\{0,1\}|t=[0\ldots n-1]}[\theta_{\max}][\theta_{\max}]$ – $2n$ square matrices of size $(\theta_{\max} \times \theta_{\max})$,
      initialised with zeros;
3. Precomputation algorithm:
    for $t = 0 \ldots n-1$
      Temporary set the constants from $\mathcal{C}$ to be just $t^{\text{th}}$ bit of the
         original ones, i.e., set $(C_1, \ldots, C_l) = (C_1[t:t], \ldots, C_l[t:t])$
      for $(X_1, \ldots, X_k) \in \{0,1\}^k$   – (all combinations for the $t^{\text{th}}$ bits of $X$'s)
        for $\theta = 0 \ldots \theta_{\max} - 1$        – (all combinations for $\Psi$)
          $(\sigma_1, \ldots, \sigma_a) = \mathtt{Index}^{-1}(\theta)$
       [z] Evaluate all $\mu_i = \sigma_i + \mathcal{A}_i(X_1, \ldots, X_n)$, but in $\mathcal{A}_i$ substitute
             all sub terms $\mathcal{A}_j$ with the values $(\mu_j \mod 2)$, correspondingly
          $\theta' = \mathtt{Index}(\mu_1 \text{ div } 2, \ldots, \mu_a \text{ div } 2)$     – (a new resulting $\Psi'$)
          Evaluate the function $f = F(\cdot) \mod 2$, but substitute
             all terms $\mathcal{A}_j$ with the values $\mu_j$, correspondingly
          $M_{f|t}[\theta'][\theta] := M_{f|t}[\theta'][\theta] + 1$

- Time Complexity: $O(n \cdot \theta_{\max} \cdot 2^k)$
- Memory Complexity: $O(2n \cdot \theta_{\max}^2)$
```

[z] Variables μ_i, which correspond to the terms $\mathcal{A}_i$, should be calculated recursively. The deepest $\mathcal{A}$ term should be calculated first, and so on.

□

Below we give an example that demonstrates all the steps of the algorithm.

Example 1. Let $k = 3$, $n = 5$. Assume that our goal is to calculate the probability $\Pr\{F(X_1, X_2, X_3) = 10110_2\}$, where:

$$F(X_1, X_2, X_3) = (X_1 \boxplus (X_2 \oplus (X_1 \boxminus X_2 \boxplus 25)))) \oplus (X_1 \ \mathtt{AND}\ X_3). \quad (9)$$

The first step is to cancel the operator ⊟ by (1), and by rewriting the expression we get

$$F(X_1, X_2, X_3) = \underbrace{(\overbrace{X_1 + (\overbrace{X_2 \oplus (\underbrace{X_1 + (\underbrace{\texttt{NOT}\ X_2}_{\mathcal{B}_1}) + 26}_{\mathcal{A}_1})}^{\mathcal{B}_2})}^{\mathcal{A}_2}) \oplus (X_1\ \texttt{AND}\ X_3)}_{\mathcal{B}_3}\,.$$

```
1.$\theta_{\max} = (k_1^+ + 1)(k_2^+ + 1) = 3 \cdot 2 = 6$;
2.for $t = 0 \ldots 4$
3.  $C = 26[t : t]$
4.  for $(X_1, X_2, X_3) \in \{0,1\}^3$
5.    for $(\sigma_1, \sigma_2) = (0 \ldots 2, 0 \ldots 1)$
6.      $\mu_1 = \sigma_1 + X_1 + (\texttt{ NOT } X_2) + C$
7.      $\mu_2 = \sigma_2 + X_1 + (X_2 \oplus \mu_1 \mod 2)$
8.      $f = (\mu_2 \oplus (X_1 \texttt{ AND } X_3)) \mod 2$
9.      $M_{f|t}[(\mu_1 \text{ div } 2) \cdot 2 + (\mu_2 \text{ div } 2)]$
            $[\sigma_1 \cdot 2 + \sigma_2]$++.
Applying Theorem 1 to construct $2n$
matrices.
```

The function $F(\cdot)$ is a PLFM, since it can be expressed in $\mathcal{A}$ and $\mathcal{B}$ terms, marked above (the $\mathcal{S}$ terms are simply elements from the set $\{X_1, X_2, X_3, 26\}$). I.e.,

$$\mathcal{B}_1(\mathcal{X}, \mathcal{C}) = \texttt{ NOT } X_2$$

$$\mathcal{A}_1(\mathcal{X}, \mathcal{C}) = \underbrace{X_1 + \mathcal{B}_1(\mathcal{X}, \mathcal{C}) + 26}_{k_1^+=2}$$

$$\mathcal{B}_2(\mathcal{X}, \mathcal{C}) = X_2 \oplus \mathcal{A}_1(\mathcal{X}, \mathcal{C})$$

$$\mathcal{A}_2(\mathcal{X}, \mathcal{C}) = \underbrace{X_1 + \mathcal{B}_2(\mathcal{X}, \mathcal{C})}_{k_2^+=1}$$

$$\mathcal{B}_3(\mathcal{X}, \mathcal{C}) = \mathcal{A}_2(\mathcal{X}, \mathcal{C}) \oplus (X_1 \texttt{ AND } X_3), \quad \text{where } F(X_1, X_2, X_3) = \mathcal{B}_3(\mathcal{X}, \mathcal{C}).$$

After all computations we receive the following matrices

$$M_{\gamma_0=0|t=0} = \begin{pmatrix} 1&0&2&0&0&0 \\ 0&5&0&0&0&0 \\ 1&0&2&0&1&0 \\ 0&1&2&2&0&5 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{pmatrix} \quad M_{\gamma_0=1|t=0} = \begin{pmatrix} 5&0&0&2&0&0 \\ 0&1&0&0&0&0 \\ 1&0&0&2&5&0 \\ 0&1&2&2&0&1 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \end{pmatrix} \quad M_{\gamma_1=0|t=1} = \begin{pmatrix} 2&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 2&0&1&0&2&0 \\ 2&2&0&5&0&0 \\ 0&0&1&0&2&0 \\ 0&0&0&1&2&2 \end{pmatrix} \quad M_{\gamma_1=1|t=1} = \begin{pmatrix} 0&2&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&2&5&0&0&2 \\ 2&2&0&1&0&0 \\ 0&0&1&0&0&2 \\ 0&0&0&1&2&2 \end{pmatrix}.$$

No need to construct the matrices for $t = 2, 3, 4$, because they will repeat as $M_{*|t=2} = M_{*|t=0}$ and $M_{*|t=4} = M_{*|t=3} = M_{*|t=1}$. This happens since there are only two different combinations for any t^{th} "bit slice" of constants from the set $\mathcal{C} = \{26\}$. In particular, for every bit t we have $26[t : t] = 0$ or 1 in step 3 in the figure above. Finally, from (8) we calculate

$$\Pr\{F(X_1, X_2, X_3) = 10110_2\} = \frac{1}{2^{15}}(1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1) \times M_{1|4} \times M_{0|3} \times M_{1|2} \times$$

$$\times M_{1|1} \times M_{0|0} \times (1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0)^{\mathrm{T}} = \frac{1}{2^{15}} \cdot 404 \approx 0.0123291015625.$$

One can check this probability by the classical solution, trying all possible values for $(X_1, X_2, X_3) \in \mathbb{Z}_{2^5}^3$ and calculating the function $F(\cdot)$ directly from (9).

Preparing the matrices requires $2 \cdot 2^3 \cdot 6 = 96$ steps (2 values for t, 8 combinations for (X_1, X_2, X_3), and the number of different local carries is $\theta_{\max} = 6$); each step requires one function evaluation. To calculate one probability we need to make 5 multiplications of a matrix and a vector, which takes $5 \cdot 6^2$ operations, plus one scalar product of two vectors at the end, i.e., in total 186 operations. Calculating the complete distribution for all possible γ's takes $2^5 \cdot 186 = 5952$ operations in total. Note that the classical way requires $2^{3 \cdot 5} = 32768$ steps with the function evaluation each step. □

The second example presented in Appendix A is taken from the real cryptanalysis. In that example we, additionally, demonstrate a new trick and show how time complexity can be reduced even more than in Theorem 1. With a precomputation, which usually takes a negligible time, the construction of the complete distribution can have a very small time complexity $O(\theta_{\max} \cdot 2^n)$. That example also shows the advantage of using proposed technique as the computation complexity 2^{96} from the classical solution is reduced down to $2^{32.585}$.

3 Distributions of Functions With Arbitrarily Distributed Inputs

The previous section assumed $X_1, X_2, \ldots$ to be uniformly distributed, allowing a combinatorial approach. In this section we consider $X_1, X_2, \ldots$ *independent* but with *arbitrary* distributions. Despite the ideas described in this section were partly mentioned in [15], we include them for completeness.

Let us have a probability space Ω of size $q = |\Omega| = 2^n$ and two distributions D_X and D_Y over Ω for two random variables X and Y, respectively. Given the distributions D_X and D_Y we consider two major types of convolution, defined as

$$\begin{aligned} & D_Z = D_X * D_Y :\Rightarrow \\ & \Pr\{Z = Z_0\} = \sum_{\substack{\forall X_0, Y_0 \in \Omega : \\ X_0 * Y_0 = Z_0}} \Pr\{X = X_0\} \cdot \Pr\{Y = Y_0\}, \quad \forall Z_0 \in \mathbb{Z}_{2^n}, \end{aligned} \tag{10}$$

where $*$ is either $\boxplus$ or $\oplus$.

In both cases the time complexity to calculate the resulting distribution D_Z is $O(q^2)$, i.e., quadratic. Due to such a high complexity, many attacks in cryptanalysis deal with at most 16-18-bit distributions only. Nowadays, when design of ciphers is often 32-bit oriented, it would be a challenging and useful task to perform a convolution of two 32-bit distributions, i.e., calculating $\Pr\{X + Y = \gamma\}$ for all γ when X and Y have some arbitrary distributions.

For notation purposes the distribution D_X will also be represented as a vector of size 2^n of probabilities as $[D_X] = \{p_X(0), p_X(1), \ldots, p_X(2^n - 1)\}$, where $p_X(X_0) = \Pr\{X = X_0\}$.

Convolution over $\boxplus$. If $[D_X]$ and $[D_Y]$ are represented as two polynomials with coefficients from these two vectors, then the resulting vector $[D_Z]$ has coefficients of the product of the polynomials $[D_X]$ and $[D_Y]$. Fast multiplication of two polynomials can be done via *Fast Fourier Transform (FFT)* [17], the complexity of which is $O(q \log q)$ [2]. The convolution over $\boxplus$ can now easily be calculated as

$$[D_Z] = [D_X \boxplus D_Y] = \mathrm{FFT}_n^{-1}(\mathrm{FFT}_n([D_X]) \cdot \mathrm{FFT}_n([D_Y])). \qquad (11)$$

Convolution over $\oplus$. A similar idea can be applied to this type of convolution. Instead, we use *Fast Hadamard Transform (FHT)* [17].

FHT is a linear transformation of a vector of size 2^n. This transformation can also be done by a matrix multiplication $H_n \times [V]$, where H_n is a well-known Hadamard matrix. FHT, however, performs this matrix multiplication for time $O(q \log q = n \cdot 2^n)$, the same as FFT. In practice, however, FHT is much faster than FFT, since it does not need to work with complex and float numbers. Therefore, approximations of kind $\boxplus \Rightarrow \oplus$ are more preferable, than otherwise. Additionally, the implementation of FHT is extremely simple and small in C/C++, and we present it in Appendix C.

Since FHT_n^{-1} differs from FHT_n by only the coefficient 2^{-n}, then the convolution over $\oplus$ via FHT is computed as

$$[D_Z] = [D_X \oplus D_Y] = \frac{1}{2^n} \cdot \mathrm{FHT}_n(\mathrm{FHT}_n([D_X]) \cdot \mathrm{FHT}_n([D_Y])). \qquad (12)$$

Finally, we point out that the convolution of a linear composition of k independent terms is derived as

$$D_{(Z = C_1 X_1 \oplus C_2 X_2 \oplus \ldots \oplus C_k X_k)} = \frac{1}{2^n} \cdot \mathrm{FHT}_n \left(\mathrm{FHT}_n([D_{C_1 X_1}]) \cdot \ldots \cdot \mathrm{FHT}_n([D_{C_k X_k}]) \right),$$

where C_i are some constants. In practice, this also means that if these distribution tables for $X_1, \ldots, X_k$ are stored with precisions $\xi_1, \ldots, \xi_k$ bits after point, respectively, then for probabilities of Z the precision of only $\xi = n + \sum_{j=1}^{k} \xi_j$ bits after point should be considered (or reserved) before the FHT procedure.

In sections above several algorithms have been derived with good time complexities, which, in most cases, allow us to operate on large distributions. However, memory complexity problems become to be the main concern for implementation aspects. We have algorithms that operate with 32-bit distributions, but how to manage the memory? We present a possible solution in Appendix B, suggest our data structures for large distributions and show how typical operations can be mounted.

4 Application: 32-bit Cryptanalysis of SNOW 2.0

A stream cipher is a cryptographic primitive used to ensure privacy on a communication channel. The SNOW family is a typical example of word-oriented

[2] The resulting polynomial $[D_X] \cdot [D_Y]$ is of degree $2q$, but its powers have to be taken modulo q. It means that the second half just need to be added to the first half of $2n$ coefficients, in order to receive $[D_Z]$. However, this is done automatically when FFT of size q is applied to $[D_X]$ and $[D_Y]$ directly.

KSGs based on a linear feedback shift register (LFSR). SNOW 2.0 is an improved version of SNOW 1.0 aimed to be more secure and still more efficient in performance. The most powerful attack on SNOW 2.0 was presented by Watanabe, Biryukov and De Cannie're [18] in 2003. It is a linear distinguishing attack similar to the general framework presented in [19, 20] and it requires a received keystream sequence of length 2^{225} bits and has a similar time complexity.

In this section we propose an improved attack on SNOW 2.0. Whereas the attack in [18] uses a binary linear approximation approach, the new attack is based on approximations of words, i.e., 32-bit vectors. This technique is more powerful and we get a reduction of the required keystream length to 2^{202}. To make the calculation of 32-bit distributions possible we use our algorithms and data structures from Appendix B.

4.1 A Short Description of SNOW 2.0

The structure of SNOW 2.0 is shown in Figure 1. It has 128- or 256-bit secret key and a 128-bit initial vector. It is based on LFSR over $\mathbb{F}_{2^{32}}[x]$ and the feedback polynomial is given by

$$\pi(x) = \alpha x^{16} + x^{14} + \alpha^{-1}x^5 + 1, \tag{13}$$

where α is a root of the polynomial

$$y^4 + \beta^{23}y^3 + \beta^{245}y^2 + \beta^{48}y + \beta^{239} \in \mathbb{F}_{2^8}[y], \tag{14}$$

and β is a root of

$$z^8 + z^7 + z^5 + z^3 + 1 \in \mathbb{F}_2[z]. \tag{15}$$

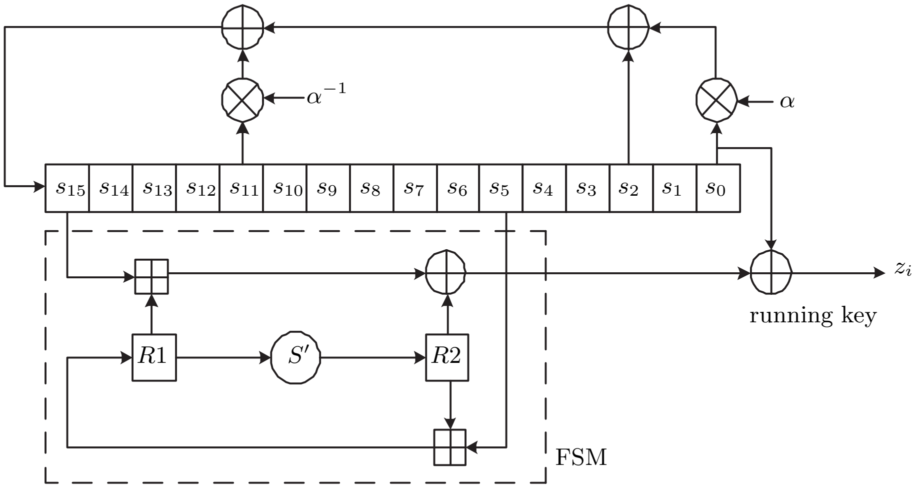

Fig. 1. The structure of SNOW 2.0

The state of the LFSR is denoted by $(s_{t+15}, s_{t+14}, \ldots, s_t)$. Each s_{t+i} is an element of the field $\mathbb{F}_{2^{32}}$. The Finite State Machine (FSM) has two 32-bit registers, $R1$ and $R2$. The output of the FSM F_i is given by

$$F_i = (s_{t+15} \boxplus R1_t) \oplus R2_t, \quad t \geq 0, \tag{16}$$

and the keystream z_t is given by

$$z_t = F_t \oplus s_t, \quad t \geq 1. \tag{17}$$

Two registers $R1$ and $R2$ are updated as follows,

$$\begin{aligned} R1_{t+1} &= s_{t+5} \boxplus R2_t, \\ R2_{t+1} &= S'(R1_t). \end{aligned} \tag{18}$$

where $S'(W)$ is a one-to-one mapping transformation $S' : \mathbb{F}_{2^{32}} \rightarrow \mathbb{F}_{2^{32}}$. If a 32-bit integer W is represented as a vector of 4 8-bit bytes $W = (\, w_0 \; w_1 \; w_2 \; w_3 \,)^{\mathrm{T}}$, then

$$S'(W) = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \cdot \begin{pmatrix} S_R[w_0] \\ S_R[w_1] \\ S_R[w_2] \\ S_R[w_3] \end{pmatrix}, \tag{19}$$

where S_R is the Rijndael 8-to-8-bit S-box, and the linear transformation (matrix multiplication) is done in the field $\mathbb{F}_{2^8}$ with generating polynomial

$$g(x) = x^8 + x^4 + x^3 + x + 1 \in \mathbb{F}_2[x]. \tag{20}$$

4.2 Basic Idea Behind the New Attack

The basic idea behind the new attack is to find such a linear combination of the output words z_i that is equal to 0 if the system is linear, or producing some biased noise if the system is approximated by a linear function. From the other hand, the linear combination representing the noise should be unbiased if the given sequence z_i is truly random.

Consider the feedback polynomial of the LFSR given in equation (13), i.e., $\pi(x) = \alpha x^{16} + x^{14} + \alpha^{-1} x^5 + 1$. A similar relation holds for the LFSR's output s_t at any time t, i.e.,

$$s_{t+16} \oplus \alpha^{-1} s_{t+11} \oplus s_{t+2} \oplus \alpha s_t = 0, \quad t \geq 1. \tag{21}$$

Next we make an approximation of the FSM to make it look linear. For any time $t \geq 1$ two output words z_t and z_{t+1} can be expressed as

$$\begin{cases} z_t = s_t \oplus (R1 \boxplus s_{t+15}) \oplus R2 \\ z_{t+1} = s_{t+1} \oplus S'(R1) \oplus (R2 \boxplus s_{t+5} \boxplus s_{t+16}). \end{cases} \tag{22}$$

Let us substitute $\boxplus \rightarrow \oplus$ and change $S'(R) \rightarrow R$. Then the sum $z_t \oplus z_{t+1}$ is expressed as

$$\begin{aligned} z_t \oplus z_{t+1} &= s_t \oplus (R1 \oplus s_{t+15} \oplus N_{c2}(R1, s_{t+15})) \oplus R2 \\ &\quad \oplus s_{t+1} \oplus (R1 \oplus N_S(S'(R1), R1)) \\ &\quad \oplus (R2 \oplus s_{t+5} \oplus s_{t+16} \oplus N_{c3}(R2, s_{t+5}, s_{t+16})) \\ &= s_t \oplus s_{t+1} \oplus s_{t+5} \oplus s_{t+15} \oplus s_{t+16} \oplus N_0(t), \end{aligned} \tag{23}$$

where $N_0(t)$ is a variable representing the error introduced by the linear approximation in time t,

$$N_0(t) = N_{c2}(R1, s_{t+15}) \oplus N_S(S'(R1), R1) \oplus N_{c3}(R2, s_{t+5}, s_{t+16}). \tag{24}$$

Here $N_{c2}(R1, s_{t+15})$ is a noise random variable introduced by the approximation of the modulo sum of two variables of the following kind "$R1 \boxplus s_{t+15} \to R1 \oplus s_{t+15} \oplus N_{c2}$". The variable $N_{c3}(R2, s_{t+5}, s_{t+16})$ is a similar approximation noise, but for the modulo sum of three variables. Finally, $N_S(S'(R1), R1))$ is the noise variable from the approximation "$S'(R1) \to R1 \oplus N_S$". Let us derive a linear relation, based on (21).

$$\begin{aligned} 0 &\overset{\text{Eq(21)}}{=} (s_{t+16} \oplus \alpha^{-1} s_{t+11} \oplus s_{t+2} \oplus \alpha s_t) \oplus (s_{t+17} \oplus \alpha^{-1} s_{t+12} \oplus_{t+3} \oplus \alpha s_{t+1}) \\ &\quad \oplus (s_{t+21} \oplus \alpha^{-1} s_{t+16} \oplus s_{t+7} \oplus \alpha s_{t+5}) \oplus (s_{t+31} \oplus \alpha^{-1} s_{t+26} \oplus s_{t+17} \\ &\quad \oplus \alpha s_{t+15}) \oplus (s_{t+32} \oplus \alpha^{-1} s_{t+27} \oplus s_{t+18} \oplus \alpha s_{t+16}) \\ &= (s_{t+16} \oplus s_{t+17} \oplus s_{t+21} \oplus s_{t+31} \oplus s_{t+32}) \oplus \alpha^{-1} \cdot (s_{t+11} \oplus s_{t+12} \\ &\quad \oplus s_{t+16} \oplus s_{t+26} \oplus s_{t+27}) \oplus (s_{t+2} \oplus s_{t+3} \oplus s_{t+7} \oplus s_{t+17} \oplus s_{t+18}) \\ &\quad \oplus \alpha \cdot (s_t \oplus s_{t+1} \oplus s_{t+5} \oplus s_{t+15} \oplus s_{t+16}) \\ &\overset{\text{Eq(22)}}{=} (z_{t+2} \oplus z_{t+3} \oplus z_{t+16} \oplus z_{t+17}) \oplus \alpha^{-1} \cdot (z_{t+11} \oplus z_{t+12}) \\ &\quad \oplus \alpha \cdot (z_t \oplus z_{t+1}) \oplus (N_0(t+2) \oplus N_0(t+16)) \oplus \alpha^{-1} \cdot N_0(t+11) \\ &\quad \oplus \alpha \cdot N_0(t) \quad = \quad \mathbf{Z}(t) \oplus \mathbf{N}(t), \end{aligned} \tag{25}$$

where $\mathbf{N}(t)$ is the 32-bit total sum of noise variables introduced by several approximations, expressed as $\mathbf{N}(t) = (N_0(t+2) \oplus N_0(t+16)) \oplus \alpha^{-1} \cdot N_0(t+11) \oplus \alpha \cdot N_0(t)$, and $\mathbf{Z}(t)$ is the "known" part calculated from the output sequence at any time t, $\mathbf{Z}(t) = (z_{t+2} \oplus z_{t+3} \oplus z_{t+16} \oplus z_{t+17}) \oplus \alpha^{-1}(z_{t+11} \oplus z_{t+12}) \oplus \alpha(z_t \oplus z_{t+1})$. Obviously, $\mathbf{N}(t) \oplus \mathbf{Z}(t) = 0$.

After all, a linear distinguishing attack can now be performed, if we know the distribution $D_{\mathbf{N}}$ of the 32-bit noise variable $\mathbf{N}$. For a sufficiently large number of received symbols from either the random distribution D_{Random}, or the distribution of the noise $D_{\mathbf{N}}$, one can construct the *type* (or *empirical distribution*) D_{Type}. We then make a decision whether the stream comes from a truly random generator or from the cipher, according to the distances from D_{Type} to $D_{\mathbf{N}}$ and D_{Random}. Note, the 32-bit noise distribution definitely contains the best binary approximation found in [18], but, clearly, it also contains some additional information, which makes the bias of the noise larger.

We will explain this procedure more in detail in the full version of the paper, but since this is a standard hypothesis testing we simply refer to e.g., [9, 21].

4.3 Computational Aspects

To calculate the bias of the 32-bit noise variable $\mathbf{N}$, its distribution table has to be constructed. It can be calculates via the distribution of N_0, expressed in (24) [3]. To construct the distributions of N_{c2} and N_{c3} we use Theorem 1 (PLFM construction). The expression for N_S is a function on one variable, i.e., it takes no more than $O(2^{32})$ operations to build the distribution D_{N_S}. Next, the distribution of N_0 is calculated via FHT with the algorithm from Section 3 (convolution over $\oplus$) and Appendix B (FHT for large distributions). Afterwards, the distribution of $\alpha \cdot N_0$ and $\alpha^{-1} \cdot N_0$ was computed using algorithms described in Appendix B (function evaluation). Finally, we again use FHT to calculate the distribution of the total noise variable $D_{\mathbf{N}}$, and then calculate the bias $\epsilon = |D_{\mathbf{N}} - D_{\text{Random}}|$.

All these operations took us less than 2 weeks on a usual Pentium IV 3.4GHz, 2Gb of memory and 256Gb of HDD.

4.4 Simulation Results and Discussions

At the end of our simulations we received the distance $\epsilon = |D_{\mathbf{N}} - D_{\text{Random}}| \approx 2^{-101}$, which means that SNOW 2.0 can be distinguished from random with the known keystream of size 2^{202}, and with a similar time complexity. The advantage of our attack is presented in the following table.

Attack on SNOW 2.0	bit(s) considered	bias (ϵ)	complexity
Watanabe et. al. [18]	1	$2^{-112.25}$	2^{225}
our attack	32	2^{-101}	2^{202}

For future research work on this topic it is left to note that the expression for the noise variable $\mathbf{N}(t)$ (25) contains two parts: $N_{c3}(R2_t, s_{t+5}, s_{t+16})$ and $N_{c3}(R2_{t+11}, s_{t+16}, s_{t+27})$, which, in our simulations, were considered as independent. However, since they both use the same input s_{t+16}, they are not really independent and, theoretically, the result should be slightly improved if one consider them as dependent.

5 Results and Conclusions

In this paper we have proposed new algorithms for computation of distributions of certain functions where the input variables are from a large alphabet. In the case when the input variables were uniformly distributed, the distribution for

[3] We adopted the data structures from Appendix B for our simulations as follows: we use 2^{10} files, each containing 2^{22} points of a sub distribution. Since the precision of the probabilities have to be at least $2^{-(192\cdot 4+32)}$ (four noises N_0, each containing N_S with precision 2^{-32}, N_{c2} with precision 2^{-64}, and N_{c3} with precision 2^{-96}; plus 32 bits must be reserved for FHT), each cell has to be of size at least 100 bytes. I.e., each sub distribution in the memory takes at least 400Mb. However, this estimate is conservative, and in our simulations we used almost 2Gb of operation memory.

a class of functions called PLFM was shown to be efficiently calculated. The second case considered the same problem but for arbitrary distribution of input variables. Efficient methods of calculating the distribution of sums of variables both in $\mathbb{Z}_{2^n}$ and $\mathbb{F}_{2^n}$ were proposed, based on Fast Fourier Transform and Fast Hadamard Transform, respectively.

The cryptologic applications of the results were demonstrated by extending the linear cryptanalysis of the stream cipher SNOW 2.0 to work over a larger alphabet. We believe that there are many instances of stream ciphers as well as block ciphers, where cryptanalytic results can be improved by considering analysis over a larger alphabet. In all these cases, the algorithms derived in this paper will be essential for calculating the performance of such attacks.

We also believe that the technique considering "local carries" presented in algorithms for PLFMs can easily be transformed for finding *one* or even *all solutions* for equations like $F(X_1, \ldots, X_k) = 0$. Finding solutions for other kinds of equations, including $F(X_1, \ldots, X_k) = \gamma$ and systems of equations, is obviously converted to finding one or all solutions for an equation of the first kind. Consequently, many properties of PLFM functions can be derived, similarly as it was done for smaller classes in, e.g., [11, 12, 14]. More details will be included in the extended version of this paper.

A few open problems can be mentioned. Clearly, we would like to find other classes of functions where we can compute the distribution efficiently. Also, we would like to find further instances of existing ciphers where linear attacks over larger alphabets are applicable.

Acknowledgements

We thank anonymous reviewers for their useful comments that helped us to improve this paper.

References

1. S. Halevi, D. Coppersmith, and C.S. Jutla. Scream: A software-efficient stream cipher. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 195–209. Springer-Verlag, 2002.
2. P. Ekdahl and T. Johansson. SNOW - a new stream cipher. In *Proceedings of First Open NESSIE Workshop*, 2000.
3. P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In K. Nyberg and H. Heys, editors, *Selected Areas in Cryptography—SAC 2002*, volume 2595 of *Lecture Notes in Computer Science*, pages 47–61. Springer-Verlag, 2002.

[0] The work described in this paper has been supported in part by Grant VR 621-2001-2149, and in part by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT.
The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

4. P. Hawkes and G.G. Rose. Primitive specification and supporting documentation for SOBER-t16 submission to NESSIE. In *Proceedings of First Open NESSIE Workshop*, 2000. Available at http://www.cryptonessie.org, Accessed August 18, 2005.
5. P. Hawkes and G.G. Rose. Primitive specification and supporting documentation for SOBER-t32 submission to NESSIE. In *Proceedings of First Open NESSIE Workshop*, 2000. Available at http://www.cryptonessie.org, Accessed August 18, 2005.
6. N. Smart. Cryptography: An Introduction, 2003.
7. M. Briceno, I. Goldberg, and D. Wagner. A pedagogical implementation of A5/1. Available at http://jya.com/a51-pi.htm, Accessed August 18, 2005, 1999.
8. T. Johansson and A. Maximov. A Linear Distinguishing Attack on Scream. In *Information Symposium in Information Theory—ISIT 2003*, page 164. IEEE, 2003.
9. P. Ekdahl and T. Johansson. Distinguishing attacks on SOBER-t16 and SOBER-t32. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 2002.
10. Jovan DJ. Golić and Philip Hawkes. Vectorial approach to fast correlation attacks. *Designs, Codes, and Cryptography*, 35(1):5–19, 2005.
11. Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In *Fast Software Encryption 2001*, pages 336–350. Springer-Verlag, 2002.
12. Helger Lipmaa, Johan Wallén, and Philippe Dumas. On the additive differential probability of exclusive-or. In *Fast Software Encryption 2004*, pages 317–331, 2004.
13. Alexander Maximov. On linear approximation of modulo sum. In *Fast Software Encryption 2004*, pages 483–484, 2004.
14. Helger Lipmaa. On differential properties of pseudo-hadamard transform and related mappings. In *Progress in Cryptology—INDOCRYPT 2002*, pages 48–61. Springer-Verlag, 2002.
15. Jovan Dj. Golic and Guglielmo Morgari. Vectorial fast correlation attacks. *Cryptology ePrint Archive, Report 2004/247*, 2004.
16. A. Klimov and A. Shamir. A new class of invertible mappings. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 470–483. Springer-Verlag, 2003.
17. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
18. D. Watanabe, A. Biryukov, and C. De Canniere. A distinguishing attack of SNOW 2.0 with linear masking method. In *Selected Areas in Cryptography—SAC 2003*, pages 222–233. Springer-Verlag, 2003.
19. D. Coppersmith, S. Halevi, and C.S. Jutla. Cryptanalysis of stream ciphers with linear masking. In M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 515–532. Springer-Verlag, 2002.
20. J.D. Golić. Linear models for keystream generators. *IEEE Transactions on Computers*, 45(1):41–49, January 1996.
21. P. Junod. On the optimality of linear, differential and sequential distinguishers. In *Advances in Cryptology—EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 2003.

Appendix A: Second Example From Real Cryptanalysis

Example 2. Let us have $k = 3$ uniformly distributed independent random variables $X_1, X_2, X_3 \in \mathbb{Z}_{2^{32}}$, i.e., $n = 32$. Assume in some cryptanalysis we perform a linear approximation '$X_1 \boxplus X_2 \boxplus X_3 \rightarrow X_1 \oplus X_2 \oplus X_3 \oplus N$', where N is a noise variable introduced due to the approximation. The task is to find the bias ϵ of the noise variable N.

The expression for N is: $N = \underbrace{(\underbrace{X_1 + X_2 + X_3}_{\mathcal{A}_1}) \oplus X_1 \oplus X_2 \oplus X_3}_{\mathcal{B}_1} \mod 2^{32}$, which is a PLFM with only one $\mathcal{A}$ term. The maximum carry-bit index value is $\theta_{\max} = (k_1^+ + 1) = 3$. Since no constants are involved all matrices $M_{*|t}$ for all t's are the same. Hence, only two matrices $M_{0|0}$ and $M_{1|0}$ have to be constructed, using Theorem 1.

$$M_{\gamma_0=0|t=0} = \begin{pmatrix} 4\,0\,0 \\ 4\,0\,4 \\ 0\,0\,4 \end{pmatrix}, \qquad M_{\gamma_0=1|t=0} = \begin{pmatrix} 0\,1\,0 \\ 0\,6\,0 \\ 0\,1\,0 \end{pmatrix}. \tag{26}$$

The probability $\Pr\{N = \gamma\}$ can now be calculated efficiently. For example, $\Pr\{N = \gamma = \text{0x72A304F8}\} = \frac{1}{2^{3\cdot 32}}(1 \;\; 1 \;\; 1) \times \left(\prod_{t=n-1}^{0} M_{\gamma[t:t]|0}\right) \times (1 \;\; 0 \;\; 0)^{\mathrm{T}} = \frac{1}{2^{96}} \cdot 2187 \cdot 2^{51} \approx 0.266967773/2^{32}$. Note that the probability for an odd γ is 0. To calculate one probability the number of $32 \cdot 3^2 + 3 = 291$ operations is required. Hence, to calculate the complete distribution would take $291 \cdot 2^{32}$ operations.

```
1. Data structures:
     FastT[2][0...2^16 − 1] – two 'fast-tables'
2. Initialisation:
     FastT[0][0] = (1 0 0), FastT[1][0] = (1 1 1)
3. Precomputation of the tables:
   for $t = 0 \dots 15$
   for $x = 1, 0$ (note, the order is backward)
   for $Y = 0 \dots 2^t - 1$
   z    $\mathrm{FastT}[0][x||Y_t] = M_{x|t} \times \mathrm{FastT}[0][Y]$
        $\mathrm{FastT}[1][x||Y_t] = \mathrm{FastT}[1][Y] \times M_{x|n-t-1}$

   Fast-tables precomputation algorithm.
```

[z] Y_t is a t-bit value of Y. I.e., in C/C++ it would look like: $(x||Y_t) \Rightarrow$ `(x<<t)|Y`

However, this time complexity can be reduced significantly with specific data structures use, which we call "fast-tables". Each table is of size 2^{16} entries, which contain 3-dimentional vectors. These tables are precomputed as shown in Figure on the right. This precomputation requires $2^{16} \cdot 2 \cdot 3^2 = 9 \cdot 2^{17}$ operations. The advantage is that any probability can now be derived as just one scalar product

$$\Pr\{N = \gamma\} = \frac{1}{2^{3\cdot 32}} \cdot <\mathrm{FastT}[0][\overline{\gamma_{15}\dots\gamma_0}],\ \mathrm{FastT}[1][\overline{\gamma_{16}\dots\gamma_{31}}]>^{4}, \tag{27}$$

which takes only 3 operations (instead of 291). Finally, the bias ϵ can be derived as follows:

[4] Note, the input for FastT[1][$\cdot$] is bit-reversed.

```
1. $\epsilon = 0.5$                  (the bias for odd values of $\gamma$)
2. for $\gamma = 0 \dots 2^{31} - 1$      (only even $2\gamma$'s are considered)
3.     $\epsilon += |\Pr\{N = 2\gamma\} - 2^{-32}|$
```

The total time for this solution is the following sum: $2 \cdot 2^3 \cdot 3 = 48$ – to compute matrices, $9 \cdot 2^{17}$ – to precompute fast-tables, and $3 \cdot 2^{31}$ – to calculate the bias ϵ. In total $6443630640 \approx 2^{32.585}$ number of operations is required. To calculate the distribution of the noise variable N the same number of operations is needed, whereas the classical solution requires 2^{96} operations. Note, when the question is only to find the bias ϵ for some large distribution with memory limits conditions, the classical solution will fail with respect to the memory limits. □

Appendix B: Data Structures for Large Distributions and Operations

B.1 Data Structure Proposal

Assume we want to operate on a distribution of size 2^n, but, however, the operation memory allows us to work only with a distribution of size at most 2^m, where $m < n$. If this is the case, to be able to work with large distributions of size 2^n we then propose to use *hard disk memory (HDD)*. Let

$$r = n - m,$$

then one need to create 2^r files on HDD, which we denote as $\text{File}^r_{(0\dots 2^r-1)}$, to store one distribution table. The upper parameter r denotes the number of files to be created (2^r), and the index on the bottom is the selector of a particular file. Sometimes we will write also $\text{File}^r_{X:(A)}$ to show that this is the sub distribution file A for the random variable X. Each file stores the corresponding sub distribution of size 2^m. I.e., the probability $\Pr\{X = X_0\}$ can be accessed by

$$\Pr\{X = X_0\} = \text{File}^r_{X:(X_0[m:n-1])}[X_0 \mod 2^m]. \tag{28}$$

Note that the *upper* $r = (n - m)$ bits select the file, and the *lower* m bits are the cell index in the sub distribution.

The operation memory is regarded as *a fast memory*, whereas the HDD memory is regarded as *a very slow memory*. Working with such data structure frequent access (loading and saving) to the files on HDD should be avoided, since these operations are extremely much slower than an access to the memory. I.e., the most operations have to be done in the operation memory domain, and the number of access to the files has to be reduced as much as possible. In the next parts of this Appendix we present efficient solutions to apply common algorithms when operating on large distributions with the proposed data structures.

B.2 A PLFM Distribution Construction

For a given pseudo-linear function $F(\cdot)$ modulo 2^n its distribution can be constructed as follows.

```
1.  for $A = 0 \dots 2^r - 1$
2.      load sub distribution SubDist$[\cdot] \leftarrow \mathrm{File}^{r}_{(A)}$
3.      calculate the vector $v = (1 \quad 1 \quad \dots \quad 1) \times (\prod_{t=r-1}^{0} M_{A[t:t]||t+m})$
4.      for $B = 0 \dots 2^m$
5.          SubDist$[B] = \Pr\{F = \overline{AB}\} = v \times (\prod_{t=m-1}^{0} M_{B[t:t]||t}) \times (1 \;\; 0 \dots \;\; 0)^{\mathrm{T}}$
6.      save sub distribution $\mathrm{File}^{r}_{(A)} \leftarrow \mathrm{SubDist}[\cdot]$
```

This algorithm requires to access each file once. Additionally, the steps 3 and 5 could be done more efficient with precomputed fast-tables (see, e.g. Appendix A).

B.3 A Function $Y = F(X)$ Evaluation Distribution

Let us have a distribution D_X of a random variable X, stored in data structures as suggested before. Let us also have a function defined on one variable $F(X)$. We need to construct the distribution of $Y = F(X)$ in an efficient way. As an example, this function could be a multiplication $\alpha \cdot X$ in some finite field, a permutation of X, a multiplication on a matrix, or some other function on X in general.

One could take the values of X consecutively, and then each time calculate Y. The problem appears when the consecutive values Y should be stored in different files. It could happen that we need to access the Y's files $O(2^n)$ times, which is expensive in time.

We suggest the following algorithm containing three stages. In the first stage the function is evaluated and the resulting Y's are separated into two files (bins), according to the upper bit value. In the second stage we perform *binary sorting* algorithm, each time dividing each bin into two new bins. The third stage accumulates probabilities from the bins and transfer the resulting sub distributions to the data structures of Y (files).

Stage I: Evaluate $Y = F(X)$ and separate into two files (narrowed distribution)

```
create two files (bins) $f_0 = {}^*\mathrm{File}^{1}_{Y:(0)}$ and $f_1 = {}^*\mathrm{File}^{1}_{Y:(1)}$
for all $A = 0 \dots 2^{n-m} - 1$
    load sub distribution SubDist$_X[\cdot] \leftarrow \mathrm{File}^{r}_{X:(A)}$
    for all $B = 0 \dots 2^m - 1$
        Evaluate $Y_0 = F(A||B)$
        Save the pair $f_{Y_0[n-1:n-1]} \leftarrow$ (SubDist$_X[B]$, $Y_0$)
close the files $f_0$ and $f_1$
```

Stage II: Expand the files ${}^*\mathrm{File}^{1}_{Y:(A_1)} \to {}^*\mathrm{File}^{2}_{Y:(A_2)} \to \dots \to {}^*\mathrm{File}^{r}_{Y:(A_r)}$

```
for $k = 1 \dots r - 1$
    for all $A = 0 \dots 2^k - 1$
      open two files $f_0 = {}^*\mathrm{File}^{k+1}_{Y:(A||0)}$ and $f_1 = {}^*\mathrm{File}^{k+1}_{Y:(A||1)}$
      while( not the end of the file ${}^*\mathrm{File}^{k}_{Y:(A)}$ )
        read the pair $(p, Y_0) \leftarrow {}^*\mathrm{File}^{k}_{Y:(A)}$
        save the pair $f_{Y_0[n-k-1:n-k-1]} \leftarrow (p, Y_0)$
      close the files $f_0$ and $f_1$
```

Stage III: Construct $\text{File}^r_{Y:(A)}$ from $^*\text{File}^r_{Y:(A)}$

```
1. for all $A = 0 \dots 2^r$
2.    clear SubDist$_Y[0 \dots 2^m - 1]$
3.    while( not the end of the file $^*\text{File}^r_{Y:(A)}$ )
4.      read the pair $(p, Y_0) \leftarrow {}^*\text{File}^k_{Y:(A)}$
5.      SubDist$_Y[Y_0]$=SubDist$_Y[Y_0]$+$p$
6.    save sub distribution $\text{File}^r_{Y:(A)} \leftarrow \text{SubDist}_Y[\cdot]$
```

The complexity of this algorithm is $O((1+r) \cdot 2^n)$. However, the coefficient r in the complexity can be reduced with a small programming trick. If at the step II.3 we, instead, open 2^d files (in Windows at most 2^9 files can be open at the same time), and perform not a binary sorting but a d-tuple bits sorting at once, then the complexity will be reduced to $O((1 + r/d) \cdot 2^n)$. For example, if the number of files is 2^{16} (r=16), then with $d = 8$ we can compute the distribution of any function $F(X)$ by reading and storing distributions of size 2^n from the files only 3 times (instead of 17).

Note that in the implementation of FFT the first operation is the construction of the distribution $D_{Rev(X)}$ for the *bit reverse* of the random variable X, which is just a sub case of the general problem of this sub section. We simply define the function $Y = F(X)$ such that Y is the bit-reverse of X, and apply the algorithm above. There are other more nice and efficient solutions for this particular problem, but we only mention their existence.

B.4 Convolution over $\oplus$

To perform a convolution over $\oplus$ we need to be able to perform FHT on the proposed data structures. We propose a modified FHT algorithm, where first local FHTs for sub distributions are separately performed, and then evaluate the "convolution" over the files as follows.

```
1. for $A = 0 \dots 2^r - 1$
2.    load sub distribution SubDist$[\cdot] \leftarrow \text{File}^r_{(A)}$
3.    FHT(m, SubDist)
4.    save sub distribution $\text{File}^r_{(A)} \leftarrow$SubDist$[\cdot]$
5. FHT$^*$(r, NULL) -- the same FHT as before but with another
                       butterfly function bfly$^*$(j+k, j+k+(1<<i)).
```

The modified butterfly function bfly* is

```
1. bfly$^*$($A$, $B$)
2.    load   SubDist$_1[\cdot] \leftarrow \text{File}^r_{(A)}$  and  SubDist$_2[\cdot] \leftarrow \text{File}^r_{(B)}$
3.    for $i = 0 \dots 2^m - 1$
4.       bfly(SubDist$_1[i]$, SubDist$_2[i]$)
5.    save   $\text{File}^r_{(A)} \leftarrow$SubDist$_1[\cdot]$  and  $\text{File}^r_{(B)} \leftarrow$SubDist$_2[\cdot]$
```

This algorithm requires to load/save each file $r = n - m$ times. The modified butterfly function `bfly`* can also be implemented memoryless. It can read one value from File$^r_{(A)}$ and one value from File$^r_{(B)}$, perform the usual butterfly operation and save the results back to the files immediately. There are two additional ideas to accelerate the FHT evaluation:

(a) In steps 3 and 4 of the algorithm above only two files are processed. Instead, we could have a larger block of 2^d files opened and processed at the same time. The calculation of the batterfly function on two probabilities `SubDist`$_1[i]$ and `SubDist`$_2[i]$ can be substituted by a 'local' FHT on 2^d inputs, instead. Since the size of each file is 2^m, we need to repeat this procedure 2^m times for each group of 2^d files (inputs are taken in parallel from a group of 2^d files opened at the same time, but the number of such parallel inputs for each group is 2^m). As the result, each file is accessed around $(r + 1)/d$ times;

(b) The computation can also be splittet into 2^c independent processes (2^c computers), and then the results can be merged together afterwards.

B.5 Convolution over $\boxplus$

A convolution over $\boxplus$ on the suggested data structures can be done in a similar way as for $\oplus$. In the first step we perform the *bit reversing* operation on the input distribution, as described in Appendix B.3. Afterwards, we use the same idea as in the previous sub section, based on the *parallel FFT circuit.* The description of the parallel FFT circuit can be found in the book [17].

Appendix C: Efficient FHT Implementation in C/C++

Fast Hadamard Transform (FHT) implementation in C/C++

```
// butterfly operation
template<class T> void inline bfly (T &a, T &b)
{ T tmp; tmp=a; a+=b; b=tmp-b; }

// FHT_n, size of the input distribution is 2^n
template<class T> void FHT(int n, T *Dist)
{ for (int i=0; i<n; ++i)
    for (int j=0; j<(1<<n); j+=1<<(i+1) )
      for (int k=0 ; k<(1<<i); ++k)
         bfly (Dist[j+k], Dist[j+k+(1<<i)]);
}
```