Tatra Mt. Math. Publ. **33** (2006), 99–112

TATRA MOUNTAINS
Mathematical Publications

CHECKING NORMALITY OF BOOLEAN FUNCTIONS

Krzysztof Nowak

ABSTRACT. Besides nonlinearity and correlation immunity, normality is another criterion for cryptographic Boolean functions. A Boolean function is called normal if it is constant on a coset of a vector subspace.

There are no known general mathematical methods for checking normality of Boolean functions, so we have to use computer algorithms.

In this paper we present an algorithm for checking normality. The presented algorithm is an improved version of the first known non-trivial algorithm for checking normality developed by Daum, Dobbertin and Leander. The time complexity of our modification is greatly reduced comparing to original algorithm.

1. Introduction

Boolean functions are widely used in cryptography. They are used in designing S-boxes and stream ciphers. The most important criterions for Boolean functions are nonlinearity, correlation immunity and algebraic degree. Boolean function used in secure cipher must fulfil these criteria. Hans Dobbertin introduced the notion of normality in [5]. Normal function is a function which is constant on a coset of some vector subspace. But cryptographic function should be complex to satisfy Shannon's principle of confusion, so the natural criterion for Boolean function is non-normality.

There are no know mathematical methods for checking normality, so it is difficult to construct non-normal functions. One must use computer algorithms for checking normality.

Magnus Daum, Hans Dobbertin and Gregor Leander have presented the first non-trivial algorithm for checking normality denoted here by DDL algorithm (cf. [3]). The DDL algorithm needed about 50 hours for checking weak-normality of Boolean function in dimension 14. In this paper we present a modified algorithm which can check weak-normality of the same function in about 34 minutes on a similar machine (in our case on Athlon XP 1800+).

2000 Mathematics Subject Classification: 94A60.

Keywords: Boolean functions, bent functions, normality, algorithm.

2. Definitions

Let V_n denote the n-dimensional vector space $\mathbb{F}_2^n$. Cosets of vector spaces will be called *flats*. Every function $f\colon V_n \to \mathbb{F}_2$ will be called a *Boolean function*. Let E be any flat. If $f \upharpoonright E$ is constant, then the flat E is called a *constant flat* for f. If $f \upharpoonright E$ is affine, then the flat E is called an *affine flat* for f.

Claude Carlet generalized the notion of normality in [2].

Definition 2.1. Let $1 \leqslant k \leqslant n$. A Boolean function $f\colon V_n \to \mathbb{F}_2$ is called *k-normal* (resp. *k-weakly-normal*) if there exists a k-dimensional constant (resp. affine) flat for f. If $k = \lceil n/2 \rceil$, then we simply call f a *normal function* (resp. *weakly-normal*).

Normality is not invariant under addition of affine functions, but weak-normality is invariant. So non-weak-normality is more desirable property than non-normality.

3. The algorithm

3.1 The main idea

The presented algorithm generates the list of all constant flats for a given function f and dimension k. Because every k-dimensional affine flat is a union of two parallel constant $(k-1)$-dimensional flats, the algorithm may also be used for checking k-weak-normality. One can simply generate all $(k-1)$-dimensional constant flats and next try to find a pair of parallel constant flats to obtain k-dimensional affine flat for f.

The main idea of the algorithm is the same as in [3]. In the first phase the algorithm generates the set of all constant flats of some small dimension d (for example $d = 1$).

Next, for every $s = d, d+1, \ldots, k-1$ the algorithm joins every pair of parallel s-dimensional constant flats such that the function takes the same value on both of them to obtain a $(s+1)$-dimensional constant flat.

Because every constant flat is a union of two parallel flats, the algorithm eventually generates all k-dimensional constant flats for the given function.

3.2 The unique representation of flats

There are many concerns with the representation of flats. To overcome the possibility of generating one constant flat many times, we need a unique representation of flats. Every flat can be represented by a basis of a vector subspace and a representative vector belonging to it. We start from unique representation of vector spaces.

Let $\overline{v}$ be a vector. Then by $\overline{v}_i$ we denote the ith element of $\overline{v}$ for $i = 1, 2, \dots, n$.

The following definition is the same as in [3].

Definition 3.1. Let $\overline{v} = (\overline{v}_1, \overline{v}_2, \dots, \overline{v}_n) \in V_n$ be any vector. Then by $\Upsilon(\overline{v})$ we denote the smallest index of nonzero element in $\overline{v}$ or $(n+1)$ if $\overline{v} = 0$, i.e.,

$$\Upsilon(\overline{v}) := \max\bigl\{i \in \{1, 2, \dots, n+1\} \colon \overline{v}_j = 0 \text{ for } 1 \leqslant j < i\bigr\}.$$

If U is any set of vectors, then

$$\Upsilon(U) := \bigl\{\Upsilon(\overline{u}) \colon \overline{u} \in U\bigr\}.$$

We will use lexicographical ordering of vectors:

$$(\overline{v}_1, \overline{v}_2, \dots, \overline{v}_n) < (\overline{u}_1, \overline{u}_2, \dots, \overline{u}_n) \iff \bigl(\exists_{1 \leqslant i \leqslant n}(\overline{v}_i < \overline{u}_i \wedge \forall_{1 \leqslant j < i}\, \overline{v}_j = \overline{u}_j)\bigr).$$

We are now ready to define unique representation of a vector space.

Definition 3.2. Let $U = (\overline{u}^{(1)}, \overline{u}^{(2)}, \dots, \overline{u}^{(s)})$ be an s-tuple of vectors. We call this s-tuple a *Gauss-Jordan basis* (GJB) if

- vectors $\overline{u}^{(1)}, \overline{u}^{(2)}, \dots, \overline{u}^{(s)}$ are linearly independent,
- $\overline{u}^{(1)} < \overline{u}^{(2)} < \dots < \overline{u}^{(s)}$ (lexicographical order),
- $\overline{u}^{(i)}_j = 0$ for $j \in \Upsilon(U) \setminus \bigl\{\Upsilon(\overline{u}^{(i)})\bigr\}$, $i = 1, 2, \dots, s$.

The above definition is similar to that in [3], but the order of vectors is ascending. The reverse ordering has two important consequences, as we explain later.

Simply speaking, a GJB is an ascending ordered basis which is completely reduced. One can take any vector basis, reduce it using full Gauss elimination and then sort it to obtain a GJB. So it is easy to check that every vector space has a unique GJB.

Let $(\overline{u}^{(1)}, \overline{u}^{(2)}, \dots, \overline{u}^{(s)})$ be a GJB of a subspace U of V_n. One can check that

$$\Upsilon(U) \setminus \{n+1\} = \Upsilon\bigl(\overline{u}^{(1)}, \overline{u}^{(2)}, \dots, \overline{u}^{(s)}\bigr).$$

Let U be a vector subspace of V_n. Using the Definition 3.1 we can define the complement subspace $\widetilde{U}$ such that V_n is a direct sum $V_n = U \oplus \widetilde{U}$:

$$\widetilde{U} := \bigl\{\overline{u} \in V_n \colon \overline{u}_i = 0 \text{ for all } i \in \Upsilon(U) \setminus \{n+1\}\bigr\}.$$

We are now ready to define a unique canonical representation of flats.

Definition 3.3. Let U be a vector subspace of V_n and let $D = \overline{a}+U$ be any flat, where $\overline{a} \in V_n$. Then the pair $(\overline{d}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)}))$ is called a *canonical representation* of the flat D if the s-tuple $(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)})$ is the GJB basis of the space U and $\overline{d} \in D \cap \widetilde{U}$.

One can easily check that every flat has a unique canonical representation. The canonical representation of flats in our algorithm is the same as in the DDL algorithm except for the order of vectors in GJB.

3. The Filter

Because every s-dimensional flat can be obtained as a union of $(2^s - 1)$ possible pairs of parallel $(s-1)$-dimensional flats, we need some filtering condition for efficiency. This filtering condition will help us to generate exactly one copy of every constant flat for a given function f. Moreover, a canonical representation of a resulting flat will be easily obtained from canonical representations of two parallel source flats.

The following lemma gives us a simpler filtering condition than in the DDL algorithm. This is a consequence of the reverse order of vectors in GJB.

Lemma 3.1. *Let D_1, D_2 be two parallel flats with canonical representations $(\overline{d}^{(1)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}))$ and $(\overline{d}^{(2)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}))$ respectively. Assume that $\overline{d}^{(1)} < \overline{d}^{(2)}$ and $\overline{d}^{(1)} + \overline{d}^{(2)} > \overline{u}^{(s-1)}$.*

Then the pair $(\overline{d}^{(1)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}, \overline{u}^{(s)}))$ is the canonical representation of the flat $D_1 \cup D_2$, where $\overline{u}^{(s)} := \overline{d}^{(1)} + \overline{d}^{(2)}$.

P r o o f . Let $U := \mathrm{lin}(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)})$. ($\mathrm{lin}(X)$ is the subspace spanned by vectors from X.)

The fact that the flat $\overline{d}_1 + U$ is the union $D_1 \cup D_2$ is obvious. We have only to show that the pair $(\overline{d}^{(1)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}, \overline{u}^{(s)}))$ is a canonical representation.

Let $W := \mathrm{lin}(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)})$. The definition of canonical representation of flat implies that $\overline{d}^{(1)}, \overline{d}^{(2)} \in \widetilde{W}$, and of course $\overline{d}^{(1)} + \overline{d}^{(2)} \in \widetilde{W}$. So we have proved that

$$\overline{u}_j^{(s)} = 0\,, \quad \text{for all } j \in \Upsilon\big(\{\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}\}\big)\,.$$

Since $\overline{d}^{(1)} + \overline{d}^{(2)} \neq 0$, we have $\Upsilon(\overline{u}^{(s)}) \notin \Upsilon(W)$. The assumption $\overline{d}^{(1)} + \overline{d}^{(2)} > \overline{u}^{(s-1)}$ implies that $\Upsilon(\overline{u}^{(s)}) < \Upsilon(\overline{u}^{(s-1)})$. From the fact that $\overline{u}^{(1)} < \overline{u}^{(2)} < \cdots < \overline{u}^{(s-1)}$ we obtain

$$\overline{u}^{(i)}_{\Upsilon(\overline{u}^{(s)})} = 0\,, \quad i = 1, 2, \ldots, s-1\,.$$

So we have proved that $\left(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)}\right)$ is the GJB of U. From the fact $\overline{d}^{(1)} < \overline{d}^{(2)}$ we know that $\overline{d}^{(1)}_{\Upsilon\left(\overline{d}^{(1)}+\overline{d}^{(2)}\right)} = 0$, moreover $\overline{d}^{(1)} \in \widetilde{W}$, so

$$\overline{d}^{(1)}_{\Upsilon(\overline{u}^{(i)})} = 0 \qquad \text{for} \quad i = 1, 2, \ldots, s-1\,,$$

which implies that $\overline{d}^{(1)} \in \widetilde{U}$, and proves that $\left(\overline{d}^{(1)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}, \overline{u}^{(s)})\right)$ is the canonical representation of the flat $D_1 \cup D_2$. □

It is easy to check that every flat with canonical representation

$$\left(\overline{d}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)})\right), \quad s \geqslant 1\,,$$

is the union of two flats with canonical representations

$$\left(\overline{d}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)})\right) \quad \text{and} \quad \left(\overline{d} + \overline{u}^{(s)}, (\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)})\right).$$

So the lemma gives us the filter which is helpful in generating every flat only once and gives us the simple construction of canonical representations of flats. It is sufficient to join only such pairs of parallel flats (D_1, D_2) that fulfil the assumptions from the Lemma 3.1.

3.4 The main optimization

There is another more important consequence of using the reverse order in GJBs. Let D be any flat in $V_n = \{0,1\} \times V_{n-1}$. Then one can check that exactly one of the three following conditions must hold:

1. $D \subset \{0\} \times V_{n-1}$,
2. $D \subset \{1\} \times V_{n-1}$,
3. $D = D_0 \cup D_1$, where $D_i \subset \{i\} \times V_{n-1}$, $\dim(D_i) = \dim(D) - 1$ for $i = 0, 1$.

Simply speaking, the condition 1 holds when $\overline{v}_1 = 0$ for all $\overline{v} \in D$, similarly, the condition 2 holds when $\overline{v}_1 = 1$ for all $\overline{v} \in D$. So it is clear that we don't have to generate all $(k-1)$-dimensional constant flats to obtain all k-dimensional constant flats. It is sufficient to generate all $(k-1)$-dimensional constant flats separately in the flats $\{0\} \times V_{n-1}$ and $\{1\} \times V_{n-1}$ and next:

1. join every pair of parallel constant flats contained in the flat $\{0\} \times V_{n-1}$ using filtering condition from the Lemma 3.1,
2. join every pair of parallel constant flats contained in the flat $\{1\} \times V_{n-1}$ using filtering condition from the Lemma 3.1,
3. join every two parallel constant flats, one contained in the flat $\{0\} \times V_{n-1}$ and the other contained in the flat $\{1\} \times V_{n-1}$, without using any filtering condition (the filtering condition from the Lemma 3.1 will be always satisfied in that case).

This idea used recursively greatly reduces the number of generated intermediate dimensional flats. The canonical representation of flats we have defined is well suited to be used with this idea.

Theorem 3.1. *Let $s \geqslant 1$ and let D be an s-dimensional constant flat for the function f. Let $\left(\overline{d}, \left(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s)}\right)\right)$ be the canonical representation of D. Then the pairs $\left(\overline{d}, \left(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}\right)\right)$ and $\left(\overline{d}+\overline{u}^{(s)}, \left(\overline{u}^{(1)}, \overline{u}^{(2)}, \ldots, \overline{u}^{(s-1)}\right)\right)$ are the canonical representations of constant flats D_0 and D_1 respectively such that $D = D_0 \cup D_1$, and, moreover,*

$$D_0 \subseteq \{\overline{d}_1\} \times V_{n-1},$$
$$D_1 \subseteq \left\{\left(\overline{d} + \overline{u}^{(s)}\right)_1\right\} \times V_{n-1}.$$

P r o o f. The proof is straightforward, because

$$\operatorname{lin}\left(u^{(1)}, u^{(2)}, \ldots, u^{(s-1)}\right) \subset \{0\} \times V_{n-1}.$$

□

The following lemma will be used in the algorithm.

Lemma 3.2. *Let U be a subspace of V_n. Let $\overline{u}, \overline{v}, \overline{w}, \overline{t} \in V_n$ be such vectors that:*

1. $\overline{t} \in U \setminus \{0\}$,
2. $\overline{u}, \overline{v}, \overline{w} \in \widetilde{U}$,
3. $\overline{u} \leqslant \overline{v} \leqslant \overline{w}$.

If $\overline{u} + \overline{w} \leqslant \overline{t}$, then $\overline{u} + \overline{v} < \overline{t}$.

P r o o f. Let $p := \Upsilon(\overline{u} + \overline{w})$. From the Definition 3.1 we obtain

$$\overline{u}_i = \overline{w}_i \quad \text{for all } 1 \leqslant i < p.$$

The relation 3 implies that

$$\overline{u}_i = \overline{v}_i \quad \text{for all } 1 \leqslant i < p.$$

Assume that $\overline{u} + \overline{w} \leqslant \overline{t}$, and let $q := \Upsilon(\overline{t})$. Because $\overline{u}, \overline{w} \in \widetilde{U}$, $\overline{t} \in U \setminus \{0\}$ and $\overline{u} + \overline{w} \leqslant \overline{t}$, then the relation $q < p$ is satisfied. Thus for all $1 \leqslant i < q$ we have $\overline{t}_i = (\overline{u} + \overline{v})_i = 0$, moreover, $\overline{t}_q = 1$ and $(\overline{u} + \overline{v})_q = 0$. So we have proved that $\overline{t} > (\overline{u} + \overline{v})$. □

3.5 The details of the algorithm

We are now ready to describe the algorithm in details. Procedures will be presented in **Pascal**-like language. We will use the following operators on machine words:

1. $\oplus$—the XOR operator (for adding two vectors),
2. $a \gg b$—the acyclic right shift of a by b positions.

In addition we will use **foreach** instruction, which iterates contained subinstruction for every element in a given sequence. Sometimes we need to iterate in ascending or descending order, so we will use **foreach asc** and **foreach desc** instructions respectively.

The main data structure of the algorithm is the set of flats of the same basis. Every set of parallel flats is represented by a GJB of some vector space U and set of vectors $p \in \widetilde{U}$ such that the function f is constant on the flat $p+U$. The set of vectors is divided into two main parts depending on the value which f takes on flats represented by vectors. Moreover, these parts are further divided into 2^m parts for some integer m (the details will be explained later).

type
  FlatsOfOneBase = **record**
    base: **list of vector** {*the GJB*};
    splitDim: **integer** {*the splitting dimension*}
    vecs: **array** $[0..1][0..2^{n-\mathrm{splitDim}}-1]$ **of list of vector**;
  end;
  FlatsList: **list of** FlatsOfOneBase;

The algorithm constructs objects of type FlatsOfOneBase in such a manner that every list vecs[i][j] is always lexicographically ordered. This is important, because the algorithm iterates over those lists in ascending and descending order without additional sorting.

Let f be a given function, let k be an integer for which the algorithm is seeking for constant k-dimensional flats. The algorithm starts with generating 1-dimensional constant flats contained in 2^{k-1} flats of the form

$$\{a_1\} \times \{a_2\} \times \cdots \times \{a_{k-1}\} \times V_{n-k+1}.$$

Next, using a recursive procedure, the algorithm combines these flats to obtain higher dimensional constant flats. For $s = 1, 2, \ldots, k-1$ the algorithm combines s-dimensional constant flats contained in the flats of the form

$$\{a_1\} \times \{a_2\} \times \cdots \times \{a_{k-s}\} \times V_{n-k+s},$$

and obtains $(s+1)$-dimensional constant flats contained in the flats of the form

$$\{a_1\} \times \{a_2\} \times \cdots \times \{a_{k-(s+1)}\} \times V_{n-k+(s+1)}.$$

For memory efficiency the algorithm does not generate all s-dimensional flats before generating $(s+1)$-dimensional flats. The main procedure of the algorithm is an iteration over all nonzero vectors in $\{0\}^{k-1} \times V_{n-(k-1)}$. In each iteration the algorithm constructs all constant flats for the function f such that the first vector of GJB is fixed. It can be done in this way because the basis of a flat of higher dimension is always obtained by appending some vector at the end of a GJB of two smaller dimensional flats.

The main procedure is as follows:

```
{Generates all k-dimensional constant
flats for a Boolean function f: V_n → F_2}
procedure GenerateConstFlats
var
  splitDim: integer;
  fbv: vector;
  flats: FlatsOfOneBase;
  fList: FlatsList;
begin
  splitDim := n − k + 1;

  foreach fbv in ({0}^{k−1} × V_{n−k+1}) \ {0} do
  begin
    {initialize flats with empty basis and empty lists
    of vectors using supplied parameter splitDim}
    init(flats, splitDim);

    {create one vector basis}
    flats.base.append(fbv);

    {generate all constant flats with basis (fbv)
    contained in splitDim-dimensional parallel flats}
    GenerateDim1(flats);

    {create the list containing one element}
    fList := ∅; fList.append(flats);

    {call a recursive procedure}
    CombineList(fList, splitDim + 1);
  end;
end;
```

The procedure GenerateDim1 is simply the brute force search. The procedure CombineList iterates over all elements on a given list and combines constant flats.

```
procedure CombineList(fList: FlatsList; splitDim: integer);
var
  flats: FlatsOfOneBase;
  newFlatsList: FlatsList;
begin
```

```
  if splitDim = n then
     {print all found constant flats from fList}
  else
     foreach flats in fList do
     begin
        newFlatsList := Combine(flats);
        {recursive call}
        CombineList(newFlatsList, splitDim + 1);
     end;
end;
```

Much more interesting is the procedure Combine. The algorithm uses the global variable flatsIndex, which is a dictionary of pointers to objects of type FlatsOfOneBase indexed by vectors. That dictionary can be implemented in such a way that it can store and retrieve pointers in constant time and it can be cleared in constant amortized time.

```
function Combine(flats: FlatsOfOneBase): FlatsList
var
  value, pIdx, seqSize: integer;
  result: FlatsList;
begin
  clear (flatsIndex); {clear the index}
  seqSize := 2^(n−flats.splitDim);

  for value := 0 to 1 do
  begin
     {combine constant flats for given value}
     for pIdx := 0 to seqSize − 2 step 2 do
     begin
 {Warning! flats.vecs[value][pIdx] must be processed
  before flats.vecs[value][pIdx + 1] for correct
  ascending order of vectors}
        CombineIn1Flat(result, flats, pIdx, value);
        CombineIn1Flat(result, flats, pIdx + 1, value);

 {combine flats contained in two parallel flats}
        CombineIn2Flats(result, flats, pIdx, value);
     end
  end

  return result;
end;
```

The procedure CombineIn1Flat joins pairs of two parallel flats using filtering condition from the Lemma 3.1. Let us note that the filtering condition from the lemma is reduced to one boolean expression (newVec <= srcFlats.base.**last**), because there are two iterations, the outer in ascending and the inner in descending order.

```
procedure CombineIn1Flat(var result: FlatsList; srcFlats:
  FlatsOfOneBase; pIdx, value: integer)
var
   newVec: vector;
   newFlats: ^FlatsOfOneBase; {a pointer}
   newPIdx: integer;
begin
   foreach asc point0 in srcFlats.vecs[value][pIdx] do
   begin
     foreach desc point1 in srcFlats.vecs[value][pIdx] do
     begin
        newVec := point0 ⊕ point1;
{compare newVec with the last base vector,
  if true, we leave the inner loop (cf. Lemma 3.2),
  if false, then point0 <= point1}
        if newVec <= srcFlats.base.last then
           break; {break the inner loop}
{check if the new flat has a basis obtained earlier}
        newFlats := flatsIndex[newVec];
        if nil = newFlats then
        begin
{create new FlatsOfOneBase}
           newFlats := new(FlatsOfOneBase);
           init(newFlats, srcFlats.splitDim + 1);
           newFlats^base := srcFlats.base; {copy the basis}
{append newVec to the basis}
           newFlats^base.append(newVec);
{append newFlats to the resulting list}
           result.append(newFlats);
{update the index}
           flatsIndex[newVec] := newFlats;
        end;
```

```
{calculate the position of the list for point0}
         newPIdx := point0 >> (newFlats^plitDim);
         newFlats^vecs[value][newPIdx].append(point0);
      end;
   end;
end;
```

The last procedure combines pairs of parallel flats contained in different flats of the form

$$\{a_1\} \times \{a_2\} \times \cdots \times \{a_{s-1}\} \times V_{n-s+1} \,.$$

```
procedure CombineIn2Flats(var result: FlatsList;
   srcFlats: FlatsOfOneBase; pIdx, value: integer);
var
   newVec: vector;
   newFlats: ^FlatsOfOneBase;
   newPIdx: integer;
begin
   foreach asc point0 in srcFlats.vecs[value][pIdx] do
      foreach point1 in srcFlats.vecs[value][pIdx + 1] do
      begin
         newVec := point0 ⊕ point1;
{check if the new flat has basis obtained earlier}
         newFlats := flatsIndex[newVec];
         if nil = newFlats then
         begin
{create new FlatsOfOneBase}
            newFlats := new(FlatsOfOneBase);
            init(newFlats, srcFlats.splitDim + 1);
            newFlats^base := srcFlats.base; {copy the basis}
{append newVec to the basis}
            newFlats^base.append(newVec);
{append newFlats to the resulting list}
            result.append(newFlats);
{update the index}
            flatsIndex [newVec] := newFlats;
         end;
         newPIdx := point0 >> (newFlats^splitDim);
```

```
      newFlats^vecs[value][newPIdx].append(point0);
    end;
  end;
end;
```

One can use the algorithm to check $(k+1)$-weak-normality of the function f. It is sufficient to join every pair of parallel k-dimensional constant flats for f in the procedure CombineList when splitDim $= n$.

3.6 Complexity of the algorithm

The evaluation of complexity of our modification is similar to that presented in [3]. The main difference is the number of intermediate dimensional constant flats generated by our algorithm.

The probability that an m-dimensional flat is constant for random Boolean function is equal:

$$\frac{1}{2^{2^m-1}}.$$

So the expected number of all m-dimensional constant flats for random function is equal (cf. [3]):

$$\frac{1}{2^{2^m-1}} \cdot 2^{n-m} \cdot \frac{\prod_{i=0}^{m-1} (2^n - 2^i)}{\prod_{i=0}^{m-1} (2^m - 2^i)}. \tag{1}$$

The above number is the expected number of m-dimensional constant flats generated by the DDL algorithm for $m = d, d+1, d+2, \ldots, k$, where d is the starting dimension, and k is the dimension of constant flats for which the algorithm is seeking. The modified algorithm generates on average

$$\frac{1}{2^{2^m-1}} 2^{k-m} \left(2^{n-(k-m)-m} \frac{\prod_{i=0}^{m-1} (2^{n-(k-m)} - 2^i)}{\prod_{i=0}^{m-1} (2^m - 2^i)} \right) = \frac{2^{n-m}}{2^{2^m-1}} \cdot \frac{\prod_{i=0}^{m-1} (2^{n-(k-m)} - 2^i)}{\prod_{i=0}^{m-1} (2^m - 2^i)}, \tag{2}$$

constant m-dimensional flats for random Boolean function. So the number of m-dimensional constant flats generated by the DDL algorithm is

$$\frac{\prod_{i=0}^{m-1} (2^n - 2^i)}{\prod_{i=0}^{m-1} (2^{n-(k-m)} - 2^i)} > 2^{(k-m)m}, \tag{3}$$

times greater than by our algorithm.

We have implemented our algorithm in C++ language. Our implementation makes use of another optimization described in [3]: if a set X of parallel constant m-dimensional flats has cardinality lower then 2^{k-m}, then it is impossible to obtain a k-dimensional constant flat by combining flats from X.

We obtained the following timings on AMD Athlon XP 1800+:

Checking normality of random function.

n	10	12	14	16	17
time(min)	0.005	0.082	2.03	129	802

Normality of $\mathrm{tr}(x^{2n-2})$ for $x \in \mathbb{F}_{2^n}$.

n	10	12	14	16
time(min)	0.004	0.08	1.8	119

4. Conclusions

We will recall the definition of Kasami functions now. Let $d = 2^{2k} - 2^k + 1$ with $(k,n) = 1$ and $\alpha \in \mathbb{F}_{2^n}$. Then the function $f_{\alpha,k}\colon \mathbb{F}_{2^n} \to \mathbb{F}_2$ defined by the formula $f_{\alpha,k}(x) := \mathrm{tr}(\alpha x^d)$ is called a *Kasami function*. It is known that some Kasami functions are bent [4]. The authors of [3] checked that the bent function $f_{\alpha,3}\colon \mathbb{F}_{2^{14}} \to \mathbb{F}_2$, where $\alpha \in \mathbb{F}_4 \setminus \mathbb{F}_2$, is non-weakly-normal. This is the first known non-weakly-normal bent function. The DDL algorithm needed about 50 hours on Pentium IV 1.5 GHz. We have verified their result using our algorithm in 34 minutes on Athlon XP 1800+.

Acknowledgments. We want to thank the authors of [3] for preliminary version of their paper and Christopher Wolf for helpful remarks.

REFERENCES

[1] CANTEAUT, A.—DAUM, M.—DOBBERTIN, H.—LEANDER, G.: *Normal and non-normal bent functions*, in: Proc. Int. Workshop on Coding and Cryptography—WCC '03, Versailles, France, March 2003, pp. 91–100.

[2] CARLET. C.: *On cryptographic complexity of Boolean functions*, in: Finite Fields with Applications to Coding Theory, Cryptography and Related Area (G. L. Mullen, H. Tiechtenoth, H. Tapia-Recillas, eds.), 6th International Conference on Finite Fields and Applications, Oaxaca, México, May 21–25, 2001, Springer-Verlag, Berlin, 2002, pp. 53–69.

[3] DAUM, M.—DOBBERTIN, H.—LEANDER. G.: *An algorithm for checking normality of Boolean functions*, in: Proc. Int. Workshop on Coding and Cryptography—WCC '03, Versailles, France, March, 2003, pp. 133–142.

[4] DILLON, J. F.—DOBBERTIN., H.: *New cyclic difference sets with singer parameters*, Finite Fields Appl. **10** (2004), 342–389.

[5] DOBBERTIN, H.: *Construction of bent functions and balanced Boolean functions with high nonlinearity*, in: Fast Software Encryption '94, Lecture Notes in Comput. Sci., Vol. 1008, Springer-Verlag, Berlin, 1995, pp. 61–74.

[6] BREAKEN, A.—WOLF, C.—PRENEEL, B.: *A randomized algorithm for checking the normality of cryptographic Boolean functions*, in: IFIP TCS '04, Vol. 223, Kluwer, Dordrecht, 2004, pp. 51-66.

Received October 18, 2004

Institute of Mathematics
University of Silesia
ul. Bankowa 14
PL–40-007 Katowice
POLAND

E-mail: knowak@ux2.math.us.edu.pl