Accelerating the Factorization of Multilinear Boolean Polynomials

Tian Chen[1[0009−0003−0588−4933]] and Michael Monagan[1[0000−0002−4652−2889]]

Simon Fraser University, Burnaby, BC, Canada
tca71@sfu.ca
mmonagan@sfu.ca

Abstract. A multilinear Boolean polynomial f is a polynomial over GF(2) in which each variable has degree at most 1. Such polynomials arise in Boolean circuit optimization, yet their efficient factorization remains challenging. We present two Monte-Carlo algorithms that advance this problem.

Our first factorization algorithm assumes f is given in the sparse representation. It has algebraic complexity $O(n^2t)$ over a suitable extension field GF(2^k), where n is the number of variables and t is the number of terms of f. Our C implementation achieves substantial speedups over both the FDE and GCD algorithms of Emelyanov–Ponomaryov [7].

Our second algorithm assumes f is given by a black box for its evaluation. Here we apply our recently developed black box factorization algorithm CMBBSHL [4,5] which is implemented in Maple and C. The black box representation allows us to reduce the parameter t (the number of terms of the input polynomial f) to $T = s_{\max}\,(\sum \#f_i + \mathfrak{C}(\text{probe } \mathbf{B}))$ (which can be $\ll t$), where the f_i are the irreducible factors of f, $s_{\max}$ is the maximum number of terms in any coefficient (in the highest ranking variable) in any factor, $\sum \#f_i$ is the total number of terms in all irreducible factors, and $\mathfrak{C}(\text{probe } \mathbf{B})$ is the cost of a single black box probe. This yields an overall algebraic complexity $O(n^2T)$. Our Maple and C implementation of our second algorithm is the fastest algorithm when $T \ll t$.

Keywords: Multilinear Polynomials · Boolean Polynomials · Polynomial Factorization · Black Box Algorithms.

1 Introduction

A polynomial f is **multilinear** if it is degree 1 in every variable. For example $f = (x_1+x_2)(x_3x_4+1)$ is multilinear. This restriction implies that the factors of f must have disjoint sets of variables. A polynomial f is a **Boolean** polynomial if all its coefficients are equal to 1. Thus $f = x_1x_3x_4 + x_2x_3x_4 + x_1 + x_2$ is both multilinear and Boolean. Notice that the factorization of a multilinear Boolean polynomial over the fields GF(2) and $\mathbb{Q}$ is the same. One problem setting where multilinear Boolean polynomials arise is Boolean circuit optimization [7].

In 2010 Shpilka and Volkovich [19] showed that factoring polynomials whose factors have distinct variables is polynomial time equivalent to identity testing. In 2018 Emelyanov and Ponomaryov [7] gave a deterministic algorithm called FDE based on such an identity test for factoring a multilinear Boolean polynomial which has bit complexity $O(n^2t^2 \log t)$ where n is the number of variables of f and t is the number of terms of f. Note the bitsize of f is $O(nt)$.

Emelyanov and Ponomaryov [7] gave a second algorithm based on GCD computation. For $f = ax_1 + b$ where $a, b \in \text{GF}(2)[x_2, \ldots, x_n]$, $a \neq 0$, they compute $g = \gcd(a, b)$ which yields the factorization $f = gh$, where $h = f/g$ is irreducible, then they factor g recursively. For the GCD computation, Emelyanov and Ponomaryov assume Zippel's probabilistic algorithm [23] is used for the GCD and state the complexity of Zippel's algorithm at $O(t^3)$ based on the work of de Kleine, Monagan, and Wittkopf [14]. They compared this GCD algorithm with the FDE algorithm and found that the FDE algorithm was somewhat faster.

Note if the GCD computation is done over GF(2), Zippel's algorithm must work in $\text{GF}(2^k)$, an extension of GF(2) for some suitable k so that it does $O(t^3)$ arithmetic operations in $\text{GF}(2^k)$. If the GCD computation is done over $\mathbb{Q}$, Zippel's algorithm can work in $\text{GF}(p)$ for a suitably large prime p so that it does $O(t^3)$ arithmetic operations in $\text{GF}(p)$.

Our first contribution in this work is a Monte Carlo algorithm with algebraic complexity $O(n^2t)$ multiplications in $\text{GF}(2^k)$ for suitable k. We implemented the algorithm in C for $k = 63$ where elements of $\text{GF}(2^k)$ are represented by 64 bit integers and monomials in f are encoded as bit vectors. For ease of use we have also written a parser to read in the polynomial f from a file in a simple text format. Our benchmarks show that our Monte Carlo algorithm is significantly faster than Emelyanov and Ponomaryov's GCD algorithm and FDE algorithm.

Our second contribution concerns the case where f is available through a black box representation. For example, $f = \det(A)$ and A is a matrix with multilinear Boolean polynomial entries. In this setting, we use our recently developed algorithm CMBBSHL [4,5] to factor f over $\mathbb{Z}$, which is equivalent to factoring f over GF(2). Algorithm CMBBSHL is currently the most efficient black box factorization method known, outperforming the seminal algorithm of Rubinfeld and Zippel [18]. It is Monte Carlo with a low failure probability [4]. In this case, the overall algebraic complexity is $O\left(n^2 s_{\max} \left(\sum \#f_i + \mathfrak{C}(\text{probe } \mathbf{B})\right)\right)$, where $s_{\max}$ is the maximum number of terms in any coefficient (in the highest ranking variable) in any factor, $\sum \#f_i$ is the total number of terms in all irreducible factors, and $\mathfrak{C}(\text{probe } \mathbf{B})$ is the cost of a single black box probe. Typically, $s_{\max} \left(\sum \#f_i + \mathfrak{C}(\text{probe } \mathbf{B})\right) < t$, and it may be much smaller.

Our paper is organized as follows. Section 2 presents the GCD algorithm and the FD explicit algorithm of Emelyanov and Ponomaryov from [7], followed by our Monte Carlo algorithm. Section 3 presents our algorithm CMBBSHL for factoring multilinear Boolean polynomials and its complexity analysis. Section 4 presents two timing benchmarks. Section 5 gives some implementation details and an example showing how to use our C code which is available on the

web at `www.cecm.sfu.ca/~mmonagan/code/boolFactor` We end with a short conclusion.

2 A Monte Carlo Algorithm with $O(n^2t)$ Complexity

2.1 Definitions and Preliminaries

Definition 1. *Let F be a field and let f be a polynomial in $F[x_1, \ldots, x_n]$. We denote the total degree of f by* $\deg(f)$ *and the degree of f in x_i by* $\deg(f, x_i)$. *We say f is multilinear if* $\deg(f, x_i) \leq 1$ *for* $1 \leq i \leq n$. *We denote the variables of f by* $\mathrm{var}(f)$.

Example 1. Let $f = (x_2x_3 + x_3x_4)x_1 + (x_2x_3 + x_3x_4)$. Then $\deg(f) = 3$, $\mathrm{var}(f) = \{x_1, x_2, x_3, x_4\}$ and f is multilinear and factors as $f = x_3(x_2 + x_4)(x_1 + 1)$.

Remark 1. If f is multilinear and $f = gh$ for some $g, h \in F[x_1, \ldots, x_n]$, then $\mathrm{var}(g) \cap \mathrm{var}(h) = \phi$ because of the degree constraint.

Definition 2. *Let F be a field and let f be a polynomial in $F[x_1, \ldots, x_n]$. If $f = \sum_{i=1}^{t} a_i M_i(x_1, \ldots, x_n)$ where $a_i \in F$ is non-zero and M_i are distinct monomials in $\{x_1, \ldots, x_n\}$, then the* monomial content *of f is $moncont(f) = \gcd(M_1, \ldots, M_t)$. If $moncont(f) = 1$, we say f is* monomial primitive. *If $f = \sum_{i=0}^{d} a_i(x_1, \ldots, x_n)x_1^i$, the* content *of f w.r.t. x_1 is $\mathrm{cont}(f, x_1) = \gcd(a_0, a_1, \ldots, a_d)$. If $\mathrm{cont}(f, x_1) = 1$, we say f is* primitive *in x_1. We denote the number of terms t of f by $\#f$.*

Example 2. Let $f = x_1x_2x_3 + x_1x_3x_4 + x_2x_3 + x_3x_4 = (x_2x_3 + x_3x_4)x_1 + (x_2x_3 + x_3x_4)$. Then $\#f = 4$, $moncont(f) = x_3$ and $\mathrm{cont}(f, x_1) = x_2x_3 + x_2x_4$.

Definition 3. *Let f be a multilinear polynomial in variables $X = \{x_1, x_2, \ldots, x_n\}$ over $\mathbb{F}_2$ and let $Y \subset X$. Let P be the set of distinct monomials obtained by replacing each variable in $X \setminus Y$ by 1 in f. We define the projection of f onto Y denoted* $\mathrm{proj}(f, Y)$ *is the sum of the terms in P.*

If the monomials in f are ordered in lexicographical order with $x_1 > x_2 > \cdots > x_n$, then $\mathrm{proj}(f, Y) \in O(nt \log t)$ where $t = \#f$. In general the monomials do not remain sorted. We give an example.

Example 3. Consider $f = f_1f_2$ where $f_1 = x_1 + 1$ and $f_2 = x_2 + 1$. Then

$$f = f_1f_2 = x_1x_2 + x_1 + x_2 + 1.$$

Our algorithms will compute one of the factors, say f_1, then divide f by f_1 to get the other factor f_2. Because f is multilinear the division can be done by the projection $\mathrm{proj}(f, \mathrm{var}(f_2)) = \mathrm{proj}(f, \{x_2\})$. Going through the monomials $x_1x_2, x_1, x_2, 1$ in order, the set P of monomials is

$$P = \{x_2, 1, x_2, 1\}$$

which when sorted gives $P = \{x_2, x_2, 1, 1\}$ from which we obtain the factor $x_2 + 1$.

Suppose f is multilinear and $f = gh$ for some $g, h \in \mathbb{F}_2[x_1, ..., x_n]$. Observe that $\text{proj}(f, \text{var}(g)) = g$. Also if we substitute all variables in $X \setminus Y$ by 1 into f, we obtain cg where $c = \#g$ which gives us another way to efficiently compute $\text{proj}(f, \text{var}(g))$.

Example 4. Let $g = x_1x_2 + 1$, $h = x_3x_4 + x_3 + x_5$ and

$$f = gh = x_1x_2x_3x_4 + x_1x_2x_3 + x_1x_2x_5 + x_3x_4 + x_3 + x_5.$$

Then $\text{proj}(f, \text{var}(g)) = \text{proj}(f, \{x_1, x_2\}) = x_1+x_2+1$ since $P = \{x_1x_2, x_1x_2, x_1x_2, 1, 1, 1\} = \{x_1x_2, 1\}$ and $f(x_1, x_2, 1, 1, 1) = x_1x_2 + x_1x_2 + x_1x_2 + 1 + 1 + 1 = 3g$.

2.2 Factorization Algorithms and GCD Algorithms

Let F be a field. To factor a polynomial $f \in F[x_1, \ldots, x_n]$, multivariate factorization algorithms use a multivariate version of Hensel's Lemma. They begin by selecting a main variable, say x_1, and compute and remove the content of f in x_1 and then perform a square-free factorization of f to identify repeated factors of f. However, if f is multilinear, computing the content of f is sufficient to factor f. Suppose $f = a(x_2, \ldots, x_n)x_1 + b(x_2, \ldots, x_n)$. Let $c = \text{cont}(f, x_1) = \gcd(a, b)$. If f is multilinear then f/c is irreducible and, since c is a polynomial in $F[x_2, \ldots, x_n]$, if $c \neq 1$, it can be factored recursively. This yields the following algorithm for factoring multilinear f.

Algorithm 1 GcdFactor

Input: Multilinear $f \in F[x_1, \ldots, x_n]$
Output: Irreducible factors of f

1: **if** $\deg(f) \leq 1$ **return** f **end if**
2: Pick $x \in \text{var}(f)$.
3: Compute $c = \text{cont}(f, x)$.
4: **if** $c = 1$ **return** f **else** compute $p = f/c$ **end if**
5: Let $c_1, \ldots, c_r$ be the output of Algorithm 1 applied recursively to factor c.
6: **return** $(p, c_1, \ldots, c_r)$.

Algorithm 1 reduces factorization of multilinear f over $\mathbb{F}_2$ to multivariate GCD computation over $\mathbb{F}_2$. GCD algorithms for sparse multivariate polynomials over finite fields fall into four general categories, (i) those that use multivariate Hensel lifting e.g. Wang's EEZGCD algorithm from [20], (ii) those that use (sparse) polynomial interpolation e.g. Brown's dense algorithm from [1] and Zippel's sparse algorithm from [23] which is used by Maple, Magma, Mathematica and Fermat, (iii) those that use Kronecker substitutions e.g. Hu and Monagan [11] and Huang and Monagan [12] and (iv) those where the polynomials are represented by black boxes for their evaluations e.g. Diaz and Kaltofen [9].

For polynomials over $F = \mathbb{F}_2$ all cited GCD algorithms need to work over $\text{GF}(2^k)$, a large extension of $\mathbb{F}_2$. Some of these GCD algorithms begin by choosing

a main variable, say y, so for $f = ax+b$, to compute $g = \gcd(a, b)$, they compute $\text{cont}(a, y)$ and $\text{cont}(b, y)$ then $\bar{a} = a/\text{cont}(a, y)$ and $\bar{b} = b/\text{cont}(b, y)$ by division. Then they use $g = \gcd(\text{cont}(a, y), \text{cont}(b, y)) \times \gcd(\bar{a}, \bar{b})$. The first gcd can be done recursively. The second can be done by trial division. If this process is done recursively then these content computations and divisions are sufficient to compute the GCD.

Algorithms which interpolate $g = \gcd(a, b)$ from univariate images require at least t images (Zippel's algorithm needs $O(nt)$ images). To get an image they must evaluate a and b at points in $\text{GF}(2^k)$ which costs at least $O(nt)$ multiplications in $\text{GF}(2^k)$ per image, thus at least $O(n^2t^2)$ multiplications in $\text{GF}(2^k)$.

2.3 FD Explicit Algorithm

In [7], Emelyanov and Ponomaryov present a GCD free algorithm for factoring multilinear f over $\mathbb{F}_2$. In their presentation it is assumed that f is also monomial primitive, that is, trivial factors x_i of f have been identified and removed.

Let $x \in \text{var}(f)$ and $f = cg$ where $c = \text{cont}(f, x)$. Their algorithm is based on the following Theorem that can be used to identify the variables in $\text{var}(c)$ and those in $\text{var}(g)$. We give a proof of the theorem because it forms the basis for our algorithm and the proof shows where an expression swell occurs in the algorithm. First, we recall a theorem about the Sysvester resultant.

Theorem 1. *Let R be a unique factorization domain and $f, g \in R[x]$ have degree at least 1. Let $\text{res}(f, g)$ denote the Sylvester resultant of f and g with respect to x. Then $\deg(\gcd(f, g), x) > 0$ iff $\text{res}(f, g) = 0$.*

Theorem 2. *Let f be non-zero, multilinear, and monomial primitive over $\mathbb{F}_2$ and let $x, y \in \text{var}(f)$ with $x \neq y$. Let $S = f(x = 0)\, \partial f/\partial x$. Then*

$$y \in \text{var}(c) \iff \frac{\partial S}{\partial y} = 0.$$

Proof. Let $f = ax + b$ where $a \neq 0$. Since f is monomial primitive $b \neq 0$ and $\partial b/\partial x = 0$. Let $f = c(\bar{a}x + \bar{b})$ where $c = \text{cont}(f, x) = \gcd(a, b)$. We have

$$S = f(x = 0)\frac{\partial f}{\partial x} = ba = \bar{a}\bar{b}c^2.$$

Case ($y \in \text{var}(c)$). Since f is multilinear, $y \in \text{var}(c)$ implies $\deg(\bar{a}\bar{b}, y) = 0$ thus

$$\frac{\partial S}{\partial y} = (\bar{a}\bar{b})\frac{\partial c^2}{\partial y} = 2(\bar{a}\bar{b})c\frac{\partial c}{\partial y} = 0 \text{ over } \mathbb{F}_2.$$

Case ($y \notin \text{var}(c)$). Let $\bar{a} = Ay + B$ and $\bar{b} = Cy + D$. Then $y \notin \text{var}(c)$ implies

$$\frac{\partial S}{\partial y} = c^2(A(Cy + D) + C(Ay + B)) = c^2(AD + BC).$$

Now $c^2 \neq 0$ so we need to show $AD+BC \neq 0$. Towards a contradiction suppose $AD+BC=0$. We consider four cases: (i) $A \neq 0$ and $B \neq 0$, (ii) $A \neq 0, B = 0$, (iii) $A = 0, B \neq 0$ and (iv) $A = B = 0$. For case (i) we have

$$\begin{aligned} AD+BC=0 \implies AD-BC=0 &\implies \operatorname{res}(Ay+B, Cy+D, y)=0 \\ &\implies \gcd(Ay+B, Cy+D) \neq 1, \end{aligned}$$

a contradiction. For case (ii) we have $A \neq 0, B = 0$ implies $AD = 0$. So either $A = 0$ which implies $\deg(f, y) = 0$ or $D = 0$ in which case $\bar{a} = Ay$ and $\bar{b} = Cy$ which implies $y|f$. Both cases lead to a contradiction. Case (iii) is similar to (ii). For case (iv) $A = B = 0$ implies $\bar{a} = 0$ implies $f = 0$, a contradiction. □

Algorithm 2 FDexplicit below uses Theorem 2 to separate $\operatorname{var}(f)$ into $\operatorname{var}(c)$ and $\operatorname{var}(\bar{a}x+\bar{b})$, two disjoint sets of variables. We can now identify the factors by projecting f onto $\operatorname{var}(c)$ and $\operatorname{var}(\bar{a}x+\bar{b})$.

Algorithm 2 FDexplicit (Emelyanov and Ponomaryov [7])

Input: Non-constant multilinear, monomial-primitive $f \in \mathbb{F}_2[x_1, \ldots, x_n]$
Output: Irreducible factors of f

if $\deg(f) \leq 1$ **return** f **end if**
Pick $x \in \operatorname{var}(f)$.
Compute $S = f(x=0)\,\frac{\partial f}{\partial x}$.
$\Sigma_g \leftarrow \{x\}, \quad \Sigma_c \leftarrow \{\}$
for $y \in \operatorname{var}(f) \setminus \{x\}$ **do**
  Compute $S_y = \frac{\partial S}{\partial y}$.
  if $S_y = 0$ **then** $\Sigma_c \leftarrow \Sigma_c \cup \{y\}$ **else** $\Sigma_g \leftarrow \Sigma_g \cup \{y\}$ **end if**
end for
if $\Sigma_c = \{\}$ **return** f **end if**
$g \leftarrow \operatorname{proj}(f, \Sigma_g), \quad c \leftarrow \operatorname{proj}(f, \Sigma_c)$
Let $c_1, \ldots, c_r$ be the output of Algorithm 2 applied recursively to factor c.
return $(g, c_1, \ldots, c_r)$

In the proof of Theorem 2, the quantity $S = \bar{a}\bar{b}c^2$ where $c = \operatorname{cont}(f, x)$. Since c can be large (c may have $\#f/2$ terms), S may have as many as $\#f^2/4$ terms. Thus multiplying out $f(x=0) \times \partial f/\partial x$ causes an expression swell. Emelyanov and Ponomaryov propose the following. Let $A = f(x=0)$ and $B = \partial f/\partial x$, $C = \partial A/\partial y$ and $D = \partial B/\partial y$. Then

$$\frac{\partial S}{\partial y} = \frac{\partial AB}{\partial y} = AD + BC.$$

Directly multiplying AD and BC still causes the expression swell. Furthermore, if z is a variable in $\operatorname{var}(f)$ distinct from x and y, then $\deg(AB, z)$ will often be 2, thus requiring a more general data structure than a bit vector for storing the terms in A, B, C, D and one must sort the products AD and BC. This means

the bit complexity of computing AD and BC is $O(n\#f^2 \log \#f)$ and the overall cost of Algorithm FDexplicit is $O(n^2\#f^2 \log \#f)$.

In this paper we propose to apply the Schwartz-Zippel Lemma in a large field extension of $\mathbb{F}_2$ to test if $AD - BC = 0$.

2.4 FD using Schwartz-Zippel

Theorem 3. *(Schwartz-Zippel Lemma) Let F be a field and S be a finite subset of F. Let f be a non zero polynomial in $F[x_1, x_2, \ldots, x_n]$. If α is chosen at random from S^n then*

$$\Pr[f(\alpha) = 0] \leq \frac{\deg f}{|S|}.$$

Our modification of Algorithm 2 uses the Schwartz-Zippel Lemma in $F = \mathrm{GF}(2^k)$ to test the identity $AD + BC = 0$. We present the algorithm and then return to the question of what to use for k. The algorithm needs to, in a pre-processing step, pick $\alpha_i \in \mathrm{GF}(2^k)$ at random for $1 \leq i \leq n$.

Algorithm 3 FDSZ

Input: $f \in \mathbb{F}_2[x_1, \ldots, x_n]$ and $\alpha \in \mathrm{GF}(2^k)^n$ such that f is non-zero, multilinear, and monomial primitive
Output: Irreducible factors of f

1: **if** $\deg(f) \leq 1$ **return** f **end if**
2: Pick $x \in \mathrm{var}(f)$.
3: Compute $A = f(x = 0)$ and $B = \frac{\partial f}{\partial x}$. // $f = A + Bx$
4: $\Sigma_g \leftarrow \{x\}$, $\Sigma_c \leftarrow \{\}$
5: **for** $y \in \mathrm{var}(f) \setminus \{x\}$ **do**
6: Compute $C = \frac{\partial A}{\partial y}$ and $D = \frac{\partial B}{\partial y}$.
7: **if** $A(\alpha)\,D(\alpha) = B(\alpha)\,C(\alpha)$ in $\mathrm{GF}(2^k)$ **then** $\Sigma_c \leftarrow \Sigma_c \cup \{y\}$ **else** $\Sigma_g \leftarrow \Sigma_g \cup \{y\}$ **end if**
8: **end for**
9: **if** $\Sigma_c = \{\}$ **return** f **end if**
10: $g \leftarrow \mathrm{proj}(f, \Sigma_g)$, $c \leftarrow \mathrm{proj}(f, \Sigma_c)$
11: Let $c_1, \ldots, c_r$ be the output of Algorithm 3 applied recursively to c with input α.
12: **return** $(g, c_1, \ldots, c_r)$

Our benchmarks will show that 99% of the time in Algorithm 3 is spent doing multiplications in $\mathrm{GF}(2^k)$ in line 7. Although we have not implemented it, the loop in line 5 on the variables in f can easily be parallelized for a parallel speedup factor of $n - 1$.

Theorem 4. *Algorithm 3 does $O(n^2\#f)$ multiplications in $\mathrm{GF}(2^k)$ and additional work of bit complexity $O(n^2\#f)$.*

Proof. Since $\#A + \#B = \#f$ it follows that $\#C + \#D < \#f$ therefore the number of multiplications in $\mathrm{GF}(2^k)$ done in step 7 is $< 2n\#f \in O(n\#f)$. Since the for loop is executed $n - 1$ times this is $O(n^2\#f)$ multiplications in total.

Since the operations $f(x = 0)$, $\partial f/\partial x$, $\partial A/\partial y$, $\partial B/\partial y$, $\partial C/\partial y$, $\partial D/\partial y$, $\text{proj}(f, \Sigma_c)$ and $\text{proj}(f, \Sigma_c)$ can all be done in time linear in the size of f (no sorting is required because after evaluation at $x = 0$, differentiation, and projection, terms remain sorted), that is, in $O(n\#f)$ bit operations, the bit complexity of steps 3, 6 and 10 is $O(n\#f)$. Since step 6 is executed $n-1$ times the total cost of steps 3, 6 and 10 is $O(n^2\#f)$.

Since f is monomial primitive, each factor of f has at least two terms, thus $\#c \leq \#f/2$ and so the cost of all the recursive calls is dominated by the first call. The result follows.

Theorem 5. *The probability that Algorithm 3 outputs an incorrect factorization is* $< n \deg(f)^2\, 2^{-k}$.

Proof. By the Schwartz-Zippel Lemma, if $AD \neq BC$ then

$$\text{Prob}[A(\alpha)D(\alpha) + B(\alpha)C(\alpha) = 0] \leq \frac{\max(\deg(AD), \deg(BC))}{2^k} \leq \frac{\deg(f)^2}{2^k}$$

Let $f = gc$ where $c = \text{cont}(f, x)$. Algorithm 3 can only return an incorrect factorization when $y \in \text{var}(g)$. If $y \in \text{var}(c)$, $AD - BC = 0$ thus no error can occur. Since each variable appears in only one factor of f, then there are only $n-1$ tests which could be wrong in all recursive calls of Algorithm 3. Therefore the probability of failure is bounded by $n \deg(f)^2\, 2^{-k}$. □

To ensure that the probability of error is bounded by 2^{-100} say, we pick $k > 100 + \log_2 n + 2\log_2 \deg(f)$. However, such a choice for k will make the multiplications in $\text{GF}(2^k)$ expensive. Before we implemented Algorithm 3 we thought this would wipe out much of the gain in saving the factor of $O(\#f)$. Instead, in our implementation of Algorithm 3, we choose $k = 63$ so that a multiplication can be done using a sequence of 64 bit hardware bit operations (see Algorithm 5), and we do two tests, i.e., we also choose $\beta \in \text{GF}(2^{63})$ at random and in step 7 test

$$\textbf{if } A(\alpha)D(\alpha) = B(\alpha)C(\alpha) \textbf{ and } A(\beta)D(\beta) = B(\beta)C(\beta)$$

Then the probability of failure is then less than $n^2 \deg(f)^4\, 2^{-126}$.

3 A Black Box Algorithm

In the previous section, a multilinear polynomial $f \in \mathbb{F}_2[x_1, \ldots, x_n]$ is given explicitly. On the other hand, if f can be represented by a black box, it can be factored using our recently developed black box factorization algorithm CMBB-SHL [4,5]. An important application is determinant computation, where one computes $\det(A)$ for a matrix A whose entries are multilinear Boolean polynomials. Typically, the irreducible factors of f contain far fewer terms than f itself. Using the black box representation of f as the input saves the memory required to store the expanded form of f. It also reduces the cost of evaluating f at many points when applying a sparse Hensel lifting algorithm to factor f [3].

Since factorization over $\mathbb{F}_2$ is equivalent to factorization over $\mathbb{Z}$ for multilinear Boolean polynomials, we apply Algorithm CMBBSHL to f over $\mathbb{Z}$. In this section, we give an overview of Algorithm CMBBSHL and a complexity analysis for the case of multilinear Boolean polynomials.

3.1 Sparse and Black Box Representation of a Polynomial

Let us denote $\mathfrak{f}$ for the input polynomial in $\mathbb{Z}[x_1, \ldots, x_n]$.

Definition 4. *(Section 16.6 of [10]) A* sparse representation *of a polynomial* $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$ *consists of a list of coefficients* $c_k \neq 0$, $c_k \in \mathbb{Z}$ *and distinct exponent vectors* $(e_{k_1}, \ldots, e_{k_n}) \in \mathbb{N}^n$ *s.t.* $\mathfrak{f} = \sum_{k=1}^{\#\mathfrak{f}} c_k \cdot x_1^{e_{k_1}} \ldots x_n^{e_{k_n}}$.

The sparse representation is a natural, readable and *explicit* representation. In contrast, the black box representation is *implicit*. Since our black box factorization algorithm is modular, we use a modular black box for $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$.

Definition 5. *A* modular black box representation *of* $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$ *is a computer program* $\mathbf{B} : \mathbb{Z}^n \times \{p\} \to \mathbb{Z}_p$ *that on input* $\boldsymbol{\alpha} \in \mathbb{Z}^n$ *and a prime* p *outputs* $\mathbf{B}(\boldsymbol{\alpha}, p) = \mathfrak{f}(\boldsymbol{\alpha}) \bmod p$.

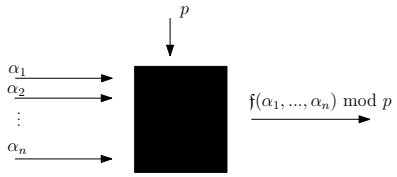

Fig. 1. A modular black box representation of $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$.

The internal structure of the black box $\mathbf{B}$ is inaccessible. We can only call the black box at a given point, $\boldsymbol{\alpha} \in \mathbb{Z}^n$, and obtain its *evaluation*, $\mathbf{B}(\boldsymbol{\alpha}, p)$.

Definition 6. *A single evaluation of the black box* $\mathbf{B} : \mathbb{Z}^n \times \{p\} \to \mathbb{Z}_p$ *at a point* $\boldsymbol{\alpha} \in \mathbb{Z}^n$ *is called a* probe *to* $\mathbf{B}$.

We now aim to solve Problem $\mathcal{P}$:

Problem $\mathcal{P}$: Given a polynomial $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$ represented by a *modular black box* $\mathbf{B} : \mathbb{Z}^n \times \{p\} \to \mathbb{Z}_p$, compute its irreducible factors in $\mathbb{Z}[x_1, \ldots, x_n]$ in their *sparse representation*.

3.2 Algorithm CMBBSHL

First, we observe that multilinear Boolean polynomials are square-free. Consequently, the general Algorithm CMBBSHL for non-monic, non-square-free, and non-primitive polynomials simplifies in this setting to the non-monic and non-primitive case (while remaining square-free). We begin by describing how to

compute the factors of $\text{pp}(\mathfrak{f}, x_1)$, the primitive part of $\mathfrak{f}$ in x_1. Then, $\text{cont}(\mathfrak{f}, x_1)$ is factored recursively. We further observe that, since each factor has a disjoint set of variables, $\text{pp}(\mathfrak{f}, x_1)$ has only one factor. The description of the algorithm below is for the multi-factor case; however, in the complexity analysis we simplify by considering only one factor in $\text{pp}(\mathfrak{f}, x_1)$.

If the input polynomial $\mathfrak{f}$ is in the sparse representation and is non-monic, two established methods exists for pre-computing the leading coefficients of its factors: Wang's *leading coefficient correction* [21] and the approach by Kaltofen [8]. However, for our black box factorization algorithm CMBBSHL, pre-computing leading coefficients is unnecessary. Instead, we use a strategy in which the bivariate images are scaled at each Hensel lifting step so that their leading coefficients match those of the corresponding input factors, as detailed in [4].

We define a Hilbertian point, which we use in our algorithm description.

Definition 7. *Let $P \in \mathbb{Z}[x_1, \ldots, x_n]$ be an irreducible polynomial over $\mathbb{Z}$. A point $\boldsymbol{\alpha} = (\alpha_2, \ldots, \alpha_n) \in \mathbb{Z}^{n-1}$ is called* Hilbertian *if $P(x_1, \alpha_2, \ldots, \alpha_n)$ remains irreducible over $\mathbb{Z}$ and $\deg(P(x_1, \boldsymbol{\alpha})) = \deg(P, x_1)$ [15].*

Remark 2. In practice, non-Hilbertian points are rare. For the sharpest known bound on the number of non-Hilbertian points of $P(x_1, \ldots, x_n)$, we refer the reader to Cohen [6].

The following steps are performed prior to sparse Hensel lifting:

1. Choose a large prime p (e.g. $p = 2^{61} - 1$) and a positive integer $\tilde{N} < p$.
2. Choose an evaluation point $\boldsymbol{\alpha} = (\alpha_2, \ldots, \alpha_n) \in \mathbb{Z}^{n-1}$ randomly from a sufficiently large set $[1, \tilde{N}-1]^{n-1}$ such that $\boldsymbol{\alpha}$ is Hilbertian with high probability.
3. Compute $d_j = \deg(\mathfrak{f}, x_j)$ for all $1 \leq j \leq n$ w.h.p. (For details, see [5])
4. Compute $\mathfrak{f}(x_1, \boldsymbol{\alpha}) \bmod p$ w.h.p. by a univariate dense interpolation. Then, recover $\mathfrak{f}(x_1, \boldsymbol{\alpha}) \in \mathbb{Z}[x_1]$ using Chinese remaindering with different primes.
5. Factor $\mathfrak{f}(x_1, \boldsymbol{\alpha})$ over $\mathbb{Z}$. Let the irreducible factorization of $\mathfrak{f}$ over $\mathbb{Z}$ be

$$\mathfrak{f} = h \prod_{i=1}^{r} f_i \in \mathbb{Z}[x_1, \ldots, x_n],$$

where $\deg(f_i, x_1) > 0$ ($= 1$ for multilinear Boolean polynomials), f_i is irreducible over $\mathbb{Z}$ and primitive in x_1 for $1 \leq i \leq r$, r is the number of factors in $\text{pp}(\mathfrak{f}, x_1)$, $h = \text{cont}(\mathfrak{f}, x_1) \in \mathbb{Z}[x_2, \ldots, x_n]$ and it is not factored at this stage. Let $\lambda_i := \text{cont}(f_i(x_1, \boldsymbol{\alpha})) \in \mathbb{Z}$ for $1 \leq i \leq r$ and let $\lambda_h := \prod_{i=1}^{r} \lambda_i$. Let

$$\begin{aligned} \hat{f}_i &:= \frac{1}{\lambda_i} f_i(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n] \text{ for } 1 \leq i \leq r \text{ and} \\ \hat{h} &:= \lambda_h h(x_2, \ldots, x_n) \in \mathbb{Z}[x_2, \ldots, x_n]. \end{aligned} \tag{1}$$

With high probability, $\boldsymbol{\alpha}$ is Hilbertian. Thus,

$$\begin{aligned}\mathfrak{f}(x_1,\boldsymbol{\alpha}) &= h(\boldsymbol{\alpha})\prod_{i=1}^{r} f_i(x_1,\boldsymbol{\alpha}) = h(\boldsymbol{\alpha})\left(\prod_{i=1}^{r}\lambda_i\hat{f}_i(x_1,\boldsymbol{\alpha})\right) \text{ w.h.p.}\\ &= \underbrace{h(\boldsymbol{\alpha})\prod_{i=1}^{r}\lambda_i}_{\hat{h}(\boldsymbol{\alpha})}\prod_{i=1}^{r}\hat{f}_i(x_1,\boldsymbol{\alpha}) = \hat{h}(\boldsymbol{\alpha})\prod_{i=1}^{r}\hat{f}_i(x_1,\boldsymbol{\alpha}) \in \mathbb{Z}[x_1],\end{aligned}$$

where $\hat{f}_i(x_1,\boldsymbol{\alpha})$ is primitive and irreducible over $\mathbb{Z}$.

Define $\Lambda := \prod_{i=1}^{r}\lambda_i$. By (1),

$$\mathrm{pp}(\mathfrak{f},x_1) = \prod_{i=1}^{r} f_i(x_1,\ldots,x_n) = \Lambda\prod_{i=1}^{r}\hat{f}_i(x_1,\ldots,x_n),$$

where $\hat{f}_i(x_1,\ldots,x_n) \in \mathbb{Q}[x_1,\ldots,x_n]$ if $|\lambda_i| > 1$.

Define $\hat{f}_{i,1} := \hat{f}_i(x_1,\boldsymbol{\alpha}) \bmod p$ for $1 \le i \le r$, and define $\hat{f}_{i,j} := \hat{f}_i(x_1,\ldots, x_j,\alpha_{j+1},\ldots,\alpha_n) \bmod p$ for $2 \le j \le n$. Define $\mathfrak{f}_j := \mathfrak{f}(x_1,\ldots,x_j,\alpha_{j+1},\ldots,\alpha_n) \bmod p$. Algorithm CMBBSHL lifts $(\hat{f}_{i,1})_{i=1}^{r} \in \mathbb{Z}_p[x_1]^r$ to $(\hat{f}_{i,2})_{i=1}^{r} \in \mathbb{Z}_p[x_1,x_2]^r$, then lifts $(\hat{f}_{i,2})_{i=1}^{r} \in \mathbb{Z}_p[x_1,x_2]^r$ to $(\hat{f}_{i,3})_{i=1}^{r} \in \mathbb{Z}_p[x_1,x_2,x_3]^r$, and so on. After the jth Hensel lifting step, $\mathrm{pp}(\mathfrak{f}_j) \equiv \Lambda\prod_{i=1}^{r}\hat{f}_{i,j} \bmod p$ and $\hat{f}_{i,j}(x_j=\alpha_j) = \hat{f}_{i,j-1}$ for all $1 \le i \le r$. After the nth Hensel lifting step, $\mathrm{pp}(\mathfrak{f}_j) \equiv \Lambda\prod_{i=1}^{r}\hat{f}_{i,n} \bmod p$. At this stage, rational number reconstruction [22] is performed on the coefficients of $\hat{f}_{i,n}$ in $\mathbb{Z}_p$ for $1 \le i \le r$ to get the integer coefficients of the factors f_i in $\mathbb{Z}$.

The jth ($j > 2$) Hensel lifting step is presented in Algorithm 4. For $j = 2$, the Hensel lifting step reduces to a bivariate Hensel lift [17]. Each sparse Hensel lifting step consists of the following four main substeps:

1. probes to $\mathbf{B}$ to interpolate the bivariate images $\mathfrak{f}_j(x_1,Y_m,x_j)$ (step 8),
2. evaluations of the factors $\hat{f}_{i,j-1}$ for $1 \le i \le r$ (step 10),
3. bivariate Hensel lifting [17] (step 13),
4. solving Vandermonde systems [24] (step 19).

The bivariate image $\mathfrak{f}_j(x_1,Y_m,x_j)$ is obtained from the black box $\mathbf{B}$ via *bivariate dense interpolation* (i.e., using Lagrange or Newton interpolations in two variables). Step 10 of Algorithm 4 evaluates the input factors to get a univariate image $\hat{f}_{i,j-1}(x_1,Y_m)$. Then at step 13, a non-monic bivariate Hensel lift (BHL) is performed (see [4] for details), so we get a bivariate image of the factors, $\hat{f}_{i,j}(x_1,Y_m,x_j)$. After obtaining s bivariate images of the factors (s is defined in step 5 in Algorithm 4), we use them to recover the variables $x_2,\ldots,x_{j-1}$ in the factor $\hat{f}_{i,j} \in \mathbb{Z}_p[x_1,\ldots,x_j]$ by solving Vandermonde systems at step 19. The verification that $\mathfrak{f}_j = \prod_{i=1}^{r}\hat{f}_{i,j}$ at the end of the jth Hensel lifting step is probabilistic (step 25), thereby avoiding explicit multiplication of the factors.

Algorithm 4 could return FAIL at several steps, i.e. at step 4, 9,11,12, or 25. If p is large, the probability that Algorithm CMBBSHL returns FAIL is low. For an analysis of failure probability, see [4].

Algorithm 4 CMBBSHLstep (non-monic and square-free)

Input: The modular black box $\mathbf{B} : \mathbb{Z}^n \times \{p\} \to \mathbb{Z}_p$ such that $\mathbf{B}(\boldsymbol{\beta}, p) = \mathfrak{f}(\boldsymbol{\beta}) \bmod p$, the list of factors $(\hat{f}_{i,j-1})_{i=1}^r \in \mathbb{Z}_p[x_1, \ldots, x_{j-1}]^r$, a prime p, the evaluation point $\boldsymbol{\alpha} = (\alpha_i)_{i=2}^n \in \mathbb{Z}^{n-1}$, the list of partial degrees $(d_i)_{i=1}^n$ with $d_i = \deg(a, x_i)$, the variable list $X = (x_i)_{i=1}^n$, and an index j with $2 < j \le n$ s.t.
(i) $\operatorname{pp}(\mathfrak{f}_j)(x_j = \alpha_j) \equiv \Lambda \prod_{i=1}^r \hat{f}_{i,j-1} \bmod p$,
(ii) each $\hat{f}_{i,j-1}$ is primitive in x_1.
Output: The list of lifted factors $(\hat{f}_{i,j})_{i=1}^r \in \mathbb{Z}_p[x_1, \ldots, x_j]^r$ s.t.
(i) $\operatorname{pp}(\mathfrak{f}_j) \equiv \Lambda \prod_{i=1}^r \hat{f}_{i,j} \bmod p$,
(ii) $\hat{f}_{i,j}(x_j = \alpha_j) = \hat{f}_{i,j-1}$ for all $1 \le i \le r$,
or FAIL

For each $1 \le i \le r$, let $\hat{f}_{i,j-1} = \sum_{k=0}^{df_i} \sigma_{i,k}(x_2, \ldots, x_{j-1})\, x_1^k$, where $df_i = \deg(\hat{f}_{i,j-1}, x_1)$ and $\sigma_{i,k} = \sum_{l=1}^{s_{i,k}} c_{i,kl}\, M_{i,kl}(x_2, \cdots, x_{j-1})$ with $c_{i,k\ell} \in \mathbb{Z}_p$.
Pick $\boldsymbol{\beta} = (\beta_2, \ldots, \beta_{j-1}) \in (\mathbb{Z}_p \setminus \{0\})^{j-2}$ at random.
For each $0 \le k \le df_i$ and $1 \le i \le r$, evaluate every monomial $M_{i,kl}$ for $1 \le l \le s_{i,k}$ at $\boldsymbol{\beta}$ and set $m_{i,kl} = M_{i,kl}(\boldsymbol{\beta})$. Let $S_{i,k} = \{m_{i,kl} \mid 1 \le l \le s_{i,k}\}$.
if any $|S_{i,k}| \ne s_{i,k}$ **then return** FAIL **end if**
$s \leftarrow \max_{i=1}^r \max_{k=0}^{df_i} s_{i,k}$
for $m = 1$ **to** s **do**
  Let $Y_m = (\beta_2^m, \ldots, \beta_{j-1}^m)$.
  $A_m \leftarrow \mathfrak{f}_j(x_1, Y_m, x_j)$ ..... $\mathcal{O}\big(s\, d_1 d_j \cdot \mathfrak{C}(\text{probe } \mathbf{B}) + s(d_1^2 d_j + d_1 d_j^2)\big)$
  if $\deg(A_m, x_1) \ne d_1$ **or** $\deg(A_m, x_j) \ne d_j$ **then return** FAIL **end if**
  $F_{i,m} \leftarrow \hat{f}_{i,j-1}(x_1, Y_m)$ for $1 \le i \le r$. ..... $\mathcal{O}\big(s \sum_{i=1}^r \#\hat{f}_{i,j-1}\big)$
  if any $\deg(F_{i,m}) < df_i$ **then return** FAIL **end if**
  if $\gcd(F_{i,m}, F_{l,m}) \ne 1$ for some $i \ne l$ **then return** FAIL **end if**
  $(\hat{f}_{i,m})_{i=1}^r \leftarrow \textsc{BivariateHenselLift}\big(A_m, (F_{i,m})_{i=1}^r, \alpha_j, p\big)$ $\mathcal{O}\big(s(d_1^2 d_j + d_1 d_j^2)\big)$
end for
For $1 \le i \le r$ and $1 \le m \le s$, write $\hat{f}_{i,m} = \sum_{\mu=1}^{t_i} \alpha_{i,m\mu}\, \widetilde{M}_{i,\mu}(x_1, x_j)$, $t_i \le d_1 d_j$.
for $i = 1$ **to** r **do**
  for $\mu = 1$ **to** t_i **do**
    Let $k \leftarrow \deg(\widetilde{M}_{i,\mu}, x_1)$.
    Solve for the coefficients $c_{i,\mu l}$ in $\sum_{l=1}^{s_{i,k}} m_{i,kl}^{\tau}\, c_{i,\mu l} = \alpha_{i,\tau\mu}, 1 \le \tau \le s_{i,k}$.
  end for ..... $\mathcal{O}\big(sd_j \sum_{i=1}^r \#\hat{f}_{i,j-1}\big)$
  Construct $\hat{f}_{i,j} \leftarrow \sum_{\mu=1}^{t_i} \Big(\sum_{l=1}^{s_{i,k}} c_{i,\mu l}\, M_{i,kl}(x_2, \ldots, x_{j-1})\Big) \widetilde{M}_{i,\mu}(x_1, x_j)$.
end for
Pick $\boldsymbol{\beta}' = (\beta_2, \ldots, \beta_j)$ at random until $\deg(\hat{f}_{i,j}(x_1, \boldsymbol{\beta}')) = df_i$ for all $1 \le i \le r$.
Compute $A_{\boldsymbol{\beta}'} \leftarrow \mathfrak{f}_j(x_1, \boldsymbol{\beta}')$ via probes to $\mathbf{B}$ and Lagrange interpolation.
if $\hat{f}_{i,j}(x_1, \boldsymbol{\beta}') \mid A_{\boldsymbol{\beta}'}$ for all $1 \le i \le r$ **then return** $(\hat{f}_{i,j})_{i=1}^r$ **else return** FAIL **end if**

Recursive factorization of the content Algorithm CMBBSHL computes, with high probability, the irreducible factors of $\operatorname{pp}(\mathfrak{f}, x_1)$ in their sparse representation. To obtain the irreducible factors of $\operatorname{cont}(\mathfrak{f}, x_1)$, we construct an additional black box for the content and factor it recursively.

Let $\mathfrak{f} = h \prod_{i=1}^r f_i$, where $h = \operatorname{cont}(\mathfrak{f}, x_1)$ and f_i are the irreducible factors $\operatorname{pp}(\mathfrak{f}, x_1)$ returned by Algorithm CMBBSHL following after rational num-

ber reconstruction. Let $\mathcal{C}_n = \mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$. We have $\mathcal{C}_n = \text{cont}(\mathcal{C}_n, x_1) \cdot \text{pp}(\mathcal{C}_n, x_1) = \text{cont}(\mathfrak{f}, x_1) \cdot \text{pp}(\mathfrak{f}, x_1)$. Define $\mathcal{C}_{n-1} = \text{cont}(\mathcal{C}_n, x_1) \in \mathbb{Z}[x_2, \ldots, x_n]$. This polynomial has one fewer variable (it no longer depends on x_1). We now construct a new black box $\mathbf{C}_{n-1} : \mathbb{Z}^{n-1} \times \{p\} \to \mathbb{Z}_p$ for $\mathcal{C}_{n-1}$ (described below), and then apply Algorithm CMBBSHL recursively to compute the irreducible factors of $\text{pp}(\mathcal{C}_{n-1}, x_2)$. When $n = 0$, $\mathcal{C}_0$ is an integer and it can be computed by evaluating the black box $\mathbf{C}_0 : \phi \times \{p\} \to \mathbb{Z}_p$ at several different primes and applying Chinese remaindering.

Constructing the black box of the content To construct the black box $\mathbf{C}_{n-1} : \mathbb{Z}^{n-1} \times \{p\} \to \mathbb{Z}_p$ for $\mathcal{C}_{n-1}$, we first construct a black box $\mathbf{F}_n : \mathbb{Z}^n \times \{p\} \to \mathbb{Z}_p$ such that $\mathbf{F}_n(\boldsymbol{\alpha}, p) = \text{pp}(\mathfrak{f}, x_1)(\boldsymbol{\alpha}) \bmod p$. Let $(f_{i,n})_{i=1}^r$ be the list of irreducible factors of $\text{pp}(\mathfrak{f}, x_1)$ returned by Algorithm CMBBSHL. Then,

$$\text{pp}(\mathfrak{f}, x_1)(\boldsymbol{\alpha}) = \prod_{i=1}^{r} f_{i,n}(\boldsymbol{\alpha}) \bmod p.$$

We then construct the black box $\mathbf{C}_{n-1}$ by

$$\mathbf{C}_{n-1}(\boldsymbol{\beta}, p) = \frac{\mathbf{C}_n([\gamma, \boldsymbol{\beta}], p)}{\mathbf{F}_n([\gamma, \boldsymbol{\beta}], p)} \bmod p$$

for a fixed $\gamma \in \mathbb{Z}_p$ chosen at random. If $\mathbf{F}_n([\gamma, \boldsymbol{\beta}], p) = 0$ then FAIL is returned.

If $\deg(\mathcal{C}_{n-1}, x_2) = 0$, then $\text{pp}(\mathcal{C}_{n-1}, x_2) = 1$, and no call to Algorithm CMBBSHL is needed. In this case, we simply define $\mathbf{C}_{n-1}(\boldsymbol{\beta}, p) = \mathbf{C}_n([\gamma, \boldsymbol{\beta}], p)$ for a fixed $\gamma \in \mathbb{Z}_p$ chosen at random, and proceed to the next level of recursion.

3.3 Complexity Analysis of Algorithm CMBBSHL

Theorem 6. *Let p be a large prime and $\tilde{N} < p$, $\tilde{N} \in \mathbb{Z}^+$. Let $\mathfrak{f} \in \mathbb{Z}[x_1, \ldots, x_n]$ be a multilinear Boolean polynomial represented by a modular black box $\mathbf{B}$. Let $\boldsymbol{\alpha} = (\alpha_2, \ldots, \alpha_n) \in \mathbb{Z}_p^{n-1}$ be randomly chosen such that $0 < \alpha_i < \tilde{N}$. Suppose $\boldsymbol{\alpha}$ is Hilbertian. Then, if Algorithm CMBBSHL returns an answer that is not FAIL, the total number of arithmetic operations in $\mathbb{Z}_p$ required to compute all irreducible factors of $\mathfrak{f}$ in the worst case is*

$$O\left(n^2 s_{\max}\left(\sum \#f_i + \mathfrak{C}(\textit{probe } \mathbf{B})\right)\right), \tag{2}$$

where $s_{\max}$ is the maximum number of terms in any coefficient (in the highest ranking variable) in any factor, $\sum \#f_i$ is the sum of the number of terms of all irreducible factors, and $\mathfrak{C}(\textit{probe } \mathbf{B})$ is the number of arithmetic operations in $\mathbb{Z}_p$ for one probe to the black box $\mathbf{B}$.

Proof. We first count the total number of arithmetic operations in $\mathbb{Z}_p$ in the jth sparse Hensel lifting step of Algorithm CMBBSHL (Algorithm 4). Let $d_j = \deg(\mathfrak{f}, x_j)$ for all $1 \leq j \leq n$. For a multilinear Boolean polynomial, $d_j = 1$ for all

$1 \leq j \leq n$. Also, there is only one factor in $\text{pp}(\mathfrak{f}, x_1)$, i.e. $r = 1$. Let us denote this factor as $f^{(1)}$. For $r = 1$, the bivariate Hensel lifting in Step 13 directly outputs the bivariate image $A_m = \mathfrak{f}(x_1, Y_m, x_j)$ (up to a scalar, to match the leading coefficient). In this case, the sparse Hensel lifting becomes sparsely interpolating $f^{(1)}(x_1, \ldots, x_n)$ one variable at a time.

Step 8 is the only step to probe the black box $\mathbf{B}$. We use bivariate dense interpolation to obtain a bivariate image $\mathfrak{f}_j(x_1, Y_m, x_j)$. It requires at most 4 probes to the black box $\mathbf{B}$ and $O(d_1 d_j^2 + d_1^2 d_j)$ arithmetic operations in $\mathbb{Z}_p$ for Lagrange or Newton interpolation in both variables. The total cost for step 8 is $O(s \cdot \mathfrak{C}(\text{probe } \mathbf{B}))$, since $d_1 = d_j = 1$.

For Step 13, matching the leading coefficient of A_m with the univariate image $\hat{f}_j^{(1)}(x_1, Y_m)$ requires $O(\#f^{(1)})$ multiplications in $\mathbb{Z}_p$, and a bivariate Hensel lift costs $O(d_1^2 d_j + d_1 d_j^2)$ arithmetic operations in $\mathbb{Z}_p$ [17]. Thus, the total cost for step 13 is $O(s(\#f^{(1)} + 2))$ arithmetic operations in $\mathbb{Z}_p$.

The proofs of the next two steps follow from [2]. We give the total number of arithmetic operations in $\mathbb{Z}_p$ for each step:

Step 10 costs $O(s \sum_{i=1}^{r} \#\hat{f}_{i,j-1})$, which is $O(s\#f^{(1)})$, since $r = 1$.

Step 19 costs $O(s d_j \sum_{i=1}^{r} \#\hat{f}_{i,j-1})$, which is $O(s\#f^{(1)})$.

Adding up, the total number of arithmetic operations in $\mathbb{Z}_p$ for the j th Hensel lifting step is

$$O\left(s\left(\#f^{(1)} + \mathfrak{C}(\text{probe } \mathbf{B})\right)\right),$$

where s is defined in step 5 of Algorithm 4.

Therefore, the total number of arithmetic operations in $\mathbb{Z}_p$ for computing $f^{(1)}$, the irreducible factor of $\text{pp}(\mathfrak{f}, x_1)$, is

$$O\left((n-2)s^{(1)}\left(\#f^{(1)} + \mathfrak{C}(\text{probe } \mathbf{B})\right)\right),$$

where $s^{(1)}$ is the maximum of s in all Hensel lifting steps (from 3 to n).

Now consider the recursive computation of the factors of the content. Let the superscript k denote the recursion level. Let $f^{(k)} \in \mathbb{Z}[x_k, \ldots, x_n]$ denote the irreducible factor computed at recursive level k, and let $s^{(k)}$ be the corresponding maximum value of s in the Hensel lifting steps. Define $s_{\max} := \max_k s^{(k)}$. When $k = 1$, the algorithm computes $f^{(1)}$, the only factor of $\text{pp}(\mathfrak{f}, x_1)$. When $k = 2$, the algorithm computes the factor of $\text{pp}(\text{cont}(\mathfrak{f}, x_1), x_2)$. At this level, the cost of one probe to $\mathbf{C}_{n-1}$ includes the additional cost of evaluating the irreducible factor $f^{(1)}$ at a point. Since the evaluations in the j th Hensel lifting step use powers of integers $\beta_2, \ldots, \beta_j$, the cost of evaluating $f^{(1)}$ once is $O(\#f^{(1)})$ (see [2]).

The total number of arithmetic operations in $\mathbb{Z}_p$ required to compute all irreducible factors is therefore

$$O\Bigg((n-2)s^{(1)}\left(\#f^{(1)}+\mathfrak{C}(\text{probe }\mathbf{B})\right)+(n-3)s^{(2)}\left(\#f^{(2)}+\mathfrak{C}(\text{probe }\mathbf{B})+\#f^{(1)}\right)$$
$$+(n-4)s^{(3)}\left(\#f^{(3)}+\mathfrak{C}(\text{probe }\mathbf{B})+\#f^{(1)}+\#f^{(2)}\right)+\cdots$$
$$+s^{(n-2)}\left(\#f^{(n-2)}+\mathfrak{C}(\text{probe }\mathbf{B})+\#f^{(1)}+\cdots+\#f^{(n-1)}\right)\Bigg)$$
$$\subseteq O\left(n^2 s_{\max}\left(\sum \#f_i+\mathfrak{C}(\text{probe }\mathbf{B})\right)\right),$$

since $\sum_{k=1}^{n-1}\#f^{(k)}\leq\sum\#f_i$, the total number of terms in all irreducible factors f_i of $\mathfrak{f}$. □

Example 5. Consider $f=\prod_{i=1}^{20}(x_{2i-1}+x_{2i})$. Here, $n=40$, $s_{\max}=1$, and $\sum\#f_i=40$. In general, $s_{\max}<\#f_i$ for all i. We construct a black box for f by forming a block-diagonal matrix A with 20 blocks $D_i=\begin{pmatrix}x_{2i-1}&-1\\x_{2i}&1\end{pmatrix}$, and then applying a random even permutation to the rows (or columns). Then $\det(A)=f$. Triangularizing each $\det(D_i)$ requires one division, one multiplication, and one addition. Thus $\mathfrak{C}(\text{probe }\mathbf{B})=20\cdot 3+39=99$, where the additional 39 multiplications comes from multiplying the diagonal entries. Therefore, $s_{\max}\left(\sum\#f_i+\mathfrak{C}(\text{probe }\mathbf{B})\right)=139$.

In contrast, using an explicit form of f, the complexity is $O(n^2t)$, where $t=\#f=2^{20}=1048576$.

4 Benchmarks

We have implemented Algorithm 1 in Maple and Algorithms 2 and 3 in C. The black box algorithm CMBBSHL is coded in Maple and C. All four substeps in Algorithm 4 are coded in C. We present CPU timings for them on a single core of an Intel Xeon Gold 6342 CPU running at 2.8GHz base and 3.5GHz turbo for the following benchmark problems.

Benchmark 1 For the different choices of t_1 and t_2 in Table 4, we generated two polynomials f_1 and f_2 in $n=100$ variables with t_1 terms and t_2 terms respectively, at random. For each pair of t_1,t_2 we created each monomial in f_1 in the variables $x_1,\ldots,x_{50}$ and each monomial in f_2 in the variables $x_{51},\ldots,x_{100}$. For each term in f_1 and in f_2 we choose each variable with probability $\frac{1}{2}$ at random. We then construct the input multilinear polynomial $f=f_1f_2\in\mathbb{F}_2[x_1,\ldots,x_{100}]$ to be factored.

Table 4 is divided into 4 blocks where the input polynomial has $t_1t_2=10^3$, $t_1t_2=10^4$, $t_1t_2=10^5$ and $t_1t_2=10^6$ terms. Column GCD is the time to factor f for Algorithm 1. Column FDexp is the time for Algorithm 2 with the percentage of the time spent computing $AD+BC$ in step 6 shown in parentheses.

Column FDSZ is the time for Algorithm 3 with the percentage of the time spent computing $A(\alpha)D(\alpha)$ and $B(\alpha)C(\alpha)$ in line 7 shown in parenthesis. Column CMBBSHL is the time for black box factorization algorithm CMBBSHL, and column #probes is the total number of probes to the black box $\mathbf{B}$. The black box is constructed by computing the determinant of a 2×2 matrix with each factor placed on the diagonal.

For this benchmark, Algorithm CMBBSHL is not advantages when the factors are large. This is likely due to the high level of recursion required when computing f_2. As can be seen in Table 4 computing $AD + BC$ dominates the cost of Algorithm 2 and evaluating $A(\alpha), D(\alpha), B(\alpha), C(\alpha)$ dominates the cost of the Algorithm 3. Obviously our C implementation of Algorithm 3 (column FDSZ) is much faster than Algorithms 1 (implemented in Maple) and 2 (implemented in C). The cost of the arithmetic in GF(2^{63}) is significant.

t_1	t_2	GCD	FDexp($AD-BC$)	FDSZ(hashing)	#GF	CMBBSHL	#probes
10	100	2.027	4.068(99.9%)	0.473(99.9%)	4.92m	32.042	12324
25	40	1.716	3.903(99.9%)	0.471(99.9%)	4.91m	18.627	10894
50	20	2.328	3.561(99.9%)	0.490(99.9%)	5.08m	20.576	14614
100	10	2.344	3.781(99.8%)	0.525(99.9%)	5.44m	37.724	22576
10	1000	207.8	641.4(99.9%)	4.658(99.9%)	48.50m	1170.124	88562
25	400	76.21	509.7(99.9%)	4.855(99.9%)	50.60m	109.115	41854
50	200	102.5	579.4(99.9%)	4.917(99.9%)	51.06m	52.902	30248
100	100	159.7	606.4(99.9%)	4.942(99.9%)	51.27m	38.502	31012
50	2000	6106.8	NA	47.19(99.9%)	492.4m	1419.494	174040
100	1000	8369.8	NA	48.16(99.9%)	498.8m	535.165	106048
200	500	12581.6	NA	47.97(99.9%)	499.1m	290.581	85852
316	316	13479.6	NA	47.80(99.9%)	496.7m	246.143	91088
100	10000	NA	NA	476.4(99.9%)	4,946m	26658.000	764670
200	5000	NA	NA	469.7(99.9%)	4,878m	8168.092	426212
500	2000	NA	NA	473.9(99.9%)	4,914m	2640.576	255890
1000	1000	NA	NA	475.8(99.9%)	4,926m	1985.960	273892

Table 1. CPU time in seconds for benchmark 1.

Benchmark 2 For our second benchmark, we factor $f_m = \prod_{i=1}^{m}(x_{2i} + x_{2i+1})$. The polynomial f_m has $n = 2m$ variables and 2^m terms and m factors. For this benchmark, Algorithm CMBBSHL is the fastest among all algorithms. The reason for this behavior has already been demonstrated in Example 5.

5 Implementation Notes

We describe our C implementation of Algorithms 2 and 3. The Maple and C implementation of Algorithm CMBBSHL is not open source and is protected under Maplesoft's copyright.

m	$\#f$	GCD	FDexp	FDSZ	CMBBSHL	#probes
10	1024	0.004	0.556	0.030	0.439	1118
11	2048	0.005	2.130	0.074	0.557	1352
12	4096	0.012	9.029	0.179	0.718	1610
13	8192	0.026	29.59	0.425	0.914	1892
14	16384	0.042	138.0	1.001	1.109	2196
15	32768	0.089	568.1	2.327	1.363	2522
16	65536	0.551	2851.4	5.363	1.683	2872
17	131072	2.256	–	12.23	2.022	3244
18	262144	6.359	–	27.58	2.404	3640
19	524288	16.21	–	62.05	2.791	4056
20	1048576	36.64	–	138.4	3.240	4496
21	2097152	86.93	–	307.4	3.929	4958

Table 2. CPU time to factor $f = \prod_{i=1}^{m}(x_{2i-1} + x_{2i})$.

5.1 Representation for multilinear polynomials in $\mathbb{F}_2[x_1, \ldots, x_n]$

If f is multilinear in n variables with t terms we encode all t monomials in an array X of 64 bit unsigned integers using the following C type definition.

```
typedef struct poly {
    int n;  // number of variables
    int t;  // number of terms
    unsigned long *terms; // t bit vectors
} poly;
```

Each monomial is encoded in $m = \lceil n/64 \rceil$ 64 bit words thus X has length mt words. The t terms in X are sorted in descending lexicographical order with $x_n > \cdots > x_2 > x_1$. For example, if $f = x_1x_2x_3 + x_3 + x_2 + 1$ we have $m = 1$, $t = 4$, $X[0] = 7$, $X[1] = 4$, $X[2] = 2$ and $X[3] = 0$.

We note that in this ordering, if $f = Ax_i + B$ for any variable, when we extract the coefficients A and B (e.g. in step 3 of Algorithm 3), the terms remain sorted. However, when we project f onto a subset of the variables (e.g. in step 10 of Algorithm 3), the terms in general will not be sorted.

5.2 Multiplication in GF(2^{63})

We have computed an irreducible polynomial $m \in \mathbb{F}_2[z]$ of degree 63 and we encode elements of $F = GF(2^{63}) = \mathbb{F}_2[z]/m(z)$ as 64 bit unsigned integers. If $a \in \mathbb{F}_2[z]$ with $\deg(a) < \deg(m) = 63$ then the encoding is just $a(2)$. The key operation $a(z)b(z) \mod m(z)$ is coded a sequence of multiplications by z and subtractions. For clarity we pseudo-code for the algorithm in $\mathbb{F}_2[z]/m(z)$ and the corresponding C code using bit operations.

We represent $A(z), B(z), M(z)$ by the 64 bit integers $a = A(2), b = B(2), m = M(2)$. The multiplication by z in line 4 becomes a left shift by 1 bit and subtraction of $M(z)$ in line 5 becomes a bit wise exclusive.

Algorithm 5 GFmultiply

Input: $m \in \mathbb{F}_2[z]$ of $d > 0$, irreducible over $\mathbb{F}_2$;
$a, b \in \mathbb{F}_2[z]$ with $\deg(a) < d$ and $\deg(b) < d$.
Output: $c = ab \bmod m$

1: **if** $a = 0$ **or** $b = 0$ **return** 0 **end if**
2: $c \leftarrow 0$
3: **for** $i = \deg(b)$ by -1 to 0 **do**
4: $\quad c \leftarrow z \cdot c$
5: $\quad$ **if** $\deg(c) = d$ **then** $c \leftarrow c - m$ **end if**
6: $\quad$ **if** $\text{coeff}(b, z, i) = 1$ **then** $c \leftarrow c + a$ **end if**
7: **end for**
8: **return** c

```
#define ULONG unsigned long
ULONG mul( ULONG a, ULONG b, ULONG B, ULONG m, ULONG M ) {
   // M = 2^deg(m), B = 2^deg(b)
   ULONG c;
   if( a==0 || b==0 ) return 0;
   c = 0;
   while( B ) {
      c = c << 1; // c = 2 c
      if( c & M ) c = c ^ m; // c = c xor m
      if( b & B ) c = c ^ a; // c = c xor a
      B = B >> 1; // B = B/2
   }
   return c;
}
```

5.3 Polynomial Library

Our software library for multinomial polynomials over $\mathbb{F}_2$ supports the following routines for the user for multilinear f and g.

- `poly* newpoly(int n, int t);` (constructor)
  Create a new polynomial with space for t terms in n variables
- `poly* randpoly(int n, int t);`
  Create a new polynomial with t terms in n variables of degree at most 1 where each term is obtained by choosing each variable with probability $\frac{1}{2}$.
- `void freepoly(poly* f);`
- `void printpoly(poly* f,char x);`
- `poly* addpoly(poly* f, poly* g);`
- `poly* mulpoly(poly* f, poly* g);`

- `poly* divpoly(poly* f, poly* g);` Compute f quo g assuming $g|f$ with 0 remainder.
- `poly* gcdpoly(poly* f, poly* g);` Compute $\gcd(f, g)$.

- `poly* evalxto0(poly *f, int x);` Compute $f(x = 0)$ the constant coefficient of f in x
- `poly* diffpoly(poly* f, int x);` Compute $\partial f/\partial x$, the coefficient of f in x
- `poly* project( poly *f, ULONG * X );` Project f onto the variables in the bit vector X
- `listpoly* facpoly(poly* f);`
  Output a linked list of the irreducible factors of f.

Our software is available at `www.cecm.sfu.ca/~mmonagan/code/facpoly`
For ease of use we have also provided a parser to read a multilinear polynomial from a text file. The sample code facpoly reads in a multilinear polynomial, factors it and print out the factors found. Reading is very fast. For example if the text file foo contains

```
x1*x3*x4*x5+x2*x3*x4*x5+x1*x3+x2*x3
```

executing the Unix command `./facpoly 5 foo` produces the output

```
f1 := x2+x1;
f2 := x3;
f3 := x4*x5+1;
```

6 Conclusion

We presented two fast algorithms for factoring multilinear Boolean polynomials: a Monte Carlo method with $O(n^2t)$ complexity for explicit sparse input, and the black box algorithm CMBBSHL for cases where f is available only through evaluation. Our implementations demonstrate substantial practical speedups over existing approaches, and the black box method further reduces complexity when $s_{\max}\left(\sum \#f_i + \mathfrak{C}(\text{probe } \mathbf{B})\right) \ll t$. Together, these results provide the most efficient practical tools currently available for factoring multilinear Boolean polynomials.

References

1. Steven W. Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors, *J. ACM*, **18**(4):478–504, ACM, 1971.
2. T. Chen and M. Monagan. The complexity and parallel implementation of two sparse multivariate Hensel lifting algorithms for polynomial factorization. In Proceedings of CASC 2020, LNCS **12291**: 150–169. Springer (2020)
3. T. Chen and M. Monagan. Factoring multivariate polynomials represented by black boxes: A Maple + C Implementation. *Math. Comput. Sci.* **16**,18 (2022)
4. T. Chen and M. Monagan. A new black box factorization algorithm – the non-monic case. In Proceedings of ISSAC 2023. ACM, 2023
5. T. Chen. Sparse Hensel lifting algorithms for multivariate polynomial factorization. PhD Thesis (2024)

6. S. D. Cohen. The distribution of Galois groups and Hilbert's irreducibility theorem. In Proceedings of the London Mathematical Society (3), 43:227–250 (1981)
7. Pavel Emelyanov and Denis Ponomaryov. On a Polytime Factorization Algorithm for Multilinear Polynomials over $\mathbb{F}_2$. Proceedings of CASC 2018, LNCS **11077**:164–176, Springer, 2018.
8. E. Kaltofen. Sparse Hensel lifting. In Proceedings of EUROCAL '85, LNCS **204**, 4–17. Springer (1985)
9. Angel Diaz and Erich Kaltofen. On Computing Greatest Common Divisors with Polynomials given by Black Boxes for their Evaluations. Proceedings of ISSAC '95, pp. 232–239, ACM, 1995.
10. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press (2013)
11. Jiaxiong Hu and Michael Monagan. A Fast Parallel Sparse Polynomial GCD Algorithm. *J. Symb. Cmpt.* **105**:(1) 28–63, Springer, July 2021.
12. Qiao-Long Huang and Michael Monagan. A New Sparse Polynomial GCD by Separating Terms. Proceedings of ISSAC 2024, pp. 134-142, ACM Digital Library, 2024.
13. E. Kaltofen E and B. M. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symb. Cmpt.* **9**(3), 301–320. Elsevier (1990)
14. Jennifer de Kleine, Michael Monagan, and Allan Wittkopf. Algorithms for the Nonmonic case of the Sparse Modular GCD Algorithm. *Proceedings of ISSAC '2005*, pp. 124–131, ACM, 2005.
15. W. S. Lee. Early termination strategies in sparse interpolation algorithms. Ph.D. Thesis (2001)
16. Michael Monagan and Roman Pearce. The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. *Communications in Computer Algebra*, **48**(4):166–186, ACM, December 2014.
17. M. Monagan and G. Paluck. Linear Hensel lifting for $\mathbb{Z}_p[x,y]$ for n factors with cubic cost. In Proceedings of ISSAC 2022, 169-166. ACM (2022)
18. R. Rubinfeld and R. E. Zippel. A new modular interpolation algorithm for factoring multivariate polynomials. In Proceedings of Algorithmic Number Theory, First International Symposium, ANTS-I (1994)
19. Amir Shpilka and Ilya Volkovich. On the relation between polynomial identity testing and finding variable disjoint factors. Proceedings of ICALP 2010. LNCS **6198**:408-419, Springer, 2010.
20. Paul S. Wang. The EEZ-GCD algorithm. SIGSAM Bulletin **14**(2):50–60, ACM, 1980.
21. P. S. Wang. An improved multivariate polynomial factoring algorithm. *Math. Comp.* **32**, 1215–1231 (1978)
22. P. S. Wang, M. J. T. Guy, and J. H. Davenport. p-adic reconstruction of rational numbers. SIGSAM Bulletin, **16**(2) (1982)
23. Richard Zippel. Probabilistic algorithms for sparse polynomials. Proceedings of EUROSAM '79, LNCS **72**:216–226, Springer, 1979.
24. R. E. Zippel. Interpolating polynomials from their values. *J. Symb. Cmpt.* **9**(3), 375–403 (1990)