A New Approach for finding Low-Weight Polynomial Multiples

Laila El Aimani

University of Cadi Ayyad - ENSA Safi
Route Sidi Bouzid BP 63, 46000 - Safi, Morocco
laila.elaimani@gmail.com

Abstract. We consider the problem of finding low-weight multiples of polynomials over binary fields, which arises in stream cipher cryptanalysis or in finite field arithmetic. We first devise memory-efficient algorithms based on the recent advances in techniques for solving the knapsack problem. Then, we tune our algorithms using the celebrated Parallel Collision Search (PCS) method to decrease the time cost at the expense of a slight increase in space. Both our memory-efficient and time-memory trade-off algorithms improve substantially the state-of-the-art. The gain is for instance remarkable for large weights; a situation which occurs when the available keystream is small, e.g. the Bluetooth keystream.

Keywords: Low-weight polynomial multiple · Stream cipher cryptanalysis · Knapsack · Collision-finding algorithm · Time-memory trade-off.

1 Introduction

We consider the following problem:

Definition 1 (The Low-Weight Polynomial Multiple (LWPM) problem). *Given a binary polynomial $P \in \mathbb{F}_2[X]$ of degree d and a bound n, find a multiple of P with degree less than n and with the least possible weight ω, where the weight of a multiple is the number of its nonzero coefficients.*

The LWPM arises in stream cipher cryptanalysis, and in efficient finite field arithmetic.

Fast correlation attacks [20,17] against LFSR-based (Linear Feedback Shift Register) stream ciphers first precompute a low-weight multiple of the constituent LFSR connection polynomial. In fact, low-weight polynomial multiples are required to keep the bias, of the linear approximation in a correlation attack, as high as possible so as to reduce the cost of key-recovery or distinguishing attacks.

Low-weight polynomial multiples find also application in finite field arithmetic. Actually, von zur Gathen and Nöker[23] found that $\mathbb{F}_{2^d} = \mathbb{F}_2[x]/(g)$, where g is a low-weight irreducible polynomial of degree d, is the most efficient representation of finite fields. However, such polynomials do not always exist. Brent and Zimmerman [3] proposed an interesting solution: take an irreducible polynomial $f \in \mathbb{F}_2[X]$ of degree d but possibly large weight, a multiple g of f with small weight, and work in the ring $\mathbb{F}_2[X]/(g)$ most of the time, going back to the field $\mathbb{F}_{2^d}$ only when necessary.

1.1 Related work

There have been several approaches for computing low-weight multiples of polynomials. Most methods first estimate the minimal possible weight ω of multiples, of the given polynomial P, with degree at most n and with nonzero constant term, then look for multiples of weight at most ω. To estimate the minimal weight, one solves for ω_e the following inequality

$$\binom{n}{\omega_e - 1} \geq 2^d \tag{1}$$

where d is the degree of P; the minimal weight ω is the smallest solution. In fact, if multiples are uniformly distributed, then one expects the inequality to hold. It is worth noting that the number of such multiples can be approximated by $\mathcal{N}_M = 2^{-d}\binom{n}{\omega-1}$.

Given a polynomial $P \in \mathbb{F}_2[X]$ of degree d and a bound n, we summarize below the strategies used to find a multiple of P of degree at most n and with the least possible weight ω. We describe the time or space complexity using the Big-O notation, which denotes the worst case complexity of the algorithms. Also, we use the approximation $\binom{n}{\omega} \approx O(n^\omega)$.

Discrete-log-based techniques They were introduced in [18], then improved and generalized in [8,19]. They work with discrete logarithms in the multiplicative group of $\mathbb{F}_{2^d}$ instead of the direct representation of the polynomials. [8] use a time-memory trade-off to solve the problem in time $O(n^{\lceil \frac{\omega-2}{2} \rceil})$ and memory $O(n^{\lfloor \frac{\omega-2}{2} \rfloor})$. [19] provide a memory-efficient algorithm that runs in approximately $O(\frac{2^d}{n})$. The methods assume however a constant cost of the discrete logarithm computations, using precomputed tables that do not require excessive storage. This is not the case if $2^d - 1$ is not smooth. Also, the methods assume some conditions on the input polynomial: primitive in case of [8] or product of powers of irreducible polynomials with coprime orders in case of [19].

Syndrome decoding This technique reduces LWPM to finding a low-weight codeword in a linear code; a popular problem for which there exist known algorithms to solve it, e.g. the so-called information-set decoding algorithms [21,5,2,15,13,16]. These algorithms introduce many parameters to optimize the running time and the memory consumption according to the problem instance, however, we can approximate the running time by $O(\mathrm{Poly}(n) \cdot (\frac{n}{d})^\omega)$, and the memory complexity by $O(d^\omega)$.

Lattice-based techniques This technique, introduced in [9], reduces the LWPM problem to finding short vectors in an n-dimensional lattice. The method uses the LLL reduction [12] to solve the problem in time $O(n^6)$ and space $O(n \cdot d)$. Unfortunately, this technique gives inaccurate results, i.e. fails to find a multiple with the least possible weight, as soon as the bound n exceeds few hundreds.

Birthday techniques This is by far the standard method for solving the LWPM problem. There exist a plethora of variations and improvements to this method. The standard one runs in $O(n^{\lceil \frac{\omega-1}{2} \rceil})$ and uses $O(n^{\lfloor \frac{\omega-1}{2} \rfloor})$ of memory. Chose et al. [7] cut down the memory utilization to $O(n^{\lfloor \frac{\omega-1}{4} \rfloor})$ using a *match-and-sort* approach. Canteaut and Trabbia [6] introduced a memory-efficient method for solving the LWPM problem that runs in $O(n^{\omega-1})$ and requires only linear memory. When the degree of the multiple gets very large and there are many low-weight multiples, but it is sufficient to find only one, *Wagner's generalized birthday paradox* [24] becomes more efficient. For instance, if $n \geq 2^{d/(1+\log_2(\omega-1))}$, then this method finds a weight-ω multiple of P of degree at most n in $O((\omega-1)n)$ and uses $O(n)$ memory.

1.2 Our Approach

We view the LWPM problem as a special instance of the following subset sum problem:

Definition 2 (Group Subset Sum Problem). *Let $(G, .)$ be an abelian group. Given $a_0, a_1, \ldots, a_n \in G$ together with $\omega, 0 < \omega \leq \frac{n}{2}$ such that there exists some solution $\mathbf{z} = (z_1, \ldots, z_n) \in \{0,1\}^n$ satisfying*

$$\prod_{i=1}^{n} a_i^{z_i} = a_0 \quad \text{with} \quad \mathrm{weight}(\mathrm{z}) = \omega$$

The goal is to recover $\mathbf{z}$ (or some other weight-ω solution $\mathbf{z}$).

This definition generalizes that in [10] as it does not impose the group order to be of bitsize n. It captures then the LWPM problem as follows. Let P be a degree-d polynomial in $\mathbb{F}_2[X]$. Consider further the group $(\mathbb{F}_2^d, +)$ of d-dimensional vectors over $\mathbb{F}_2$, where the group law is the bitwise addition over $\mathbb{F}_2$. A weight-ω multiple $1 + \sum_{i=1}^{n} z_i X^i$ of P, with nonzero constant term and degree at most n satisfies:

$$\sum_{i=1}^{n} z_i a_i = a_0 \quad \text{with } \mathrm{a_i} = \mathrm{X^i} \bmod \mathrm{P}, \;\; 0 \leq \mathrm{i} \leq \mathrm{n}$$

Note that the condition on the weight ($\omega \leq \frac{n}{2}$) is not restrictive. Actually, the searched weight ω is obviously smaller than the weight of P, which is often smaller than $\frac{d}{2}$, and thus smaller than $\frac{n}{2}$. Also, for convenience purposes, we consider throughout the document the relative weight $\omega_n = \omega/n$.

The (group) subset sum problem is one of the most popular and ubiquitous problems in cryptography. It has undergone an extensive analysis with a focus on polynomial-memory algorithms to solve it. In fact, it is known that random-access memory is usually more expensive than time. Most algorithms for solving the subset sum problem [1,10] try to find as many representations as possible of the solution; in fact, the more representations there exist the faster the solution can be found. For example, the folklore algorithm, described in [11], represents the solution $z = x \parallel y$ as a concatenation of two $\frac{n}{2}$-dimensional vectors x and y with weight(x) = weight$(y) = \frac{\omega}{2}$. In the same spirit, [1] split the solution z into two n-dimensional vectors x and y , with weight(x) = weight$(y) = \frac{\omega}{2}$, that add up to z. Recently, [10] further increase the number of representations by splitting z into a sum *over* $\mathbb{Z}$ of two integers of smaller weight by exploiting the carry propagation.

Contributions We view the solution z to LWPM as a collision (x, y) of some random function f mapping from a set $\mathcal{T}$ to itself (in order to use known cycle-finding algorithms to compute collisions). The set $\mathcal{T}$ is determined by how z splits into (x, y). Also, $\mathcal{T}$ ought to allow for many "representations" (x, y) of the solution z, so as to reduce the number of function calls needed before finding a collision. More precisely, we make the following contributions.

First, we present two memory-efficient algorithms for LWPM that improve the state-of-the-art in polynomial-memory algorithms for LWPM. The idea behind the algorithms consists in splitting the solution z into two n-dimensional vectors x and y that add up to z over $\mathbb{F}_2$. The weight of both x and y is some function of ω to be determined.

More precisely, Algorithm 1 assumes and puts in place a Bernoulli distribution on the representation of z, then determines the optimal weight $\phi(\omega)$ to be used for x and y. As a result, we significantly improve the running time offered by the state-of-the-art methods, i.e. the birthday and the discrete-log methods (see Figure 1; the x-axis represents the relative weight $\omega_n = \omega/n$, and the y-axis represents the relative exponent $\log(T)/n$ of the time cost T).

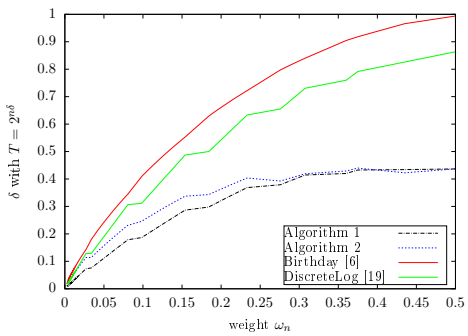
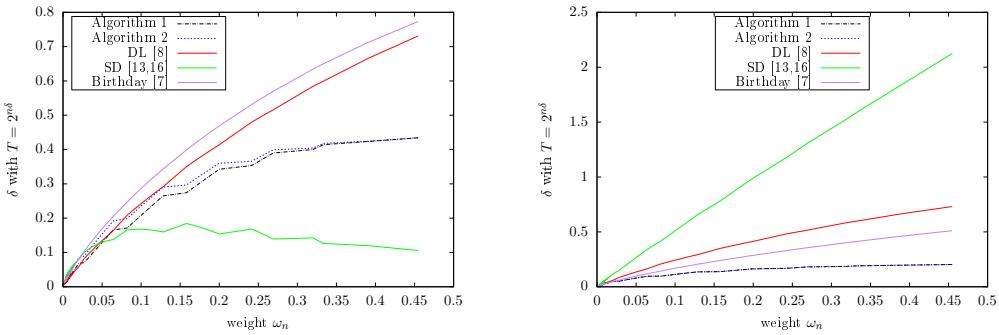


Fig. 1. Comparison between the memory-efficient techniques and our algorithms

Since Algorithm 1 uses a pseudo-random number generator to establish the desired Bernoulli distribution, it incurs a slight overhead in the computations. Therefore, we reinforce our contribution with Algorithm 2 which gets rid of the Bernoulli distribution; the result still substantively outperforms the state-of-the-art (see Figure 1).

We show the practicality of our technique with an implementation of the algorithms that confirm our theoretical estimates.

Second, we tune our algorithms via the Parallel Collision Search (PCS) technique [22] to decrease the running time at the expense of memory. Again, we improve the classic Time-Memory Trade-off (TMTO) or birthday method, described earlier in the text, in both time and space (see Figure 2; the x-axis represents the relative weight $\omega_n = \omega/n$, whereas the y-axis represents the relative exponent $\log(T)/n$ (resp. $\log(M)/n$) of the time (resp. memory) cost T (resp. M)).



(a) **Time complexity** (b) **Memory complexity**

Fig. 2. Comparison between the classic TMTO and our time-memory trade-off algorithms

The rest of the paper is organized as follows. Section 2 recalls the necessary background and establishes the notation that will be used throughout the document. Sections 3 & 4 respectively describe, analyze, and experimentally validate our algorithms. Section 5 compares the performance of our algorithms with the state-of-the-art. Finally, the time-memory trade-off tuning of the proposed algorithms is given in Section 6.

2 Theoretical Background

2.1 Notations and Conventions

Let $a, b \in \mathbb{N}$ with $a < b$. We conveniently write $[a, b] := \{a, a+1 \ldots, b\}$. For a vector $z = (z_1, \ldots, z_n) \in \{0, 1\}^n$, we denote by $\text{weight}(z) := |\{i \in [1, n] \colon z_i = 1\}|$. $\mathbb{Z}_N$ denotes the ring of integers modulo N. $\mathbb{F}_2$ denotes the field of two elements where the additive identity and the multiplicative identity are denoted 0 and 1, as usual. $\mathbb{F}_2[X]$ refers to the ring of polynomials with coefficients in $\mathbb{F}_2$. $\mathbb{R}^+$ denotes the set of positive real numbers.

Let $P \in \mathbb{F}_2[X]$. $\deg(P)$ and $\text{weight}(P)$ refer to the degree and weight of P respectively; the weight of a polynomial in $\mathbb{F}_2[X]$ corresponds to the number of its non-zero coefficients. In the text, we identify polynomials in $\mathbb{F}_2[X]$ with their coefficient vectors. For instance, the sum of two polynomials in $\mathbb{F}_2[X]$ is the sum over $\mathbb{F}_2$ of their coefficient vectors termwise.
Suppose $\deg(P) = d$. $\mathbb{F}_2[X]/P$ denotes the ring of polynomials modulo P; addition and multiplication are performed modulo P. Finally, $(\mathbb{F}_2^d, +)$ refers to the group of d-dimensional vectors over $\mathbb{F}_2$, where the group law $+$ is the bitwise addition and the identity is referred to as $0_{\mathbb{F}_2^d}$.

The Big-O, Θ, and $\tilde{\Theta}$ notations. The Big-O notation represents the upper bound of the running time of an algorithm; it gives then the worst case complexity of an algorithm.

$$O(g) = \{f : \exists\, c, x_0 \in \mathbb{R}^+ : 0 \leq f(x) \leq cg(x)\ \forall x \geq x_0\}$$

The Θ notation represents the upper and the lower bound of the running time of an algorithm. It is useful when studying the average case complexity of algorithms.

$$\Theta(g) = \{f : \exists\, c_1, c_2, x_0 \in \mathbb{R}^+ : 0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)\ \forall x \geq x_0\}$$

The $\tilde{\Theta}$ notation suppresses the polynomial factors in the input. For example $\tilde{\Theta}(2^n)$ suppresses the polynomial factors in n.

Binomial coefficient. The binomial coefficient $\binom{n}{k}$ refers to the number of distinct choices of k elements within a set of n elements. We have: $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$.
Often, we need to obtain asymptotic approximations for binomials of the form $\binom{n}{\alpha\, n}$ or $\binom{n}{\lfloor \alpha\, n \rfloor}$ for values $\alpha \in]0, 1[$. This is easily achieved using Stirling's formula: $n! = (1 + o(1))\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. Thus $\binom{n}{\alpha n} \approx \frac{1}{\sqrt{2\pi\, n\alpha(1-\alpha)}} \cdot 2^{nH(\alpha)}$, where H is the binary entropy function defined as $H(x) := -x\log_2(x) - (1-x)\log_2(1-x)$; $\log_2$ is the logarithm in base 2. We can then write

$$\binom{n}{\alpha n} = \Theta\left(n^{-1/2} 2^{nH(\alpha)}\right) \quad \text{or} \quad \binom{n}{\alpha n} = \tilde{\Theta}\left(2^{nH(\alpha)}\right)$$

Probability laws. For a finite set E, $e \in_R E$ refers to drawing uniformly at random an element e from E. The PMF of a random variable denotes its probability mass function.

Let X be a random variable, $p \in [0, 1]$, and $n \in \mathbb{N}$. $X \sim \text{Bernoulli}(p)$ signifies that X takes the value 1 with probability p and the value 0 with probability $1 - p$.
$X := (X_1, \ldots, X_n) \sim \text{Bernoulli}(p, n)$ means that the X_i are independent and identically distributed with $X_i \sim \text{Bernoulli}(p)$, for $i \in [1, n]$. $X \sim \text{Binomial}(p, n)$ means that X follows the Binomial distribution with PMF: $\Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$, $k \in [0, n]$. Finally, if $X \sim \text{Bernoulli}(p, n)$, then the random variable Y corresponding to the number of successes of X follows the binomial distribution, i.e. $Y := \text{weight}(X) \sim \text{Binomial}(p, n)$.

2.2 Random Functions

Birthday paradox. Let E be a finite set of n elements. If elements are sampled uniformly at random from E, then the expected number of samples to be taken before some element is sampled twice is less than $\sqrt{\pi n/2} = \Theta(\sqrt{n})$. The element that is sampled twice is called a **collision** . See [11] for the details.

Expected number of collisions. Let $f : E \to F$ be a random function. We are interested in the expected number of collisions of f, i.e. the number of distinct pairs $\{x, y\}$ with $f(x) = f(y)$. For instance, if k elements have the same value, this counts as $\binom{k}{2}$ collisions.

Fact 1 *Let $f : E \to F$ be a random function, with $|E| = n$ and $|F| = m$. The expected number of f collisions is $\Theta\left(\frac{n^2}{2m}\right)$.*

Proof. For each pair $\{x, y\}$ $(x \neq y)$, we define the following indicator random variable:

$$I_{\{x,y\}} = \begin{cases} 1 & \text{if } f(x) = f(y) \\ 0 & \text{otherwise} \end{cases}$$

Let C denote the number of collisions of f. The expectation $E(C)$ of C is given by:

$$E(C) = \sum_{\{x,y\}\in E\times E, x\neq y} E(I_{\{x,y\}}) = \frac{1}{m} \sum_{\{x,y\}\in E\times E, x\neq y} 1 = \frac{1}{m}\binom{n}{2} = \Theta\left(\frac{n^2}{2m}\right)$$

□

Collision-finding algorithms Let $f : E \to F$, with $F \subseteq E$, be a random function. According to the birthday paradox, a collision of f can be found in roughly $\Theta(\sqrt{|F|})$ evaluations. Common search algorithms, e.g. Brent's cycle-finding algorithm [4], achieve this by computing a chain of invocations of f from a random starting point s until a collision occurs. In the text, the notation $(x, y) \longleftarrow \mathrm{Rho}(f, s)$ refers to the collision (x, y) returned by f from starting point s, using a cycle-finding algorithm.

In [22], van Oorschot and Wiener extend this idea to search collisions between two functions f_1 and f_2 (both have the same domain E and range F, with $F \subseteq E$). The construction defines a new function f that alternates between f_1 and f_2 depending on the input. The new function f is a random function, thus any cycle-finding algorithm applies and finds a collision for the new function in $\Theta(\sqrt{|F|})$ and polynomial memory. The found collision is a collision between f_1 and f_2 with probability $\frac{1}{2}$. Therefore the running time will roughly double if collisions are random. This is achieved by randomizing the output of the algorithm. In fact, Brent's cycle-finding algorithm is likely to produce always the same collision. To remediate this problem, [1,10] consider a family of permutations $(P_k)_{k\in\mathbb{N}}$ in E addressed by k: they apply the collision-finding algorithm to $g\colon E \to E$ with $g(x) = P_k(f(x))$, where P_k is a random permutation from the considered family. I.e., a new permutation is used with each invocation of the collision-finding algorithm, which ensures that the produced collisions are uniformly distributed.

3 First Algorithm

Let P be a d-degree polynomial over $\mathbb{F}_2$ with nonzero constant term, and $n > d$ be an integer. Our goal is to compute a multiple of P with the least possible weight, and with nonzero constant term and degree at most n. We proceed as follows.

We first determine the minimal weight using Inequality 1. Let ω be the found weight, and $1 + z = 1 + \sum_{i=1}^{n} z_i X^i$ be a weight-ω solution to the LWPM problem. We decompose z to $z = x + y$, with $x, y \in (\mathbb{F}_2^n, +)$ and $\mathrm{weight}(x) = \mathrm{weight}(y) = \phi = n * \phi_n$, where ϕ is a weight to be determined as a function of ω. Then, we compute x and y as a collision to a random function f, using any collision-finding algorithm, e.g. [4].
To compute ϕ, we assume and put in place a Bernoulli distribution on x and y. That is, we ensure the coordinates (of x and y) are independent and equal to 1 with the constant probability $\phi_n = \phi/n$.

This section is organized as follows. Subsection 3.1 defines the building blocks that will be used in the algorithm, namely the weight ϕ, the random function f and a further function that puts in place the Bernoulli distribution. Subsection 3.2 describes our first algorithm for solving LWPM. Finally Subsections 3.3 and 3.4 are dedicated respectively to the analysis and experimental validation of the presented algorithm.

3.1 Building blocks

Computation of ϕ. Assume a Bernoulli distribution on x and y. I.e. the coordinates of both x and y are considered independent trials with the constant probability of success $\Pr(x_i = 1) = \Pr(y_i = 1) = \phi_n = \frac{\phi}{n}$ for $i \in [1, n]$.

Therefore $z = x + y$ follows also a Bernoulli law with PMF $\Pr(z_i = 1) = 2\phi_n(1 - \phi_n)$, for $i \in [1, n]$. Moreover weight$(z) \sim$ Binomial$(2\phi_n(1 - \phi_n), n)$. Since weight$(z) = \omega - 1$, thus $\omega - 1 = 2n\phi_n(1 - \phi_n)$, which is equivalent to $\phi_n = \frac{1}{2}(1 \pm \sqrt{1 - 2\omega_n})$, where $\omega_n := \frac{\omega - 1}{n}$. Note that we assumed $\omega \leq \frac{n}{2}$, thus $\omega_n \leq \frac{1}{2}$.

Random function f. Let ϕ and ϕ_n be the quantities computed in the previous paragraph. Define the set $\mathcal{T}$:

$$\mathcal{T} = \left\{x \in \{0,1\}^n \colon \text{weight}(x) = \phi = n * \phi_n\right\} \tag{2}$$

Let further $a_i = X^i \bmod P$ for $i \in [0, n]$. Consider the functions f_0, f_1:

$$\begin{aligned} f_0, f_1 &\colon \mathcal{T} \longrightarrow \mathbb{F}_2^d \\ f_0(x) = \sum_{i=1}^{n} x_i a_i \ \text{ and } \ f_1(x) &= a_0 + \sum_{i=1}^{n} x_i a_i \end{aligned} \tag{3}$$

Define further the function f:

$$\begin{aligned} f &\colon \mathcal{T} \longrightarrow \mathbb{F}_2^d \\ x &\longmapsto \begin{cases} f_0(x) & \text{if } h(x) = 0 \\ f_1(x) & \text{if } h(x) = 1 \end{cases} \end{aligned} \tag{4}$$

where $h \colon \{0,1\}^n \to \{0,1\}$ is a random bit function. In other terms, f alternates between applications of f_0 and f_1 depending on the input. It is clear that a collision (x, y) of the function f will lead to a multiple of P with expected weight less than ω. In fact, a collision of type $f_i(x) = f_i(y)$, $i = 0, 1$ gives a multiple with expected weight $\omega - 1$, and a collision of type $f_i(x) = f_{1-i}(y)$, $i = 0, 1$ gives a multiple with expected weight ω.

Finally, since we will use a cycle-finding algorithm to search collisions of f, we need the function range and domain to be the same. To achieve this, we consider an injective map $\tau \colon \mathbb{F}_2^d \longrightarrow \mathcal{T}$ (provided $2^d \leq |\mathcal{T}|$). Therefore, all collisions (x, y) of f satisfy

$$f(x) = f(y) \iff \tau \circ f(x) = \tau \circ f(y)$$

In this way, any cycle-finding technique can be applied to $\tau \circ f$ to search for collisions of f.
In the rest of the text, we conveniently identify $\tau \circ f$ with f; that is we assume that f outputs elements in $\mathcal{T}$, provided that $2^d \leq |\mathcal{T}|$, but we keep in mind that $|f(\mathcal{T})| = 2^d$.

Bernoulli distribution on the input of f. Recall that function f inputs vectors of $\mathcal{T}$ that follow a Bernoulli distribution with parameters ϕ_n and n. That is, coordinates of the input vectors are independent and identically distributed with the constant probability ϕ_n of being equal to one. With this assumption, a collision of f leads to a multiple of P with expected weight less than ω. We achieve such a distribution by using a random function σ

$$\begin{aligned} \sigma \colon \{0,1\}^n &\longrightarrow \{0,1\}^n \\ x &\longmapsto \sigma(x) \colon \sigma(x) \sim \text{Bernoulli}(\phi_n, n) \end{aligned}$$

More precisely, σ uses the input elements as a seed to produce n-bit vectors that satisfy the Bernoulli distribution. Therefore, the input elements are only used to "remember" the state of the function, so that when it is called with the same value, it produces the same output.
Note that σ outputs elements with weight ϕ with non-negligible probability:

$$\Pr[\sigma(x) \in \mathcal{T}, x \in_R \{0,1\}^n] = \binom{n}{\phi} \phi_n^{\phi}(1-\phi_n)^{n-\phi} = \binom{n}{n\phi_n} 2^{-nH(\phi_n)} \approx \frac{1}{\sqrt{2\pi n\phi_n(1-\phi_n)}}$$

On other note, σ induces a uniform distribution on $\mathcal{T}$. In fact, let $y \in \mathcal{T}$ be a given element in $\mathcal{T}$, and x a random input element to σ

$$\Pr[\sigma(x) = y \mid \sigma(x) \in \mathcal{T}] = \frac{\Pr[\sigma(x) = y, \sigma(x) \in \mathcal{T}]}{\Pr[\sigma(x) \in \mathcal{T}]} = \frac{\phi_n^{\phi}(1-\phi_n)^{n-\phi}}{\binom{n}{\phi}\, \phi_n^{\phi}(1-\phi_n)^{n-\phi}} = \frac{1}{|\mathcal{T}|}$$

Therefore, we conveniently assume in the rest of this section that σ has range $\mathcal{T}$ on which it induces a uniform probability distribution.

3.2 The algorithm

Consider the following map:

$$\begin{aligned} g\colon \{0,1\}^n &\longrightarrow \mathcal{T}(\subset \{0,1\}^n) \\ x &\longmapsto f \circ \sigma(x) \end{aligned}$$

g is well defined as we assumed that σ has range $\mathcal{T}$. Moreover, g is a random function from $\{0,1\}^n$ to $\{0,1\}^n$, and thus we can apply any cycle-finding algorithm to search collisions for g. Note that σ will introduce some unnecessary collisions as we are only interested in collisions of f. We explain later how we compute this fraction of "useful" collisions among the total number of g collisions.

Now therefore, in consideration of the foregoing, a cycle-finding algorithm for g picks a random starting point $s \in_R \{0,1\}^n$, then computes a chain of invocations of g, i.e. $g(s), g^2(s) := g \circ g(s), \ldots$ until finding a repetition. If such a repetition leads to a valid collision (x, y), i.e. $g(x) = g(y)$ and $x \neq y$, return it otherwise start again with a new starting point. Termination of the algorithm is guaranteed if the execution paths from different starting points are independent. In other words, a random collision should be returned for each new starting point.

To randomize collisions, we introduce our last ingredient, a family of permutations P_k addressed by integer k:

$$P_k\colon \{0,1\}^n \longrightarrow \{0,1\}^n$$

The new function subject to collision search is

$$g^{[k]} = g \circ P_k\colon \mathcal{T} \longrightarrow \mathcal{T}$$

Note that the restriction of P_k to $\mathcal{T}$ is still a permutation from $\mathcal{T}$ to $P_k(\mathcal{T})(\subset \{0,1\}^n)$.

$g^{[k]}$ is a random function, with domain and range $\mathcal{T}$, which satisfies the randomness requirement on the computed collisions. In fact, for each new starting point s, a freshly random element $P_k(s)$ is obtained thanks to P_k (the permutation P_k is picked new with each new starting point), which is then used as a seed to σ to produce a random n-bit vector in $\mathcal{T}$ (with non-negligible probability) that satisfies the Bernoulli distribution. Therefore, execution paths, in cycle-searching algorithms for $g^{[k]}$, from different starting points are independent.
Moreover, (x, y) is a collision for $g^{[k]}$ if and only if $(P_k(x), P_k(y))$ is a collision for g. Therefore, we can apply any cycle-finding algorithm to $g^{[k]}$ to search collisions for g.

We can now describe Algorithm 1 for solving the LWPM problem.

Algorithm 1 for LWPM

Input A polynomial P with degree d, and a bound n
Output A multiple M of P such that $\deg(M) \leq n$ and with the least possible weight.
Compute the expected minimal weight ω by solving Inequality 1
$\omega_n \longleftarrow (\omega - 1)/n$; $\mu \longleftarrow \omega - 1$
repeat
  $\mu_n \longleftarrow \mu/n$; $\mu \longleftarrow \mu + 1$
  $\phi_n \longleftarrow \frac{1}{2}(1 \pm \sqrt{1 - 2 * \mu_n})$; $\phi \longleftarrow n * \phi_n$
until $\binom{n}{\phi} \geq 2^d$ ▹ to ensure that f has range $f(\mathcal{T}) \subseteq \mathcal{T}$
repeat
  choose a random permutation P_k
  choose a random starting point $s \in_R \mathcal{T}$
  $(x, y) \longleftarrow \mathrm{Rho}(g^{[k]}, s)$
  $(p, q) \longleftarrow (\sigma \circ P_k(x), \sigma \circ P_k(y))$
  $M \longleftarrow \begin{cases} X * (p + q) & \text{if } f_i(p) = f_i(q),\ i = 0, 1 \\ 1 + X * (p + q) & \text{if } f_i(p) = f_{1-i}(q),\ i = 0, 1 \end{cases}$
until $M \equiv 0 \bmod P$ and $\mathrm{weight}(M) \in [1, \omega]$
return M

Remark 1. Algorithm 1 finds weight-ω multiples provided they exist. When Inequality 1 predicts a weight that does not exist, the algorithm runs indefinitely. As a safety valve, one can allow a margin in the breaking condition, and accept multiples with weights within that margin.

Remark 2. The μ_n's considered in the first loop are all less than $\frac{1}{2}$. In fact, they satisfy $\mu_n = 2\phi_n(1 - \phi_n)$, and the function $x \longmapsto 2x(1 - x)$ is upper bounded by $\frac{1}{2}$ for $x \in [0, 1]$.

Remark 3. Both the values $\frac{1}{2}(1 + \sqrt{1 - 2\mu_n})$ and $\frac{1}{2}(1 - \sqrt{1 - 2\mu_n})$ for ϕ_n give the same expected time in terms of function calls, however, the latter value finds the solution faster as it is easier to manipulate sparse vectors.

3.3 Complexity analysis

Theorem 1. *Algorithm 1 runs in time $\Theta(2^{C_t})$ with*

$$C_t = \frac{d}{2} + n(-H(w_n) + H_1(\omega_n)) + \frac{3}{2}\log_2(2\pi n\omega_n(1 - \omega_n))$$

where $H_1(\omega_n) = -\omega_n \log_2(2\omega_n(1 - \omega_n)) - (1 - \omega_n)\log_2(1 - 2\omega_n(1 - \omega_n))$.

We first note that $\omega - 1 = \phi$. In fact, ω is the smallest integer such that the inequality $\binom{n}{\omega-1} \geq 2^d$ holds. On other note, ϕ is the smallest integer such that $\binom{n}{\phi} \geq 2^d$, thus $\phi = \omega - 1$ and $\phi_n = \omega_n$.

Moreover, g and thus $g^{[k]}$ induces the uniform distribution on $g^{[k]}(\mathcal{T})$. In fact, σ induces the uniform distribution on $\mathcal{T}$, and f alternates with probability $\frac{1}{2}$ between applications of the deterministic functions f_0 and f_1. Thus, the birthday paradox applies and a collision of $g^{[k]}$ costs on average $2^{d/2}$. Actually, $g^{[k]}$ has domain $\mathcal{T}$ and range $g^{[k]}(\mathcal{T}) \subseteq T$, with $|g^{[k]}(\mathcal{T})| = 2^d$. Also, the expected number of $g^{[k]}$ collisions is $\Theta(\frac{|\mathcal{T}|^2}{2^{d+1}})$ according to Fact 1.

Proof. The algorithm searches collisions (x, y) for $g^{[k]}$ that correspond to f collisions, and that satisfy a weight condition. We call such collisions "useful collisions". Let $(x, y) \in_R \mathcal{T}^2$ with $(p, q) = (\sigma \circ P_k(x), \sigma \circ P_k(y))$. (x, y) is a useful collision for $g^{[k]}$ if the following hold:

Event E_1: "$p, q \in \mathcal{T}$" (so that the function g and thus $g^{[k]}$ is well-defined)

Event E_2**:** "weight$(p+q) = n * \omega_n$"
Event E_3**:** "$X*(p+q)$ or $1+X*(p+q)$ is a multiple of P"

Therefore the number of useful collisions is given by $|\mathcal{T}|^2 * \Pr[E1 \wedge E2 \wedge E_3]$.

According to the previous study of σ, we have $\Pr[E_1] \approx \frac{1}{2\pi n\phi_n(1-\phi_n)}$.

Moreover, $p \sim \text{Bernoulli}(\phi_n, n)$ and $q \sim \text{Bernoulli}(\phi_n, n)$. Therefore $p+q \sim \text{Bernoulli}(2\phi_n(1-\phi_n), n)$, and weight$(p+q) \sim \text{Binomial}(2\phi_n(1-\phi_n), n)$. Thus:

$$\begin{aligned}\Pr[E_2 \mid E_1] \approx \Pr[E_2] &= \binom{n}{n*\omega_n}(2\phi_n(1-\phi_n))^{n*\omega_n}(1-2\phi_n(1-\phi_n))^{n-n*\omega_n}\\ &= \binom{n}{\omega-1}(2\phi_n(1-\phi_n))^{n*\omega_n}(1-2\phi_n(1-\phi_n))^{n-n*\omega_n}\end{aligned}$$

Finally, the probability that a random weight-ω polynomial with nonzero constant term and degree at most n equals a weight-ω multiple of P with nonzero constant term and degree at most n is $\binom{n}{\omega-1}^{-1}\mathcal{N}_M$, where $\mathcal{N}_M$ is the number of such multiples which equals $\binom{n}{\omega-1}2^{-d}$. Similarly, the probability that a random weight-$(\omega-1)$ polynomial with zero constant term and degree at most n equals a weigh-$(\omega-1)$ multiple of P with zero constant term and degree at most n is $\binom{n}{\omega-1}^{-1}\mathcal{N}'_M$, where $\mathcal{N}'_M$ is the number of such multiples which equals $\binom{n}{\omega-1}2^{-d}$. Thus $\Pr[E_3 \mid E_2, E_1] = 2^{-d+1}$.

Since $\phi_n = \omega_n$ ($\phi = \omega - 1$), we conclude that the number of useful collisions is given by

$$\begin{aligned}N_{\text{useful-collisions}} &= |\mathcal{T}|^2 * \Pr[E1 \wedge E2 \wedge E_3]\\ &\approx |\mathcal{T}|^2 2^{-d+1}\binom{n}{\omega-1}(2\phi_n(1-\phi_n))^{n*\omega_n}(1-2\phi_n(1-\phi_n))^{n-n*\omega_n}\frac{1}{2\pi n\phi_n(1-\phi_n)}\\ &= |\mathcal{T}|^3 2^{-d+1}(2\omega_n(1-\omega_n))^{n*\omega_n}(1-2\omega_n(1-\omega_n))^{n-n*\omega_n}\frac{1}{2\pi n\omega_n(1-\omega_n)}\end{aligned}$$

And the probability of a useful collision is:

$$\begin{aligned}\Pr[\text{useful}-\text{coll}] &= \frac{N_{\text{useful-collisions}}}{N_{g_k\text{-collisions}}}\\ &\approx \Theta\left(2^{-2}|\mathcal{T}|(2\omega_n(1-\omega_n))^{n*\omega_n}(1-2\omega_n(1-\omega_n))^{n-n*\omega_n}\frac{1}{2\pi n\omega_n(1-\omega_n)}\right)\\ &= \Theta\left(2^{nH(\omega_n)}(2\omega_n(1-\omega_n))^{n*\omega_n}(1-2\omega_n(1-\omega_n))^{n-n*\omega_n}\frac{1}{(2\pi n\omega_n(1-\omega_n))^{3/2}}\right)\end{aligned}$$

Finally, the running time (in terms of function calls) of the algorithm is the product of $\Pr[\text{useful}-\text{coll}]^{-1}$ and the cost of a $g^{[k]}$-collision, i.e. $2^{d/2}$. Thus, on average, the running time exponent is approximately:

$$C_t = \frac{d}{2} + n(-H(w_n) + H_1(\omega_n)) + \frac{3}{2}\log_2(2\pi n\omega_n(1-\omega_n)$$

where $H_1(\omega_n) = -\omega_n\log_2(2\omega_n(1-\omega_n)) - (1-\omega_n)\log_2(1-2\omega_n(1-\omega_n))$.

□

3.4 Experimental results

We run Algorithm 1 on the following polynomial P for $n \in [30, 1100]$. The results are depicted in Figure 3.

$$P = X^{19} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^3 + X^2 + X^1 + 1$$

Further experiments are deferred to Appendix A.

We used the the $\tilde{\Theta}$ notation for the estimated time, which explains the differences between the estimates and the experiments; the polynomial factor $(2\pi n\omega_n(1-\omega_n))^{\frac{3}{2}}$ is ignored in the estimated time.

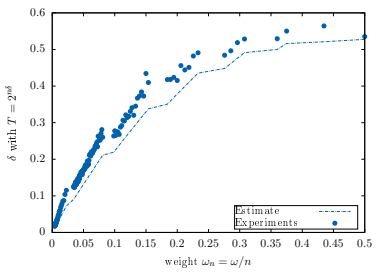


Fig. 3. Averaged function calls T for Algorithm 1 run on Polynomial P

4 Second Algorithm

Algorithm 1 in Section 3 incurs an overhead in the computations due to function σ. Actually, with each invocation of the function f, we make a call to σ which uses a pseudo-random number generator to establish the Bernoulli distribution on the input.

We remediate this problem in this section. Therefore, we decompose the solution z of LWPM into a pair (x, y), where x, y are n-bit vectors that do not enjoy any specific properties except having the same weight ϕ to be determined. We then look for such pairs by searching collisions of f.

Consider the set $\mathcal{T}$ defined in Statement 2, and let $x, y \in_R \mathcal{T}$. We proceed as follows. We first determine the PMF of the random variable $Y = \text{weight}(x + y)$ and compute ϕ as a function of ω. Then, we describe, analyze and experimentally validate our second algorithm in the subsequent subsections.

4.1 Computation of ϕ

Probability law of $Y = \text{weight}(x + y)$

Fact 2 $\Pr[Y = 2k + 1] = 0, \forall k \in \mathbb{N}$.

Proof. $\Pr[Y = 2k + 1]$ denotes the probability that x and y disagree on exactly $2k + 1$ positions. Let $\bar{x}$ and $\bar{y}$ be the $(2k + 1)$-bit strings extracted from x and y respectively, and composed of the bits where x and y disagree. Let further $x \backslash \bar{x}$ and $y \backslash \bar{y}$ be the remaining strings of x and y after extraction of $\bar{x}$ and $\bar{y}$ respectively. We have $\bar{x}_i = 1 - \bar{y}_i$, for $i \in [1, n]$. That is, there are $2k + 1$ ones distributed between the bits of $\bar{x}$ and $\bar{y}$.
Since $\text{weight}(x) = \text{weight}(y) = \phi$. Then, we will have $2\phi - 2k - 1$ ones distributed equally between the bits of $x \backslash \bar{x}$ and $y \backslash \bar{y}$ since $x \backslash \bar{x} = y \backslash \bar{y}$. This is impossible as $2\phi - 2k - 1$ is odd. We conclude that x and y cannot disagree on an odd number of positions. □

Fact 3 $\Pr[Y = k] = 0$, *for* $k \notin [0, \min(2\phi, n)]$.

Proof. There is a total of 2ϕ ones in both x and y. Therefore, x and y can disagree on at most 2ϕ positions. That is $\Pr[Y > 2\phi] = 0$. On other note, it is obvious that $\Pr[Y > n] = \Pr[Y < 0] = 0$. □

Let now, $k \leq \min(\phi, n/2)$ be an integer. $\Pr[Y = 2k]$ is given by the number of strings x and y that disagree on $2k$ positions, divided by the size of the probability space. The number of such strings is given by the product of:

- $\binom{n}{2k}$: the number of ways to choose the positions where x and y disagree.

- $\binom{2k}{k}$: the number of ways to distribute k ones in those $2k$ positions. In fact, let $\bar{x}$ and $\bar{y}$ be the $(2k)$-bit strings extracted from x and y respectively, and composed of the bits where x and y disagree. Then, $\bar{x}$ and $\bar{y}$ have the same weight, namely k, as x and y have the same weight ϕ, and agree on the remaining $n-2k$ positions. Thus, the $2k$ ones must be equally distributed among $\bar{x}$ and $\bar{y}$.
- $\binom{n-2k}{\phi-k}$: the number of ways to choose $(n-2k)$-bit strings with weight $(\phi-k)$. I.e. the number of sub-strings where x and y agree.

The size of the probability space is given by $|\mathcal{T}|^2 = \binom{n}{\phi}^2$. Thus

$$\Pr[Y = 2k, k \leq \min(\phi, n/2)] = \binom{n}{2k}\binom{2k}{k}\binom{n-2k}{\phi-k}\Big/\binom{n}{\phi}^2 = \binom{\phi}{k}\binom{n-\phi}{k}\Big/\binom{n}{\phi}$$

We conclude that:

$$\Pr[\text{weight}(x+y) = 2k] = \begin{cases} \binom{\phi}{k}\binom{n-\phi}{k}\Big/\binom{n}{\phi} & \text{if } 0 \leq k \leq \min(\phi, n/2) \\ 0 & \text{otherwise} \end{cases}$$

Computation of ϕ Note that the PMF of $Y = \text{weight}(x+y)$ is reminiscent of the hypergeometric distribution G given by PMF:

$$\Pr[G = k] = \begin{cases} \binom{t}{k}\binom{n-t}{\phi-k}\Big/\binom{n}{\phi} & \text{if } 0 \leq t, \phi \leq n \text{ and } 0 \leq k \leq \min(\phi, t) \\ 0 & \text{otherwise} \end{cases}$$

and expectation $E(G) = \phi^2/n$. Actually, for $t = \phi$, we get

$$\Pr[G = k] = \begin{cases} \binom{\phi}{k}\binom{n-\phi}{\phi-k}\Big/\binom{n}{\phi} & \text{if } 0 \leq \phi \leq n \text{ and } 0 \leq k \leq \phi \\ 0 & \text{otherwise} \end{cases}$$

Therefore $\Pr[\text{weight}(x+y) = 2k] = \Pr[G = \phi - k]$. We derive the expectation of $Y = \text{weight}(x+y)$ as follows.

$$\begin{aligned} E(Y) &= \sum_{k=0, k=2p}^{2\phi} k \Pr[Y = k] = \sum_{k=0}^{\phi} 2k \Pr[Y = 2k] \\ &= \sum_{k=0}^{\phi} 2k \Pr[G = \phi - k] = 2\sum_{k=0}^{\phi} (\phi - k) \Pr[G = k] \\ &= 2\phi - 2E(G) = 2\phi(1 - \phi/n) \end{aligned}$$

Therefore, if we conserve our previous notations: $\phi = n * \phi_n$, and $\omega - 1 = \omega_n * n$, and solve for ϕ_n the equation $\omega_n * n = 2\phi(1 - \phi/n)$. We get $\phi_n = \frac{1}{2}(1 \pm \sqrt{1 - 2\omega_n})$ $(\omega_n \leq \frac{1}{2})$. Note that we get the same value we found for ϕ in Section 3, when we assumed a Bernoulli distribution on x and y, and consequently a binomial distribution on weight$(x+y)$ ($x + y \sim \text{Bernoulli}(\phi_n(1-\phi_n), n)$ and thus weight$(x+y) \sim \text{Binomial}(2\phi_n(1-\phi_n), n)$). This is not surprising; we know that for increasing n, the hypergeometric law converges to the binomial law.

4.2 The algorithm

Let (P, d, n) be a LWPM instance. We compute the minimal weight ω as usual by solving Inequality 1, then we compute ϕ_n as $\frac{1}{2}(1 \pm \sqrt{1 - 2(\omega - 1)/n})$ and ϕ as $n\phi_n$.
To compute a weight-ω multiple of P with degree less than n, we similarly search for collisions (p, q) of the function f defined earlier, where p and q are n-bit vectors with weight ϕ. There is a small particularity of this algorithm depending on the parity of ω. In fact, collisions of f are of two types:

Type 1 collisions that correspond to $f_i(p) = f_{1-i}(q)$, $i = 0, 1$. These collisions produce multiples of type $1 + X(p + q)$, with weight $1 + 2k$, $1 \leq k \leq \min(\phi, n/2)$.
Type 2 collisions that correspond to $f_i(x) = f_i(y)$, $i = 0, 1$. These collisions produce multiples of type $X(p + q)$, with weight $2k$, $1 \leq k \leq \min(\phi, n/2)$

Therefore, if $\omega = 1 + 2k$, we set $\mu =: \omega - 1$ and $\phi = n\phi_n$, with $\phi_n = \frac{1}{2}(1 \pm \sqrt{1 - 2\mu/n})$. As in Algorithm 1, we ensure that f outputs values in $\mathcal{T}$ (using the injective map $\tau\colon \mathbb{F}_2^d \longrightarrow \mathcal{T}$) by satisfying the condition $|\mathcal{T}| \geq 2^d$, where $|\mathcal{T}| = \binom{n}{\phi}$: we keep increasing μ until the inequality holds. Similarly, if $\omega = 2k$, then we initially set $\mu := \omega$ and keep increasing it until $\binom{n}{\phi} \geq 2^d$, where $\phi = n\phi_n$ and $\phi_n = \frac{1}{2}(1 \pm \sqrt{1 - 2\mu/n})$. We note again that both $\frac{1}{2}(1 + \sqrt{1 - 2\mu/n})$ and $\frac{1}{2}(1 - \sqrt{1 - 2\mu/n})$ lead to the same expected function calls, however, the latter value finds the solution faster as it is easier to manipulate sparse vectors.

Finally, to randomize collisions, it is enough to use any family of permutations $P_k\colon \mathcal{T} \longrightarrow \mathcal{T}$. The collision-finding algorithm is then applied to $f^{[k]} := P_k \circ f$.

We are now ready to give the pseudo-code description of our second algorithm for LWPM in Algorithm 2.

Algorithm 2 for LWPM

Input A polynomial P with degree d, and a bound n
Output A multiple M of P such that $\deg(M) \leq n$ and with the least possible weight.
Compute the expected minimal weight ω by solving Inequality 1
if $\omega\%2 = 1$ **then**
  $\omega_n \longleftarrow (\omega - 1)/n$; $\mu \longleftarrow \omega - 1$
else
  $\omega_n \longleftarrow \omega/n$; $\mu \longleftarrow \omega$
end if
repeat
  $\mu_n \longleftarrow \mu/n$; $\mu \longleftarrow \mu + 1$
  $\phi_n \longleftarrow \frac{1}{2}(1 \pm \sqrt{1 - 2 * \mu_n})$; $\phi \longleftarrow n * \phi_n$
until $\binom{n}{\phi} \geq 2^d$ ▹ to ensure that f has range $f(\mathcal{T}) \subseteq \mathcal{T}$
repeat
  choose a random permutation $P_k\colon \mathcal{T} \longrightarrow \mathcal{T}$
  choose a random starting point $s \in_R \mathcal{T}$
  $(p, q) \longleftarrow \mathrm{Rho}(f^{[k]}, s)$
  $M \longleftarrow \begin{cases} X * (p + q) & \text{if } f_i(p) = f_i(q),\ i = 0, 1 \\ 1 + X * (p + q) & \text{if } f_i(p) = f_{1-i}(q),\ i = 0, 1 \end{cases}$
until $M \equiv 0 \bmod P$ and $\mathrm{weight}(M) \in [1, \omega]$
return M

First, we note that Remarks 1 & 2 & 3 for Algorithm 1 apply also here. Moreover, for even ω, Algorithm 2 finds multiples of the form $X * (p + q)$, where $p + q$ is a polynomial with degree at most $n - 1$. That is, the algorithm finds a weight-ω multiple with nonzero constant term and degree at most $n - 1$ (since P has nonzero constant term) provided it exists. One could change,

in this case, the definition of $\mathcal{T}$ and f and manipulate $(n+1)$-bit vectors instead of n-bit vectors in order to find multiples of degree at most n, but we opted for the above description to keep the algorithm simple.

4.3 Complexity analysis

Let $p, q \in_R \mathcal{T}$ and $j, \omega \in [1, n]$. Define the following events:

Event W**:** "weight$(p+q) = \omega$"
Event P_j**:** "$(p+q)_{1\ldots j} = \underbrace{0\ldots0}_{\text{j-1}}1$", where $(x)_{1\ldots j}$ denotes the length-j prefix of vector x.

Actually, when ω is even, then Algorithm 2 computes the solution as a **Type 2 collision** (p, q), i.e. produces multiples of the form $X(p+q)$. As we are interested in multiples with nonzero constant term, we need to measure the probability of the event $W \wedge P_j$. Thus the necessity of the following fact.

Fact 4 *Let ω be an* even *weight in* $[1, n]$. *Then* $\Pr[W \wedge P_j] = \dfrac{\omega}{n-j+1}\Pr[W]\displaystyle\prod_{l=0}^{j-2}\left(1 - \frac{\omega}{n-l}\right)$. *Moreover, for small ω and i, with $j \le i \le n$:* $\displaystyle\sum_{j=1}^{i}\Pr[W \wedge P_j] \ge i\Pr[W]\frac{\omega}{n-i+1}\left(\frac{n}{n-i+1}\right)^{\omega}$.

Proof. Let $\overline{P_j}$ denote the event "$(p+q)_{1\ldots j} = 0\ldots0$". We prove by induction that $\Pr[W \wedge \overline{P_j}] = \Pr[W]\displaystyle\prod_{l=0}^{j-1}\left(1 - \frac{\omega}{n-l}\right)$.

For $j = 1$:

$$\begin{aligned}
\Pr[W \wedge (p+q)_1 = 0] &= \Pr[p_1 = q_1 = 0]\Pr[W \mid p_1 = q_1 = 0] + \Pr[p_1 = q_1 = 1]\Pr[W \mid p_1 = q_1 = 1] \\
&= \left(\frac{\binom{n-1}{\phi}}{\binom{n}{\phi}}\right)^2 \Pr[W \mid p_1 = q_1 = 0] + \left(\frac{\binom{n-1}{\phi-1}}{\binom{n}{\phi}}\right)^2 \Pr[W \mid p_1 = q_1 = 1] \\
&= (1-\phi_n)^2 \Pr[\text{weight}(p'+q') = \omega] + {\phi_n}^2 \Pr[\text{weight}(p''+q'') = \omega]
\end{aligned}$$

Where, p', q' are random $(n-1)$-bit vectors with weight(p') = weight$(p') = \phi$, and p'', q'' are random $(n-1)$-bit vectors with weight(p'') = weight$(p'') = \phi - 1$. Using the PMF of weight$(p+q)$, we compute $\Pr[\text{weight}(p'+q') = \omega]$ and $\Pr[\text{weight}(p''+q'') = \omega]$, and find that the expression of $\Pr[W \wedge (p+q)_1 = 0]$ simplifies to $\Pr[W]\left(1 - \frac{\omega}{n}\right)$.

Let now $j \ge 1$, and suppose the result holds true until j. We have

$$\Pr[W \wedge \overline{P_{j+1}}] = \Pr[W \wedge \overline{P_j} \wedge (p+q)_{j+1} = 0]$$

The event "$W \wedge \overline{P_j}$" is equivalent to the event W': weight$(p'+q') = \omega$", where p', q' are $(n-j)$-bit vectors such that weight(p') = weight$(q') = \phi_j$ with ϕ_j taking values in the interval $[\phi - j, \phi]$. Therefore:

$$\begin{aligned}
\Pr[W \wedge \overline{P_{j+1}}] &= \Pr[W' \wedge (p'+q')_1 = 0] \\
&= \left(1 - \frac{\omega}{n-j}\right)\Pr[W'] = \left(1 - \frac{\omega}{n-j}\right)\Pr[W \wedge \overline{P_j}] \\
&= \Pr[W]\prod_{l=0}^{j}\left(1 - \frac{\omega}{n-l}\right)
\end{aligned}$$

Since $\Pr[W \wedge P_j] = \Pr[W \wedge \overline{P_{j-1}}] - \Pr[W \wedge \overline{P_j}]$, then $\Pr[W \wedge P_j] = \frac{\omega}{n-j+1} \Pr[W] \prod_{l=0}^{j-2} \left(1 - \frac{\omega}{n-l}\right)$.

On the other hand, for small ω and i such that $j \leq i \leq n$, we have

$$\begin{aligned}
\log_2\left(\sum_{j=1}^{i} \Pr[W \wedge P_j]\right) &= \log_2\left(\sum_{j=1}^{i} \frac{\omega}{n-j+1} \Pr[W] \prod_{l=0}^{j-2}\left(1 - \frac{\omega}{n-l}\right)\right) \\
&\geq \log_2\left(i \Pr[W] \frac{\omega}{n-i+1} \prod_{l=0}^{i-2}\left(1 - \frac{\omega}{n-l}\right)\right) \\
&= \log_2\left(i \Pr[W] \frac{\omega}{n-i+1}\right) + \sum_{l=0}^{i-2} \log_2\left(1 - \frac{\omega}{n-l}\right) \\
&\approx \log_2\left(i \Pr[W] \frac{\omega}{n-i+1}\right) + \sum_{l=0}^{i-2} \frac{\omega}{n-l} \\
&\approx \log_2\left(i \Pr[W] \frac{\omega}{n-i+1}\right) + \omega\left(\log_2(n) - \log_2(n-i+1)\right)
\end{aligned}$$

The last equation is due to the approximation of the harmonic series $\sum_{k=1}^{n} \frac{1}{k} \approx \ln(n)$.

Finally: $\sum_{j=1}^{i} \Pr[W \wedge P_j] \geq i \Pr[W] \frac{\omega}{n-i+1} \left(\frac{n}{n-i+1}\right)^{\omega}$.

Theorem 2. *Algorithm 2 runs in time $\tilde{\Theta}(2^{C_t})$ where $C_t = \frac{d}{2} + n\left(-H_2(\omega_n) + H(\omega_n)\right)$, with $H_2(\omega_n) = \omega_n + (1-\omega_n) H\left(\frac{\omega_n}{2(1-\omega_n)}\right)$.*

Proof. The algorithm searches for two types of f-collisions: **Type 1 collisions** when ω is odd, and **Type 2 collisions** when ω is even. We detail below the cost of each collision.

Type 1 collisions. A Type 1 collision (p, q) satisfies for an odd ω (i) weight$(p+q) = \omega - 1$ and (ii) $1 + X * (p+q)$ is a weight-ω multiple of P.
Define the following events for a pair $(p, q) \in_R \mathcal{T}^2$: W: "weight$(p+q) = \omega - 1$" and M: "$f \mid 1 + X * (p+q)$".
According to the probability law of weight$(p+q)$, we have

$$\begin{aligned}
\Pr[W] &= \binom{\phi}{(\omega-1)/2}\binom{n-\phi}{(\omega-1)/2}\Big/\binom{n}{\phi} = \binom{\omega-1}{(\omega-1)/2}\binom{n-\omega+1}{(\omega-1)/2}\Big/\binom{n}{\omega-1} \\
&\approx 2^{n\left(\omega_n + (1-\omega_n)H\left(\frac{\omega_n}{2(1-\omega_n)}\right) - H(\omega_n)\right)} \frac{4(1-\omega_n)}{\sqrt{2\pi n \omega_n (2-3\omega_n)}}
\end{aligned}$$

In fact $\phi = \omega - 1$ (and thus $\phi_n = \omega_n$) since ϕ and $\omega - 1$ are the smallest integers that satisfy the inequality $\binom{n}{x} \geq 2^d$.

Further, and as argued previously, the probability that a random weight-ω polynomial with nonzero constant term and degree at most n equals a weight-ω multiple of P with nonzero constant term and degree at most n is $\binom{n}{\omega-1}^{-1} \mathcal{N}_M$, where $\mathcal{N}_M$ is the number of such multiples which equals $\binom{n}{\omega-1} 2^{-d}$.
Therefore, for a pair $(p, q) \in_R \mathcal{T}^2$ and an odd ω

$$\Pr[(p,q) \text{ is a Type 1 collision}] = \Pr[W \wedge M] = \Pr[W] \Pr[M \mid W] = 2^{-d} \Pr[W]$$

This implies that we have heuristically $N_{\text{Type1-collisions}} = |\mathcal{T}|^2 2^{-d} \Pr[W]$ many Type 1 collisions. The probability $p_{\text{type1-collisions}}$ of finding such collisions is given by the ratio of $N_{\text{Type1-collisions}}$

and the total number of f collisions, estimated by $|\mathcal{T}|^2 2^{-d-1}$,

$$p_{\text{type1-collisions}} = \frac{|\mathcal{T}|^2 2^{-d} \Pr[W]}{|\mathcal{T}|^2 2^{-d-1}}$$
$$\approx \Theta\left(2^{n\left(\omega_n + (1-\omega_n)H\left(\frac{\omega_n}{2(1-\omega_n)}\right) - H(\omega_n)\right)} \frac{8(1-\omega_n)}{\sqrt{2\pi n \omega_n (2-3\omega_n)}}\right)$$

Each collision costs $\Theta(2^{d/2})$, therefore, the expected number of function calls before the algorithm terminates is $\Theta(2^{C_t}\text{Poly}_1(n))$:

$$C_t = \frac{d}{2} + n\left(-\omega_n - (1-\omega_n)H\left(\frac{\omega_n}{2(1-\omega_n)}\right) + H(\omega_n)\right) \text{ and } \text{Poly}_1(n) = \frac{\sqrt{2\pi n \omega_n (2-3\omega_n)}}{8(1-\omega_n)}$$

Type 2 collisions. When ω is even, the algorithm produces a Type 2 collision (p, q), characterized by: (i)weight$(p+q) = \omega$, (ii) $(p+q)_{1\ldots j} = 0\ldots01$, where $1 \le j \le i$ and i is the largest integer such that there exists a weight-ω multiple of P with nonzero constant term and degree $n-i$, and (iii) $X(p+q)$ is a weight-ω multiple of P of degree at most $n-1$.
For a pair $(p, q) \in_R \mathcal{T}^2$, consider the events W and P_j defined earlier in this subsection, in addition to the event M: "$f \mid X * (p+q)$". Therefore

$$\Pr[(p,q) \text{ is a Type 2 collision}] = \sum_{j=1}^{i} \Pr[W \wedge P_j \wedge M] = \sum_{j=1}^{i} \Pr[W \wedge P_j] \Pr[M \mid W, P_j]$$

Again, the probability that a random weight-ω polynomial with nonzero constant term and degree $n-i$ equals a weight-ω multiple of P with nonzero constant term and degree $n-i$ is $\binom{n-i}{\omega-1}^{-1} \mathcal{N}'_M$, where $\mathcal{N}'_M$ is the number of such multiples which equals $\binom{n-i}{\omega-1}2^{-d}$. Therefore $\Pr[M \mid W, P_j] = 2^{-d}$ for $j \in [1, i]$. Furthermore, according to Fact 4, we have:

$$\Pr[(p,q) \text{ is a Type 2 collision}] = 2^{-d} \sum_{j=1}^{i} \Pr[W \wedge P_j] \quad \ge 2^{-d} i \Pr[W] \frac{\omega}{n-i+1} \left(\frac{n}{n-i+1}\right)^{\omega}$$

With

$$\Pr[W] = \binom{\phi}{\omega/2}\binom{n-\phi}{\omega/2} \Big/ \binom{n}{\phi} = \binom{\omega-1}{\omega/2}\binom{n-\omega+1}{\omega/2} \Big/ \binom{n}{\omega-1}$$

Using $\binom{n-1}{k} = \frac{n-k}{n}\binom{n}{k}$ and $\binom{n}{k-1} = \binom{n}{k}\frac{k}{n-k+1}$, we get:

$$\Pr[W] \approx \frac{(n-\omega+1)^2}{\omega(2n-3\omega+2)} \cdot 2^{n\left(\omega_n + (1-\omega_n)H\left(\frac{\omega_n}{2(1-\omega_n)}\right) - H(\omega_n)\right)} \cdot \frac{4(1-\omega_n)}{\sqrt{2\pi n \omega_n (2-3\omega_n)}}$$

By proceeding in the same way as for Type 1 collisions, we show that Algorithm 2 produces Type 2 collisions in $\Theta(2^{C_t}\text{Poly}_2(n))$:

$$C_t = \frac{d}{2} + n\left(-H_2(\omega_n) + H(\omega_n)\right) \text{ with } H_2(\omega_n) = \omega_n + (1-\omega_n)H\left(\frac{\omega_n}{2(1-\omega_n)}\right)$$

and

$$\text{Poly}_2(n) = \frac{(2n-3\omega+2)}{(n-\omega+1)^2} \cdot \frac{\sqrt{2\pi n \omega_n (2-3\omega_n)}}{8(1-\omega_n)} \frac{n-i+1}{i} \left(\frac{n-i+1}{n}\right)^{\omega}$$

Note that $\left(\frac{n-i+1}{n}\right)^{\omega} \le 1$, thus $\text{Poly}_2(n)$ is indeed polynomial in n. □

4.4 Experimental results

We consider the same test polynomial in Subsection 3.4 for the same range of values $n \in [30, 1100]$; the results are depicted in Figure 4. Note that we used the $\tilde{\Theta}$ notation for the estimated time, which explains the slight differences between the estimates and the experiments. Further experiments are given in Appendix A.

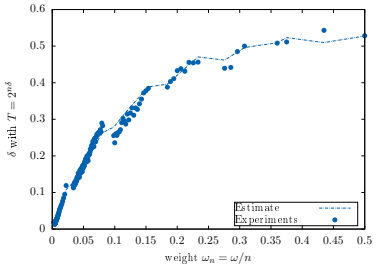


Fig. 4. Averaged function calls T for Algorithm 2

5 Comparison with the State-of-the-art

In this section, we compare the performance of our algorithms with existing memory-efficient methods for LWPM (discrete-log and birthday methods). These lasts run in $\tilde{\Theta}(2^d)$ and $\tilde{\Theta}(2^{nH(\omega_n)})$ respectively. Actually, we discard the lattice method as it becomes inaccurate with increasing n (few hundreds).

Method	Exhaustive search	Discrete Log [19]	Birthday [6]	Algorithm 1	Algorithm 2
$\log_2(\tilde{\Theta}(T))$	$\min(n-d, nH(\omega_n))$	d	$nH(\omega_n)$	$\frac{d}{2} + n(-H(w_n) + H_1(\omega_n))$	$\frac{d}{2} + n\,(-H_2(\omega_n) + H(\omega_n))$

Table 1. Comparison between the memory-efficient techniques and our algorithms

Figure 5 depicts the performance of our algorithms in comparison with the state-of-the-art methods. Note that our algorithms apply to any polynomial, and do not use any precomputed tables of discrete logarithms, unlike some existing memory-efficient methods (discrete-log-based ones).

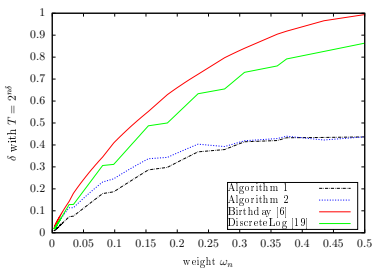


Fig. 5. Comparison between the memory-efficient techniques and our algorithms

Cryptanalytic application I: the Bluetooth summation generator polynomial The Bluetooth polynomial is the product of the four constituent LFSRs feedback polynomials; $P_{BT} = P_1 \cdot P_2 \cdot P_3 \cdot P_4$ where:

$$P_1(x) = x^{25} + x^{20} + x^{12} + x^8 + 1; P_2(x) = x^{31} + x^{24} + x^{16} + x^{12} + 1;$$

$$P_3(x) = x^{33} + x^{28} + x^{24} + x^4 + 1; P_4(x) = x^{39} + x^{36} + x^{28} + x^4 + 1;$$

P_{BT} has degree 128 and weight 49; at degree $n = 668$, the authors in [14] found a multiple of weight $\omega = 31$. Note that the maximum keystream length for the Bluetooth combiner is 2745. That is, the maximum value for the multiple degree is 2745. We note in Table 2 the performances of the different polynomial memory algorithms on this instance.

Method	Exhaustive search	Discrete Log [19]	Birthday method [6]	Algorithm 1	Algorithm 2
$\log_2(\tilde{\Theta}(T))$	181	128	177	76	108

Table 2. Time-complexities of the memory-efficient techniques and our algorithms on the Bluetooth polynomial

Cryptanalytic application II [8] We consider the test polynomial used in [8]

$$P_{53} = X^{53} + X^{47} + X^{45} + X^{44} + X^{42} + X^{40} + X^{39} + X^{38} + X^{36} + X^{33} + X^{32} + X^{31} + X^{30} + X^{28} + X^{27} + X^{26} + X^{25} + X^{21} + X^{20} + X^{17} + X^{16} + X^{15} + X^{13} + X^{11} + X^{10} + X^{7} + X^{6} + X^{3} + X^{2} + X^{1} + 1.$$

The authors in [8] found multiples of weight $\omega = 5$ at degree $n = 2^{13}$. We note in Table 3 the performances of the different polynomial memory algorithms on this instance.

At degree $n \geq 2^{20}$, the authors found multiples with weight $\omega = 4$. However, at this degree, the condition $n \geq 2^{d/(1+\log_2(\omega-1))}$ is satisfied; thus, the generalized birthday method [24] outperforms with a time and a memory cost polynomial in n.

Method	Exhaustive search	Discrete Log [19]	Birthday method [6]	Algorithm 1	Algorithm 2
$\log_2(\tilde{\Theta}(T))$	61	53	50	28	45

Table 3. Time-complexities of the memory-efficient techniques and our algorithms on the [8] instance

6 Time-Memory Trade-off Variants

Our previously described algorithms allow fortunately for a time-memory trade-off, thanks to van Oorschot-Wiener's Parallel Collision Search (PCS) technique [22]. This technique has been extensively used in cryptanalysis since its introduction; it allows to efficiently find multiple collisions, of a random function, at a low amortized cost per collision. More precisely, let C be the time complexity to find a collision with polynomial memory, then PCS finds 2^m collisions in time $\tilde{\Theta}(2^{\frac{m}{2}}C)$ using $\tilde{\Theta}(2^m)$ memory.

In the following, we apply PCS to Algorithms 1 & 2 in order to decrease their time complexity at the expense of memory.

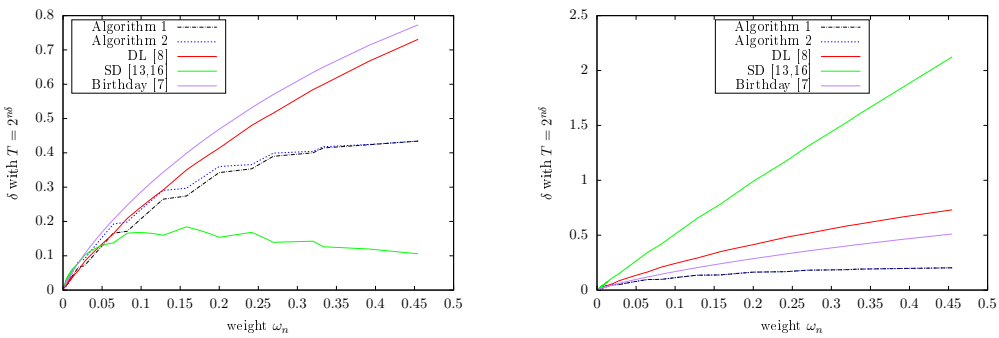
Algorithm 1 Trade-off. According to the analysis in Section 3, Algorithm 1 requires to find $\tilde{\Theta}(2^{n(-H(w_n)+H_1(\omega_n))})$ collisions. In fact, this value corresponds to the number of examined collisions before coming across a so-called *useful collision*, i.e. a collision that leads to a solution to the LWPM problem. Each collision comes at the cost of $\tilde{\Theta}(2^{\frac{d}{2}})$. Therefore, using $M_{\text{tmto-1}} = \tilde{\Theta}(2^{n(-H(w_n)+H_1(\omega_n))})$ memory, the time complexity of the trade-off variant of Algorithm 1 reduces to $T_{\text{tmto-1}} = \tilde{\Theta}(2^{\frac{n(-H(\omega_n)+H_1(\omega_n))}{2}} \cdot 2^{\frac{d}{2}})$.

Algorithm 2 Trade-off. Similarly, Algorithm 2 requires to find $\tilde{\Theta}(2^{n(H(\omega_n)-H_2(\omega_n))})$ collisions, each at the cost of $\tilde{\Theta}(2^{\frac{d}{2}})$. Therefore, using $M_{\text{tmto-2}} = \tilde{\Theta}(2^{n(H(\omega_n)-H_2(\omega_n))})$ memory, the time complexity of the trade-off variant of Algorithm 2 reduces to $T_{\text{tmto-2}} = \tilde{\Theta}(2^{\frac{n(H(\omega_n)-H_2(\omega_n))}{2}} \cdot 2^{\frac{d}{2}})$.

We depict in Table 4 and Figure 6 the time/memory costs of the trade-off variants of Algorithms 1 & 2 and of the state-of-the-art.

	DL [8]	Syndrome Dec [13,16]	Birthday [7]	GBP [1][24]	Algo1	Algo2
$\log_2(T)$	$nH(\frac{\omega-2}{2n})$	$\omega(\log_2 n - \log_2 d)$	$nH(\frac{\omega-1}{2n})$	$\log_2(n)$	$\frac{1}{2}(d - n(H(\omega_n) + H_1(\omega_n)))$	$\frac{1}{2}(d + n(H(\omega_n) - H_2(\omega_n)))$
$\log_2(M)$	$nH(\frac{\omega-2}{2n})$	$\omega \log_2 d$	$nH(\frac{\omega-1}{4n})$	$\log_2(n)$	$n(-H(\omega_n) + H_1(\omega_n))$	$n(H(\omega_n) - H_2(\omega_n))$

Table 4. Comparison between the time-memory trade-off techniques and our algorithms



(a) **Time complexity** (b) **Memory complexity**

Fig. 6. Time/Memory costs of the state-of-the-art and our trade-off algorithms

Our trade-off variants outperform obviously the state-of-the-art in memory, however, they loose the lead in the running time as the weights get smaller. Note however that when the weights get very small, then the generalized birthday method [24] imposes itself as we will see below in Cryptanalytic application II.

Cryptanalytic application I: The Bluetooth Polynomial We note in Table 5 the performances of the time-memory tradeoff methods on the Bluetooth instance considered in Section 5. We discard the generalized birthday method as the condition $n \geq d/(1 + \log_2(\omega - 1))$ is not satisfied.

	DL [8]	Syndrome Dec [13,16]	Birthday [7]	Algo1	Algo2
$\log_2(T)$	101	73	103	70	86
$\log_2(M)$	101	217	59	12	44

Table 5. Time/memory costs of the time-memory trade-off techniques on the Bluetooth polynomial

Cryptanalytic application II [8] We consider the instance polynomial P_{53} defined earlier in Section 5. We provide in Table 6 the performances of the time-memory tradeoff methods on this polynomial at degree $n = 2^{13}$ and weight $\omega = 5$.

[1] provided $n \geq 2^{d/(1+\log_2(\omega-1))}$

It is clear that, despite requiring a big precomputation step and applying only to primitive polynomials with smooth orders, the DL [8] method provides the best time/memory trade-off on this instance. However, we remark that if go up to degree $n = 2^{20}$, we can get a 4-weight multiple using Wagner's generalized birthday, in time and memory polynomial in 2^{20}. This is of course only possible when the available keystream allows it (since n is upper bounded by the available keystream length).

	DL [8]	Syndrome Dec [13,16]	Birthday [7]	Algo1	Algo2
$\log_2(T)$	21	36	27	27	36
$\log_2(M)$	21	29	14	13	19

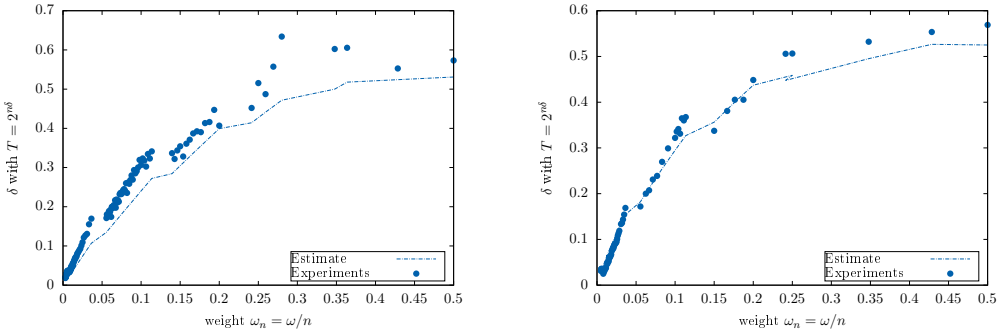
Table 6. Time/memory costs of the time-memory trade-off techniques on the [8] instance

References

1. A. Becker, J-S. Coron, and A. Joux, *Improved Generic Algorithms for Hard Knapsacks*, Advances in Cryptology - EUROCRYPT 2011 -, 2011, pp. 364–385.
2. A. Becker, A. Joux, A. May, and A. Meurer, *Decoding Random Binary Linear Codes in $2^{n/20}$: How 1+1=0 Improves Information Set Decoding*, EUROCRYPT 2012 -, 2012, pp. 520–536.
3. R. P. Brent and P. Zimmermann, *Algorithms for Finding Almost Irreducible and Almost Primitive Trinomials*, Lectures in Honour of the Sixtieth Birthday of Hugh Cowie Williams (2003), 2003.
4. Richard P. Brent, *An Improved Monte Carlo Factorization Algorithm*, BIT Numerical Mathematics (1980), 176184.
5. A. Canteaut and F. Chabaud, *A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511*, IEEE Trans. Inf. Theory (1998), 367–378.
6. A. Canteaut and M. Trabbia, *Improved Fast Correlation Attacks Using Parity-Check Equations of Weight 4 and 5*, Advances in Cryptology - Eurocrypt 2000 (B. Preneel, ed.), LNCS, vol. 1807, Springer, 2000, pp. 573,588.
7. A. Chose, P. Joux and M. Mitton, *Fast Correlation Attacks: An Algorithmic Point of View*, Advances in Cryptology - Eurocrypt 2002 (L. R. Knudsen, ed.), LNCS, vol. 2332, Springer, 2002, pp. 209,221.
8. J. Didier and Y. Laigle-Chapuy, *Finding Low-Weight Polynomial Multiples using Discrete Logarithm*, IEEE International Symposium on Information Theory ISIT'07 (Nice,France), June 2007, p. to appear.
9. L. El Aimani and J. von zur Gathen, *Finding Low Weight Polynomial Multiples Using Lattices* , Poster session of the LLL + 25 conference, 2007, Full version available at the Cryptology ePrint Archive, Report 2007/423.
10. Andre Esser and Alexander May, *Low Weight Discrete Logarithm and Subset Sum in $2^{0.65n}$ with Polynomial Memory*, Advances in Cryptology - EUROCRYPT 2020 -, 2020, pp. 94–122.
11. Steven D. Galbraith, *Mathematics of Public Key Cryptography*, Cambridge University Press, 2012.
12. H. Lenstra, A. Lenstra and L. Lovasz, *Factoring Polynomials With Rational Coefficients*, Mathematische Annalen **261** (1982), no. 4, 515–534.
13. Carl Löndahl and Thomas Johansson, *Improved Algorithms for Finding Low-Weight Polynomial Multiples in $F_2[x]$ And Some Cryptographic Applications*, Des. Codes Cryptogr. (2014), 625–640.
14. Y. Lu and S. Vaudenay, *Faster correlation attack on Bluetooth keystream generator E0*, Advances in Cryptology - Crypto 2004 (M. K. Franklin, ed.), LNCS, vol. 3152, Springer, 2004, pp. 407,425.
15. Alexander May, Alexander Meurer, and Enrico Thomae, *Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$*, Advances in Cryptology - ASIACRYPT 2011 -.
16. Alexander May and Ilya Ozerov, *On Computing Nearest Neighbors with Applications to Decoding of Binary Linear Codes*, Advances in Cryptology - EUROCRYPT 2015, 2015, pp. 203–228.
17. W. Meier and O. Staffelbach, *Fast Correlation Attacks on Certain Stream Ciphers*, J. Cryptology (1989), 159–176.
18. W. T. Penzhorn and G. J. Kühn, *Computation of Low-Weight Parity Checks for Correlation Attacks on Stream Ciphers*, Cryptography and Coding, 1995, pp. 74–83.
19. Pietro Peterlongo, Massimiliano Sala, and Claudia Tinnirello, *A Discrete Logarithm-Based Approach to Compute Low-Weight Multiples of Binary Polynomials*, Finite Fields Their Appl. (2016), 57–71.
20. T. Siegenthaler, *Decrypting a Class of Stream Ciphers Using Ciphertext Only.*, IEEE Trans.Computers **C-34** (1985), no. 1, 81–84.

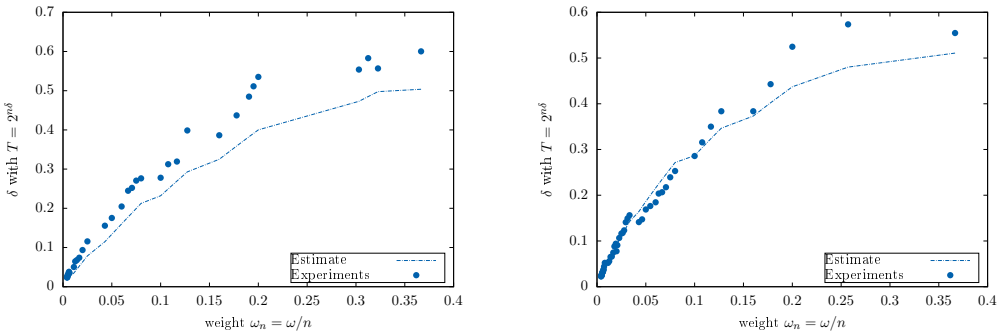
21. Jacques Stern, *A Method for Finding Codewords of Small Weight*, Coding Theory and Application, 1988, pp. 106–113.
22. Paul C. van Oorschot and Michael J. Wiener, *Parallel Collision Search with Cryptanalytic Applications*, J. Cryptol. (1999), 1–28.
23. J. von zur Gathen and M. Nöcker, *Polynomial and Normal Bases for Finite Fields*, J.Cryptology **18** (2005), no. 4, 337–355.
24. D. Wagner, *A Generalized Birthday Problem.*, Advances in Cryptology - Crypto 2002 (M. Yung, ed.), LNCS, vol. 2442, Springer, 2002, pp. 288–304.

A Experiments



(a) **Algorithm 1** (b) **Algorithm 2**

Fig. 7. Averaged function calls T for Algorithms 1 & 2 run on Polynomial:
$P_{17} = X^{17} + X^{15} + X^{14} + X^{13} + X^{11} + X^{10} + X^9 + X^8 + X^6 + X^5 + X^4 + X^2 + 1$



(a) **Algorithm 1** (b) **Algorithm 2**

Fig. 8. Averaged function calls T for Algorithms 1 & 2 run on Polynomial:
$P_{24} = X^{24} + X^{21} + X^{19} + X^{18} + X^{17} + X^{16} + X^{15} + X^{14} + X^{13} + X^{10} + X^9 + X^5 + X^4 + X^1 + 1$