Classification of Highly Nonlinear Boolean Power Functions with a Randomised Algorithm for Checking Normality

An Braeken, Christopher Wolf, and Bart Preneel
{An.Braeken, Christopher.Wolf, Bart.Preneel}@esat.kuleuven.ac.be
K.U.Leuven, ESAT-COSIC
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium http://www.esat.kuleuven.ac.be/cosic/

Abstract. A Boolean function is called *normal* if it is constant on flats of certain dimensions. This property is relevant for the construction and analysis of cryptosystems. This paper presents an asymmetric Monte Carlo algorithm to determine whether a given Boolean function is normal. Our algorithm is far faster than the best known (deterministic) algorithm of Daum *et al.* In a first phase, it checks for flats of low dimension whether the given Boolean function is constant on them and combines such flats to flats of higher dimension in a second phase. This way, the algorithm is much faster than exhaustive search. Moreover, the algorithm benefits from randomising the first phase. As an application, we determine the level of normality for several, highly nonlinear Boolean power functions.

Keywords: Normality, Boolean Functions, Asymmetric Monte Carlo, Power Functions

Current Revision: 2005-04-15

First Version: 2004-08-30

First Revision: 2004-10-07
Second Revision: 2004-10-11
Third Revision: 2004-11-04

Cryptology ePrint Archive, Report 2004/214
http:://eprint.iacr.org/

A preliminary version of this article under the title "A Randomised Algorithm for Checking the Normality of Cryptographic Boolean Functions" has been published in *Jean-Jacques Levy, Ernst W. Mayr, John C. Mitchell (editors): "Exploring New Frontiers of Theoretical Informatics"* as the proceedings of the *"3rd International Conference on Theoretical Computer Science (TCS2004)", Kluwer Academic Publishers, ISBN 1-4020-8140-5 or 1-4020-8141-3, pp. 51-66;* this conference was a co-located conference within the *"IFIP 18th World Computer Congress (WCC04)", 22–27 August 2004, Toulouse, France.*

1 Introduction

1.1 Motivation

Boolean functions and maps play a central role in cryptology. They are basic building blocks of bit-oriented block and stream ciphers. In order to construct secure cryptographic ciphers, *i.e.*, ciphers which resist all known attacks, it is important to study the structure and behaviour of Boolean functions.

Normality of a Boolean function is the property which determines if the function is constant on a flat of dimension $\lceil n/2 \rceil$. This concept was introduced by [Dob94], in order to construct highly nonlinear balanced Boolean functions. Later, this property was used to distinguish different classes of bent functions. As the first bent function which is non-normal occurs for dimension 14 [CDDL03], we need a highly optimised algorithm for determining the normality of Boolean functions. This is non-trivial as the total number of flats increases exponentially for increasing dimension n [MS91]. Table 1 lists the number of flats of dimension $\lceil n/2 \rceil$; this clearly shows that even for moderate dimensions ($n \geq 13 \ldots 15$) establishing normality by exhaustive search is infeasible.

Table 1. The number of flats of dimension $\lceil \frac{n}{2} \rceil$ to test for different dimensions n

n	8	9	10	11	12	13	14	15	16	17	18	19	20
$\log_2(\#\text{ flats})$	22	26	32	37	44	50	58	65	74	82	92	101	112

1.2 Related Work

The first attempt for determining the normality of a Boolean function, better than exhaustive search, is due to [DDL03]. The main idea of their algorithm is to search exhaustively all flats of small dimension on which the function is constant and then to combine these to flats of higher dimension.

Another deterministic algorithm can be found in [Now04]. In terms of running time, it outperforms both [DDL03] and the algorithm presented in this paper.

1.3 Achievement

In our algorithm, we replace the exhaustive search through all flats of small dimension by a random search. This has several advantages over the algorithm of Daum *et al.* First, we do not need a unique representation of flats which means less conditions to test and therefore a lower time complexity. Second, the number of repetitions needed to determine with high probability that a function is non-normal, is far smaller than an exhaustive search on all flats of small dimension (cf Sect. 5.2). Our algorithm is of the *asymmetric Monte Carlo* type and may output "non-normal" with probability 2^{-c} for a normal function and some confidence level $c \in \mathbb{N}$. The output "normal" is always correct. This asymmetric Monte Carlo algorithm has a far smaller running time than the deterministic algorithm of [DDL03] — even with a reasonable error-probability ($c = 80$ in our case). In particular, we are able to use this algorithm to classify various Boolean power functions of cryptographical interest.

1.4 Outline

This paper is organised as follows. In Sect. 2, we introduce the basic definitions together with a description of the main ideas in our algorithm. Sect. 3 shows as an application of the algorithm the results on the computation of normality for highly nonlinear power functions up to dimension 16. Sect. 4 presents more details and explains several optimisations for our algorithm. In Sect. 5, we give a detailed complexity analysis of the algorithm and compare the total time complexity of our algorithm with the time complexity of the previous algorithm from [DDL03]. This paper concludes with Sect. 6.

2 Background

In this section we present some definitions and a simplified algorithm to test the normality of a Boolean function.

2.1 Definitions

Before we can describe our algorithm, we need to define several objects. We start with vectors and vector spaces and finish with some definitions concerning Boolean functions.

Let a vector $\overline{u} \in \mathbb{F}_2^n$ be represented by the n-tuple $(u_{n-1}, \ldots, u_0)$ with the coefficients $u_i \in \mathbb{F}_2$ from the field with 2 elements. Let $\overline{u_1}, \ldots, \overline{u_k} \in \mathbb{F}_2^n$ be k linearly independent vectors. Then they form the base of the subspace

$$<U> := <\overline{u_1}, \ldots, \overline{u_k}> := \ \{\alpha_1 \overline{u_1} \oplus \ldots \oplus \alpha_k \overline{u_k} \mid \alpha_i \in \mathbb{F}_2\}.$$

Here, the dimension of $<U>$ is k. For a given vector $\overline{a} \in \mathbb{F}_2^n$, we represent the coset of this subspace by $U_{\overline{a}} := \overline{a} \oplus <U>$. Throughout this paper, we call the coset $U_{\overline{a}}$ a *flat*. The vector $\overline{a}$ of the flat $U_{\overline{a}}$ is called the *offset* of this flat. In addition, two flats are said to be *parallel* if they are cosets of the same subspace $<U>$, *i.e.*, all flats of the form $U_{\overline{a}}, \overline{a} \in \mathbb{F}_2^n$ are parallel flats by this definition. Finally, we denote the set of all flats of dimension s by Flat_s, *i.e.*,

$$\mathrm{Flat}_s := \{U_{\overline{a}} \mid \overline{a} \in \mathbb{F}_2^n,\ <U> \subseteq \mathbb{F}_2^n,\ \dim <U>= s\}.$$

We now move on to Boolean functions. A Boolean function f is a mapping from $\mathbb{F}_2^n$ into $\mathbb{F}_2$. The property of normality for a Boolean function f is defined as follows:

Definition 1. *A Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is called* normal *if there exists a flat $W_{\overline{a}} \subset \mathbb{F}_2^n$ of dimension $\lceil n/2 \rceil$ such that f is constant on $W_{\overline{a}}$, i.e., $\forall \overline{w} \in W_{\overline{a}} : f(\overline{w}) = c$ for some fixed $c \in \{0, 1\}$. We call the flat $W_{\overline{a}}$ a* witness *for the normality of the function f.*

As we see from Definition 1, the property of normality is related to the question of the highest dimension of the flats on which the function f is constant. As a consequence, it is natural to generalise the previous definition by the introduction of k-normality [Dub01,Car01]:

Definition 2. *For a natural number $k : 1 \leq k \leq n$, a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$, is said to be "k-normal" if there exists a flat $V_{\overline{a}} \in Flat_k$ such that f is constant on $V_{\overline{a}}$, i.e., $\forall \overline{v} \in V_{\overline{a}} : f(\overline{v}) = c$ for some fixed $c \in \{0, 1\}$. We call the flat $V_{\overline{a}}$ a "k-witness" for the normality of the function f.*

Remark 1. It is clear that a constant function $f(\overline{x}) = c, \forall \overline{x} \in \mathbb{F}_2^n, c \in \mathbb{F}_2$ is n-normal. An affine function $f(\overline{x}) = \overline{a} \cdot \overline{x} \oplus b, \forall \overline{x}, \overline{a} \in \mathbb{F}_2^n, b \in \mathbb{F}_2$ is $(n-1)$-normal, because it is normal on the flats $\{\overline{x} : \overline{a} \cdot \overline{x} \oplus b = 0\}$ and $\{\overline{x} : \overline{a} \cdot \overline{x} \oplus b = 1\}$ of dimension $n - 1$.

2.2 A Simple Algorithm

The previous section shows that it is important for the definition of normality and k-normality, *i.e.*, for a given dimension $e := k$ (k-normality) or $e := \lceil n/2 \rceil$ (ordinary normality), to find a witness $W_{\overline{a}} \in \mathrm{Flat}_e$. To ease the understanding of the algorithm of Sect. 5, we start with a highly non-optimised version of it (cf Fig. 1). Both algorithms are based on the observation made by [DDL03], that a Boolean function which is constant on a flat $W_{\overline{a}}$ is also constant on all flats contained in $W_{\overline{a}}$, *i.e.*, $f_{|W_{\overline{a}}} = c$ for some $c \in \{0, 1\}$ implies $f_{|V_{\overline{b}}} = c$ for all $V_{\overline{b}} \subseteq W_{\overline{a}}$. We call the flat $V_{\overline{b}}$ a *sub-witness* of $W_{\overline{a}}$.

Fig. 1. Simplified Algorithm for Checking Normality

Input: function f, start dimension s, end dimension e, repetitions r
Output: 1 if the function is e-normal
for $i \leftarrow 1$ **to** r **do**
  pick a flat $U_{\overline{a}} \in_R \mathrm{Flat}_s$ at random
  if $f_{|U_{\overline{a}}} = c$ for some $c \in \{0, 1\}$ **then** SearchFurther($U_{\overline{a}}, e$)
endfor

procedure SearchFurther($U_{\overline{a}}, e$)
  $c = f_{|U_{\overline{a}}}$
  if $\dim U_{\overline{a}} = e$ **then**
    OUTPUT 1
  endif
  forall $\overline{b} \in \mathbb{F}_2^n \setminus U_{\overline{a}}$ **do**
    if $(f_{|U_{\overline{b}}} = c)$ **then** SearchFurther($\overline{a} \oplus <U, \overline{a} \oplus \overline{b}>, e$)
  endfor
endproc

Our algorithm starts with a randomly chosen flat $U_{\overline{a}}$ of dimension s, the *starting dimension*. If this flat is a sub-witness, the function f must be constant on it. So, if the function f is constant on the flat $U_{\overline{a}}$, this is a possible candidate for a sub-witness and we search for a parallel flat $U_{\overline{b}}$, on which the function is constant, too. Both flats $U_{\overline{a}}, U_{\overline{b}}$ can now be combined to a flat of higher dimension, namely $\overline{a} \oplus <U, \overline{a} \oplus \overline{b}>$. We repeat this process recursively until we reach the "end dimension" e. In this case, we have found a witness $W_{\overline{a}}$ and output 1.

Depending on the "confidence level" c we want to achieve, we need to repeat the above algorithm several times. The value for r, *i.e.*, the number of repetitions, depends on c. We discuss the choice of r in Corollary 1 (cf Sect. 5.1).

3 Applications: Classification of Highly Nonlinear Power functions

Using our algorithm for normality, we investigated the level of normality and weakly normality of some non quadratic highly nonlinear Boolean power functions. For even dimension of n, we consider the Kasami class [Kas71], the Cusick and Dobbertin classes ([CD96]) and the Dobbertin classes [Dob98]. For odd dimensions, the Welch class, [CCD00] and [DHKM01] and also the Niho class, [DHKM01] and [Dob99], are considered. The inverse class, [Nyb93], is investigated for all dimensions $n = 8...16$.

The results can be found below. Note that all these power functions in odd dimensions $n \geq 15$ are non $\lceil \frac{n}{2} \rceil$-normal and weakly normal. For even dimension and $n \geq 16$, all these power functions are non-normal except the power function with power 21931 belonging to the Dobbertin class ($\sum_{i=0}^{\frac{n}{2}} 2^{ik}$ with $k = 7$). As expected, the level of normality highly decreases with respect to the dimension. It is left as an open problem how to prove these results analytically. For instance, the inverse function is normal on exactly one flat for dimensions 8, 10, 12, 14, and 16.

The inverse: $2^n - 2$, $n = 0 \mod 2$

n	Power	# $\frac{n}{2}$-flats (nor)
8	254	27
10	1022	1
12	4094	1
14	16382	1
16	65534	1

The inverse: $2^n - 2$, $n = 1 \mod 2$

n	Power	# $\lfloor \frac{n}{2} \rfloor$-flats (nor)	# $\lfloor \frac{n}{2} \rfloor$-flats (w-nor)
9	510	0	35.496
11	2046	0	1122
13	8190	0	0
15	32766	0	0

Kasami: $2^{2k} - 2^k + 1$, $n = 2 \mod 4$, $\gcd(k, n) = 2$, $k < \frac{n}{2}$

n	k	Power	# $\frac{n}{2}$-Flats (nor)
10	2	13	110
10	4	241	7
14	2	13	1
14	4	241	1

Kasami: $2^{2k} - 2^k + 1$, $n = 1 \mod 2$, $\gcd(k, n) = 1$, $k < \frac{n}{2}, k \neq 1$

n	k	Power	# $\lfloor \frac{n}{2} \rfloor$-flats (nor)	# $\lfloor \frac{n}{2} \rfloor$-flats (w-nor)
9	2	13	3798	76.416
9	4	241	1773	38.988
11	2	13	2574	98.766
11	3	57	77	3354
11	4	241	88	1892
11	5	993	33	1793
13	2	13	0	13.390
13	3	57	0	0
13	4	241	0	0
13	5	993	0	0
13	6	4033	0	0
15	2	13	0	0
15	4	241	0	0
15	6	4033	0	0
15	7	16257	0	0

Cusick and Dobbertin: $2^{\frac{n}{2}} + 2^{\frac{n+2}{4}} + 1$, $n = 2 \mod 4$

n	Power	# $\frac{n}{2}$-Flats
10	41	432
14	145	0

Cusick and Dobbertin: $2^{\frac{n}{2}} + 2^{\frac{n}{2}-1} + 1$, $n = 2 \mod 4$

n	Power	# $\frac{n}{2}$-Flats
10	49	592
14	193	2142

Dobbertin: $\sum_{i=0}^{\frac{n}{2}} 2^{ik}$, $n = 0 \mod 4$, $\gcd(k,n) = 1$, $k < \frac{n}{2}$

n	k	Power	# $\frac{n}{2}$-Flats
8	1	31	65
8	3	91	117
12	1	127	1
12	5	1387	18
16	1	511	1
16	3	37741	1
16	5	51001	1
16	7	21931	257

Dobbertin: $2^{\frac{n}{2}} + 2^{\frac{n}{4}} + 1$, $n = 4 \mod 8$

n	Power	# $\frac{n}{2}$-Flats
12	73	3800

Welch: $2^{\frac{n-1}{2}}$, $n = 1 \mod 2$

n	Power	# $\lfloor \frac{n}{2} \rfloor$-flats (nor)	# $\lfloor \frac{n}{2} \rfloor$-flats (w-nor)
9	19	3546	76.416
11	35	2453	98.766
13	67	195	13.338
15	131	0	0

Niho (1): $2^{\frac{n-1}{2}} + 2^{\frac{n-1}{4}} - 1$, $n = 1 \mod 4$

n	Power	# $\lfloor \frac{n}{2} \rfloor$-flats (nor)	# $\lfloor \frac{n}{2} \rfloor$-flats (w-nor)
9	19	3546	76.416
13	71	0	0

Niho (2): $2^{\frac{3n-1}{4}} + 2^{\frac{n-1}{2}} - 1$, $n = 3 \mod 4$

n	Power	# $\lfloor \frac{n}{2} \rfloor$-flats (nor)	# $\lfloor \frac{n}{2} \rfloor$-flats (w-nor)
11	287	55	1694
15	2175	0	0

4 Optimisations

After given a short outline of our algorithm and also showing an application to different, cryptographically important power functions, we describe how to optimise it. We want to stress that the results of the previous section would not have been possible with the simplified algorithm as it has a far too high time-complexity. Hence, the optimisations of the following section are worthwhile.

4.1 Complement Vector Space

There are in total $2^n - 2^s$ parallel flats $U_{\overline{a}}, \overline{a} \in \mathbb{F}_2^n \setminus <U>$ for a given subspace $<U>$ of dimension s. However, some parallel flats are equivalent as they contain the same points.

Example 1. Consider some parallel flats of the following subspace of dimension 2 which is defined by $<U> := <(0,0,1),(0,1,0)> \subseteq \mathbb{F}_2^3$.

$$\begin{aligned}(1,0,0) \oplus <(0,0,1),(0,1,0)> &= (1,1,0) \oplus <(0,0,1),(0,1,0)> \\ &= (1,1,1) \oplus <(0,0,1),(0,1,0)> \\ &= (1,0,1) \oplus <(0,0,1),(0,1,0)>\end{aligned}$$

As a consequence, the parallel flats can be divided into equivalence classes. Therefore, we use the *complement* of a subspace $<U>$, *i.e.*, the subspace $<\overline{U}>$ which satisfies

$$<\overline{U}> \oplus <U> = \mathbb{F}_2^n \text{ and } <\overline{U}> \cap <U> = \{0\}.$$

This allows us to determine the representatives of the equivalence classes of the parallel flats, namely the flats $U_{\overline{a}}$, for $\overline{a} \in <\overline{U}>$. Because the dimension of $<\overline{U}>$ is equal to $n - s$, there are in total 2^{n-s} different parallel flats. To compute the complement $<\overline{U}>$ of a given subspace $<U>$ efficiently, we make use of the *Permuted Gauss Basis* (PGB) of a subspace. To define the PGB, we need to introduce the concept of left-most-one of a vector first.

Definition 3. *For a given vector $\overline{u} = (u_{n-1}, \ldots, u_0)$, we define the* left-most-one *as the position of the left-most one in its representation:*

$$\nu(\overline{u}) := \min\{i \in \{-1, \ldots, n-1\} \mid u_j = 0 \ \textit{for} \ \ i < j \le n\}.$$

Definition 4. *The vectors $\overline{u_1}, \ldots, \overline{u_k}$ form a PGB basis iff*

$$\nu(\overline{u_i}) \ne \nu(\overline{u_j}), \quad 0 \le i < j < n\,.$$

Remark 2. The name Permuted Gauss Basis is motivated as follows. Thinking about the base vectors $\overline{u_1}, \ldots, \overline{u_k}$ as a matrix, we would perform Gaussian elimination on it, without swapping rows. The result would not be a triangular structure but a row permutation.

For a subspace $<U>$, we denote the set of the different left-most-ones of its elements

$$\Upsilon(<U>) := \{\nu(\overline{u}) \mid \overline{u} \in <U> \setminus\{0\}\}.$$

The complement $<\overline{U}>$ of a subspace $<U>$ where $<U>$ is in PGB can be computed as follows:

$$<\overline{U}> = \{\overline{a} \in \mathbb{F}_2^n \mid a_i = 0, \text{ where } i \in \Upsilon(<U>)\}\,.$$

4.2 Random Points instead of Random Bases

Instead of selecting a random flat with a PGB, we choose $(s+1)$ points at random. This is cheaper than selecting a vector space at random which satisfies the PGB-criterion. In addition, we only need to transfer a set of $(s+1)$ points into a PGB if the function f is constant on the corresponding flat. As this only happens with probability 2^{-2^s+1}, we obtain very low costs on average. For s points, we can compute the PGB by the iterative algorithm from Fig. 2. The point $\overline{p}_0$ is the offset of the flat $\overline{p}_0 \oplus <\overline{p}_1, \ldots, \overline{p}_s>$ and has to be reduced as outlined in the previous section.

Fig. 2. Algorithm for computing the PGB of a set of points

procedure ComputePGB($\overline{p}_1, \ldots, \overline{p}_s$)
  Input: s points $\overline{p}_1, \ldots, \overline{p}_s$
  Output: a PGB of the $\overline{p}_1, \ldots, \overline{p}_s$
  for $k \leftarrow 2$ **to** s **do**
    while $\nu(\overline{p}_k) \in \{\nu(\overline{p}_1), \ldots, \nu(\overline{p}_{k-1})\}$ **do**
      for $i \leftarrow 1$ **to** $k-1$ **do**
        if $\nu(\overline{p}_i) = \nu(\overline{p}_k)$ **then** $\overline{p}_k \oplus \leftarrow \overline{p}_i$
endproc

Finally, we have to check whether the $(s+1)$ points form a flat of dimension s. The contrary happens only with very small probability:

$$(2^n)(2^n-1)\cdots(2^n-2^s)/2^{n\cdot(s+1)}.$$

4.3 Combining

In the original algorithm, we searched for all parallel flats and started a recursion on each of them. This is obviously superfluous as we will find the same witness several times this way. As we know from the previous section, we will obtain at least 2^{e-s} parallel flats $U_{\overline{b_i}}$ on which the function is constant. Here, e denotes the end-dimension and s the start-dimension.

To avoid this costly computation, we use a different strategy, based on [DDL03]: instead of recursively searching for all parallel flats of higher dimension, we combine flats of low dimension to obtain flats of higher dimension. This is based on the following observation:

$$(\overline{b}_i \oplus <U>) \cup (\overline{b}_j \oplus <U>) = \overline{b}_i \oplus <U, \overline{b}_i \oplus \overline{b}_j> \ .$$

Hence, we only need to consider pairs $(\overline{b}_i, \overline{b}_j) \in <\overline{U}> \times <\overline{U}>$ which lead to the same sum and then combine them recursively until we obtain a flat of dimension e. To do this efficiently, we introduce 2^n lists (depending on a vector $\overline{v} \in \mathbb{F}_2^n$) which hold an offset for each possible sum, *i.e.*, Append$(L^{\overline{b_i} \oplus \overline{b_j}}, \overline{b_i})$. In the following section, we develop a branching condition for the combine method, which allows to decrease its running time even further.

4.4 Branching

Let the function f take a constant value $c \in \{0,1\}$ on the flat $U_{\overline{a}}$ of dimension d. Denote with $P(U_{\overline{a}})$ the set of all flats parallel to $U_{\overline{a}}$ on which the function yields the same constant. The following branching condition defined by the cardinality of the set $P(U_{\overline{a}})$ has been observed by [DDL03]. We are able to improve their result by giving a shorter proof.

Theorem 1. *If $|P(U_{\overline{a}})| < 2^{e-d}$, we can terminate the current branch of the combine-method in $<U>$ without violating its correctness.*

Proof. Let $W_{\overline{b}}$ be a witness and $U_{\overline{a}} \subset W_{\overline{b}}$ its subwitness. Now, there exist exactly $(e-d)$ linearly independent vectors $\overline{w_1}, \ldots, \overline{w_{e-d}} \in <W>$ with $\overline{w_1}, \ldots, \overline{w_{e-d}} \notin <U>$ and consequently $\overline{w_1}, \ldots, \overline{w_{e-d}} \in <\overline{U}>$. These vectors exist due to dimension reasons as $\dim W_{\overline{b}} = e$ and $\dim U_{\overline{a}} = d$. Therefore, for any subwitness $U_{\overline{a}} \subset W_{\overline{b}}$ exist 2^{e-d} parallel subwitnesses. This implies that $|P(U_{\overline{a}})| \geq 2^{e-d}$. As a consequence, we can stop at any step in the algorithm if this condition is violated because we will not be able to extend the flat $U_{\overline{a}}$ to a witness of dimension e. □

5 The Improved Algorithm

Using the ideas from the previous section, we obtain the algorithm of Fig. 3. The method SearchForParallelFlats can be found in Fig. 4 and the optimised version of the combine-method is presented in Fig. 5. In the following sections, we analyse this optimised algorithm.

Fig. 3. Main loop for the optimised algorithm

Input: function f, start dimension s, end dimension e, repetitions r
Output: one witness if the function is e-normal
for $i \leftarrow 1$ **to** r **do**
  $S_0 \leftarrow \{\}$, $S_1 \leftarrow \{\}$
  for $i \leftarrow 1$ **to** $2s+1$ **do**
    $\overline{p} \in_R \mathbb{F}_2^n$
    $c \leftarrow f(\overline{p})$
    $S_c \cup \leftarrow \{\overline{p}\}$
  endfor
  if $((|S_0| \neq s+1)$ **and** $(|S_1| \neq s+1))$ **then continue**
  $c \leftarrow |S_1| - s$
  if $f|_{\overline{p}_0 \oplus <\overline{p}_0 \oplus \overline{p}_1, \ldots, \overline{p}_0 \oplus \overline{p}_s>}$ $(p_i \in S_c, i \in \{0, \ldots, s\})$ *not constant* **then continue**
  $\overline{a} \oplus <U> \leftarrow$ ComputePGB$(\overline{p_0}, \ldots, \overline{p_s})$
  if $\dim <U> \neq s$ **then continue**
  SearchForParallelFlats($<U>$)
endfor

Fig. 4. SearchForParallelFlats for the optimised algorithm

procedure SearchForParallelFlats($<U>$)
  $<\overline{U}> \leftarrow$ ComputeComplement($<U>$)
  $L \leftarrow \emptyset$, $c \leftarrow f(\overline{a})$
  for $\overline{b} \in <\overline{U}> \setminus \{\overline{a}\}$ **do**
    if $f_{|U_{\overline{b}}} = c$ **then** Append$(L, \overline{b})$
  if $|L| \geq 2^{e-s}$ **then** Combine($<U>$,L)
endproc

Fig. 5. Combine-method for the optimised algorithm

Global Initialisation:
forall $\overline{a} \in \mathbb{F}_2^n$ **do**
  $L^{\overline{a}} \leftarrow \emptyset$

procedure Combine($<U>, L$)
  $d \leftarrow \dim <U>$
  if $d \geq e$ **then**
    Let $\overline{a} \in L$
    OUTPUT $U_{\overline{a}}$
  endif
  forall $(\overline{b_i}, \overline{b_j}) \in L \times L : i < j$ **do**
    Append($L^{\overline{b_i} \oplus \overline{b_j}}$, $\overline{b_i}$)
  forall $(\overline{b_i}, \overline{b_j}) \in L \times L : i < j$ **do**
    $\overline{a} \leftarrow \overline{b_i} \oplus \overline{b_j}$
    if $|L^{\overline{a}}| \geq 2^{e-d-1}$ **then**
      $L' \leftarrow \emptyset$
      forall $\overline{b} \in L^{\overline{a}}$ **do**
        if $\overline{b} \in \overline{<U, \overline{a}>}$ **then** Append(L', $\overline{b}$) **else** Append(L', $\overline{a} \oplus \overline{b}$)
      Combine($<U, \overline{a}>, L'$)
    endif
    $L^{\overline{a}} \leftarrow \emptyset$
  endfor
endproc

5.1 Complexity Analysis

We start the analysis of the algorithm with determining the number r of repetitions. Then we analyse the complexity of the main loop from Fig. 3, the complexity of the SearchForParallelFlats from Fig. 4 and the complexity of the Combine-procedure from Fig. 5 in different steps.

Number of Repetitions
For determining the number of repetitions, we need the following lemma from [MS91], concerning the number of subspaces and flats of a certain dimension in a vector space.

Lemma 1. *The number of subspaces of dimension s in a vector space of dimension n is given by*

$$NS(n,s) := \prod_{i=0}^{s-1} \frac{2^{n-i}-1}{2^{s-i}-1}.$$

The number of flats of dimension s in a vector space of dimension n is given by

$$NF(n,s) := 2^{n-s} \prod_{i=0}^{s-1} \frac{2^{n-i}-1}{2^{s-i}-1} = 2^{n-s} NS(n,s).$$

Before determining a bound on r, we first introduce the term complaisant flat.

Definition 5. *A flat $U_{\overline{a}}$ is called* complaisant *if the function is constant on the flat, the flat is parallel to a sub-witness, but the flat is not contained in any witness.*

Theorem 2. *When choosing* $(s+1)$ *points* $\overline{p}_0, \ldots, \overline{p}_s \in \mathbb{F}_2^n$ *at random, the probability* $PF(n,s,e)$ *that the flat* $U_{\overline{a}}$ *formed by these* $(s+1)$ *points pass the first step in the algorithm is equal to*

$$PF(n,s,e) = Pr(U_{\overline{a}} \text{ is a sub-witness}) + Pr(U_{\overline{a}} \text{ is a complaisant flat})\,,$$

where

$$Pr(U_{\overline{a}} \text{ is a sub-witness}) := 2^{e-n} \cdot \prod_{i=1}^{s} \frac{2^e - 2^{i-1}}{2^n}$$
$$Pr(U_{\overline{a}} \text{ is a complaisant flat}) := 2^{-2^s+1} \cdot \frac{2^{n-e} NS(n,s) - NF(e,s)}{NS(n,s)}\,.$$

In the above formula, e is the dimension of the witness. The formulas for $NS(\cdot,\cdot)$ *and* $NF(\cdot,\cdot)$ *are given in Lemma 1.*

Proof. We first determine the probability that the flat $U_{\overline{a}}$ is a sub-witness. This probability is justified with an inductive argument on the dimension of the sub-witness: for one point (*i.e.*, a flat of dimension 0), the probability of being a sub-witness is $\frac{2^e}{2^n}$. Here, the witness has 2^e points. This probability is also true for extending the sub-witness from dimension $(i-1)$ to dimension i (we have $1 \leq i \leq s$). In addition, we have to consider the case $\overline{p}_i \in \overline{p}_0 + <\overline{p}_1, \ldots, \overline{p}_{i-1}>$, *i.e.*, the new point $\overline{p}_i$ lies in the sub-witness of dimension $(i-1)$ generated by the points $\overline{p}_0, \ldots, \overline{p}_{i-1}$.

The probability that $U_{\overline{a}}$ is a complaisant flat is equal to the probability that the function is constant on $U_{\overline{a}}$ times the number of flats which are parallel with a witness but not part of a witness. This is exactly expressed in the formula. □

From the previous theorem we can deduce the following corollary.

Corollary 1. *For a given start dimension s and an end dimension e, we need at most*

$$Rep(n,s,e,c) = \frac{c}{PF(n,s,e)}$$

repetitions to achieve a confidence of 2^{-c} *that the function f is not e-normal.*

Table 2 shows some numerical values of r in $\log_2$. In this and all following tables, we concentrate on even choices for n and fix $e = \frac{n}{2}$ as these cases are particularly relevant in cryptography.

Table 2. Number of repetitions (in $\log_2$) for different values of n and s

$s \backslash n$	8	10	12	14	16	18	20
2	18.60	21.46	24.39	27.35	30.33	33.33	36.32
3	23.02	26.65	30.48	34.40	38.36	42.34	46.33
4		32.06	36.67	41.49	46.40	51.36	56.34

Complexity of the main loop

Obviously, picking $(s + 1)$ random points and checking if the function is constant for a given flat, will be the most expensive operations. Therefore, we start with a lemma on the average complexity for checking that a function is constant on a given set of points.

Lemma 2. *For a given random function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ and a given set of points $P \subseteq \mathbb{F}_2^n$, the algorithm from Fig. 6 needs on average 3 evaluations of f to check if this function is constant when restricted to vectors in the set P.*

Fig. 6. Algorithm to determine if a function is constant on a set of points

Input: function f, a set P with $p := |P|$ points
Output: 1 if f is constant on P and 0 otherwise
Let $\overline{q}_1 \in P$, $c \leftarrow f(\overline{q}_1)$
for $\overline{q} \in P \setminus \{\overline{q}_1\}$ **do**
  if $f(\overline{q}) \neq c$ **then** OUTPUT 0
OUTPUT 1

Proof. The average number of evaluations depends on the number of points $p := |P|$ of this algorithm; it is given by

$$Ev(p) := \sum_{i=1}^{p-1} \frac{1}{2^i}(i + 1) + \frac{1}{2^{p-1}} p = 3 - \frac{1}{2^{p-2}} .$$

To justify this formula, we observe that we need to evaluate f at least once to obtain the constant c. As the function is a random function by definition, we have a probability of $\frac{1}{2}$ to obtain a different constant for every further evaluation, *i.e.*, to terminate this algorithm. After checking a total of p points, the algorithm terminates. For this last check, we still have a probability of $\frac{1}{2}$ to output 0. However, the workload of outputting 0 or 1 is exactly the same, namely p evaluations. □

As a consequence, the complexity of the main loop so far depends on the costs of picking the $(s + 1)$ random points at random, evaluating the function f on the corresponding flat and some other negligible operations whose complexity we set to one, *i.e.*, $(3+3+1)r = 7r$, where r represents the number of repetitions.

Complexity of the SearchForParallelFlats-method

From a computational point of view, the for-loop is very expensive, as we have to check $2^{n-s} - 1$ parallel flats every time. However, each flat costs only 3 operations on average (cf Lemma 2). In addition, we only need this for-loop in 2^{-2^s+1} of all cases as this is the probability that the function is constant on the corresponding flat. The other steps in the method are negligible in comparison to the for-loop. We therefore identify their average workload as 1. Consequently, the complexity of the SearchForParallelFlats-method can be approximated by $3 \cdot 2^{n-2^s-s+1}r$, where r denotes the number of repetitions.

Consequently, the total complexity of the main loop without the combine procedure equals to $(7 + 3 \cdot 2^{n-2^s-s+1})r$. Numerical values for the time-complexity (in $\log_2$) of the main loop without the combine procedure are presented in Table 3.

Comparing these numbers with the results of the overall complexity of exhaustive search (cf Table 1 on Sect. 1), we see that we achieve far lower numbers. This completes our complexity analysis of the main loop. We go on with an analysis of the combine-procedure.

Table 3. Time-complexity (in $\log_2$) of the main loop and "search for parallel flats"

n	$s = 2$	$s = 3$	$s = 4$	$s = 5$
8	24.83	24.94		
10	31.22	29.40	33.65	
12	38.02	35.23	38.27	44.67
14	44.95	42.32	43.12	50.27
16	51.92	50.03	48.16	56.08
18	58.91	57.95	53.59	61.99
20	65.91	65.92	59.92	67.95

Complexity of the Combine-procedure

The complexity analysis of the combine-procedure is a little more tricky. In particular, we have to deal with the problem that its complexity depends quadratically on the number of parallel flats we find, *i.e.*, the number $|P(U_{\overline{a}})|$ for a given flat $U_{\overline{a}}$. Therefore, we cannot simply take the average number of flats for this analysis as the result does not reflect the real time complexity of this algorithm. In addition, we have to deal with the branching condition (cf Sect. 4.4).

As we did not expect to find a closed formula for the time complexity of the combine-procedure, we used [MAG] to compute it numerically. As all computations are done with rational numbers, there are no rounding errors in MAGMA. In particular, we computed the probability for the different numbers of parallel flats we obtain in the searchForParallelFlats-method. We only took numbers $\geq 2^{e-s}$ into account (cf Thm. 1) and neglected levels of recursion which appear with too small probability ($< 2^{-40}$), due to the branching condition. In addition, we truncated the sum at points which did not contribute to the overall workload anymore (expected workload smaller than 1). We present the corresponding values ($\log_2$) for different choices of n and s in Table 4.

Table 4. Numerical results for the time-complexity (in $\log_2$) of the Combine-method

n	$s = 2$	$s = 3$	$s = 4$	$s = 5$
8	24.17	15.97		
10	31.15	22.87	≈ 0	
12	38.03	15.76	≈ 0	≈ 0
14	44.97	23.68	≈ 0	≈ 0
16	51.93	35.02	≈ 0	≈ 0
18		43.34	≈ 0	≈ 0
20		51.33	≈ 0	≈ 0

These computations were matched by our empirical results. In particular, the branching condition proved to be very powerful for $s \geq 3$ and $n \geq 12$ (note difference between $n = 10$ and $n = 12$ for $s = 3$). In these cases, we never needed a recursive call of the combine-method for non-normal functions. In addition, the probability for a function to be constant on a given flat decreases exponentially with increasing dimension of the flat. Therefore, we expect to find less than 2^{e-s} flats for $s \geq 4$ and $n \leq 20$ which means that the combine-method is never invoked in these cases (fields with ≈ 0 in the above table).

All in all, it is necessary to chose the starting dimension s correctly, *i.e.*, high enough such that the combine-method is still efficient and low enough such that SearchForParallelFlats and the main loop do not need too much time. For dimension $n \geq 10$, the choice $s = 3$ turns out to be optimal.

Asymptotic Analysis

Here we sketch the asymptotic analysis of the above algorithm: we begin with the observation that for large n and subsequently large s, the running time will only depend on the number of repetitions necessary. We justify this reasoning as follows: as we saw for the combine-method, we have a very powerful branching condition, *i.e.*, asymptotically, this part will not contribute to the overall complexity. The same is true for the search of parallel flats: we have a complexity of $O(2^{(-2^s+1)(n-s)})$ here, *i.e.*, negligible for $n \to \infty$. For our analysis, we chose $s = \frac{1}{4}n$ and $e = \frac{1}{2}n$ and obtain the following asymptotically upper bound on the number of repetitions and thus the running time of the algorithm:

$$Rep(n, \frac{1}{4}n, \frac{1}{2}n, c) = O(c.2^{\frac{1}{8}n^2+\frac{3}{4}n}) \,,$$

where c is the target confidence level. To obtain this upper bound, we observe that the probability to have a complaisant flat is asymptotically very small. In addition, we notice that for large n the factor $2^{e-n+s(e-1-n)}$ is a tight lower bound on the probability $PF(n, s, e)$. Using Theorem 2 and Corollary 1 yields the result.

5.2 Comparison with the Algorithm from Daum *et al.*

In Table 5 we compare the time complexities of our algorithm with that of [DDL03], for computing the normality of a function in dimension n. We are not aware of an asymptotic analysis of the algorithm from [DDL03].

Table 5. Comparison of the time-complexity (in $\log_2$)

n	s	[DDL03]	Our alg.
14	2	**43**	45
	3	46	**42**
	4	52	43
16	2	**52**	52
	3	54	**50**
	4	62	48

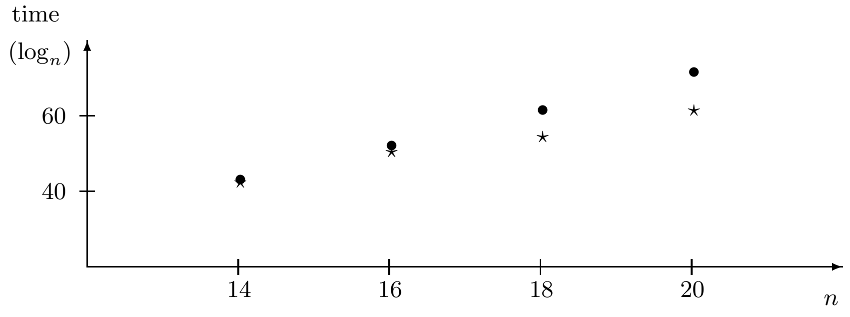
n	s	Daum *et al.*	Our alg.
18	2	**61**	> 60
	3	**61**	58
	4	> 61	**54**
20	2	**71**	> 60
	3	**71**	58
	4	>71	**60**

The time complexity of algorithm of [DDL03], is computed using the formulas given there. According to these results, we expect that it is outperformed by our algorithm for increasing dimension n, cf Fig. 7.

5.3 Empirical Results

We have implemented our algorithm in a programme with 14,000 lines of C++ code. Checking random functions on an AMD Athlon XP 2000+, we obtained the following results for $e = \frac{n}{2}$ (normality) and $s = 3$:

Fig. 7. Time-complexity (in $\log_2$) of this paper ($\star$) and from [DDL03] ($\bullet$)



n	8	10	12	14	16
time [min]	0.020	0.364	16.6	300.8	21,320

As we see in this table, the running time gets quickly out of hand. According to [DDL03], their programme needs approximately 50 h on a Pentium IV 1.5 GHz for the case $n = 14$. Our algorithm needs approximately 5 h for $n = 14$ and approximately 15 d for $n = 16$. Using the complexity analysis of [DDL03], we expect a running time of more than a year for their algorithm to handle functions of dimension $n = 16$.

For our C++ implementation, we have included several improvements:

Combinatorial Gray codes In order to compute vectors more efficiently for a given basis, we used combinatorial Gray codes [Sav97] and computed all intermediate values in a Gray code like fashion. This way, we only needed one computation on average rather than $\frac{n}{2}$ when computing elements of the vector space $<\overline{U}>$.

Optimised Pseudo-Random Number Generator As the programme spends approx. 60% of its time computing random numbers, we concluded that it could benefit from a fast way of generating pseudo-random numbers. However, due to the high number of repetitions, we still need a long period for the pseudo-random number generator. To meet both aims, we used a pseudo-random number generator from [Rho] which combines a multiply with carry generator and a simple multiplicative generator. It achieves a period of more than 2^{60}, has good statistical properties, and is also very fast according to our measurements. For the future, tests with the cryptographically secure pseudo-random number generator using Shamir's T-functions class [KS04] are certainly an interesting option.

Function storage For the Boolean function to be checked, we can use several ways of storing it: bit-wise, byte-wise or in processor-words (32 bit). To make the best use of the internal cache of the processor, a bit-wise storage turned out to have the best performance for dimensions $n \geq 12$. For dimensions $n \leq 10$, an word-wise storage was clearly better as we do not have the overhead of retrieving single bits from a word.

6 Conclusions

In this paper, we present a fast asymmetric Monte Carlo algorithm to determine the normality of Boolean functions. It uses the fact that a function which is constant on a flat of a certain dimension is also constant on all sub-flats of lower dimension. Starting with flats of dimension s and combining them until a flat of dimension e is obtained, we achieve a far lower time-complexity than with exhaustive search on flats of dimension e.

In particular, this algorithm is far faster than the previously known algorithm (5 h in comparison to 50 h) for dimension 14 (cf 5.2). Moreover, it is the first time that the important case $n = 16$ can be computed on non-specialised hardware (previously: more than a year). Using the fact that our algorithm can be parallelised easily, this figure can even be improved.

As an application of the algorithm developed in this paper, we investigated several classes of highly nonlinear Boolean power functions up to dimension $n = 16$. Our experiments suggest that the property of normality is not satisfied anymore for increasing n (cf Sect. 3). We have to leave it as an open problem for the cryptographic community to prove these experimental results on the normality of (highly nonlinear) power functions in an analytic way.

7 Acknowledgements

We want to thank the authors of [DDL03], for helpful remarks and sending us both an early and an extended version of their work. Moreover, we want to thank Krzysztof Nowak (University of Silesia, Poland) for helpful remarks, in particular by pointing out errors in our tables, cf Sect. 3, helping us tracking down an error in our implementation of the algorithm, and sending us a preliminary version of his article [Now04].

The authors were partially supported by Concerted Research Action GOA-MEFISTO-666 of the Flemish Government and An Braeken is research assistant of the Fund for Scientific research - Flanders (Belgium).

References

[Car01] Claude Carlet. On the complexity of cryptographic Boolean functions. In *6th Conference on Finite Fields and Applications, 21th – 25th May*, pages 53–69. Gary L. Mullen, Henning Stichtenoth, and Horacio Tapia-Recillas, editors, Springer, 2001.

[CCD00] A. Canteaut, P. Charpin, and H. Dobbertin. A three-valued crosscorrelation function for binary m-sequences: a proof of welch's conjecture. *IEEE Transactions on Information Theory*, IT-46(1):4–8, 2000.

[CD96] T. Cusick and H. Dobbertin. Some new 3-valued crosscorrelation functions on binary m-sequences. *IEEE Transactions on Information Theory*, IT-42:1238–1240, 1996.

[CDDL03] Anne Canteaut, Magnus Daum, Hans Dobbertin, and Gregor Leander. Normal and non-normal bent functions. In WCC [WCC03]. 19 pages.

[DDL03] Magnus Daum, Hans Dobbertin, and Gregor Leander. An algorithm for checking normality of boolean functions. In WCC [WCC03]. 14 pages.

[DHKM01] H. Dobbertin, T. Helleseth, V. Kumar, and H. Martinson. Ternary m-sequences with three-valued crosscorrelation: New decimations of welch and niho type. *IEEE Transactions on Information Theory*, IT-47:1473–1481, November 2001.

[Dob94] Hans Dobbertin. Construction of Bent functions and balanced Boolean functions with high nonlinearity. In *Fast Software Encryption — FSE 1994*, volume 1008 of *Lecture Notes in Computer Science*, pages 61–74. Bart Preneel, editor, Springer, 1994.

[Dob98] H. Dobbertin. One-to-one highly nonlinear power functions on $gf(2^n)$. *Applicable Algebra in Engineering, Communication, and Computation*, 9:139–152, 1998.

[Dob99] H. Dobbertin. Almost perfect nonlinear power functions on $gf(2^n)$: The niho case. *Information and Computation*, 151(1-2):57–72, 1999.

[Dub01] Sylvie Dubuc. *Etude des propriétés de dégénérescene et de normalité des fonctions booléennes et construction des fonctions q-aires parfaitement non-linéaires.* PhD thesis, Université de Caen, 2001.

[Kas71] T. Kasami. The weight enumerators for several classes of subcodes of the second order binary reed-muller codes. *Information and Control*, 18:369–394, 1971.

[KS04] Alexander Klimov and Adi Shamir. New cryptographic primitives based on multiword t-functions. In *Fast Software Encryption — FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 1–15. Bimal K. Roy and Willi Meier, editors, Springer, 2004.

[LB00] David Landau and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics.* Cambridge University Press, 2000. ISBN 0-521-65314-2.

[MAG] Computational Algebra Group, University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry.* `http://magma.maths.usyd.edu.au/magma/`.

[MS91] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* Elsevier Science Publisher, 1991. ISBN 0-444-85193-3.

[Now04] Krzysztof Nowak. Checking normality of Boolean functions. *Tatra Mountins*, 2004. 12 pages, *to appear.*

[Nyb93] K. Nyberg. Differentially uniform mappings for cryptography. In *Advances in Cryptology — EUROCRYPT 1993*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Tor Helleseth, editor, Springer, 1993.

[Rho] Glenn Rhoads. Random number generator in c. `http://remus.rutgers.edu/ rhoads/Code/rands.c`.

[Sav97] Carla Savage. A survey of combinatorical Gray codes. *SIAM Review*, 39(4):605–629, 1997. `http://www.csc.ncsu.edu/faculty/savage/AVAILABLE_FOR_MAILING/survey.ps`.

[WCC03] Daniel Augot, Pascal Charpin, and Grigory Kabatianski, editors. *Workshop on Coding and Cryptography 2003.* l'Ecole Supérieure et d'Appliction des Transmissions, 2003. ISBN 2-7261-1205-6.