A Comprehensive Analysis of Constant-time Polynomial Inversion for Post-quantum Cryptosystems

Alessandro Barenghi*
Politecnico di Milano
Milano, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano
Milano, Italy
gerardo.pelosi@polimi.it

ABSTRACT

Post-quantum cryptosystems have currently seen a surge in interest thanks to the current standardization initiative by the U.S.A. National Institute of Standards and Technology (NIST). A common primitive in post-quantum cryptosystems, in particular in code-based ones, is the computation of the inverse of a binary polynomial in a binary polynomial ring. In this work, we analyze, realize in software, and benchmark a broad spectrum of binary polynomial inversion algorithms, targeting operand sizes which are relevant for the current second round candidates in the NIST standardization process. We evaluate advantages and shortcomings of the different inversion algorithms, including their capability to run in constant-time, thus preventing timing side-channel attacks.

CCS CONCEPTS

• **Security and privacy** → **Public key encryption**; • **Theory of computation** → *Cryptographic primitives.*

KEYWORDS

Post-quantum Cryptosystems, Code-based Cryptography, Constant-time Algorithms, Timing Side-channel Attacks.

ACM Reference Format:
Alessandro Barenghi and Gerardo Pelosi. 2020. A Comprehensive Analysis of Constant-time Polynomial Inversion for Post-quantum Cryptosystems. In *17th ACM International Conference on Computing Frontiers (CF '20),May 11–13, 2020, Catania, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3387902.3397224

1 INTRODUCTION

The research interest in post-quantum cryptography has been growing since August 2015 when the U.S.A. National Security Agency published an online note[1] announcing preliminary plans for transitioning from factoring- and discrete logarithm-based cryptographic algorithms to quantum-computing resistant ones. The popularity of the topic further increased in December 2016 with the U.S.A. National Institute of Standards and Technology (NIST) announcing an international call for proposals for post-quantum cryptographic algorithms[2]. The proposals currently selected for the second round of the contest encompass a large part of the state-of-the-art in computational techniques such as algorithms in algebraic geometry, coding theory, and lattice theory. An essential requirement for the proposals that will be recommended in the future portfolio of post-quantum cryptographic primitives is to provide efficient software implementations optimized for x86_64 architectures, with the NIST specifying Intel Haswell as its primary benchmark target.

Code based cryptosystems have a remarkably good security track; however, such strength comes at the disadvantage of quite large public key sizes (in the megabyte range). A promising avenue to reduce the key size, is to employ code families admitting a space-efficient representation, such as quasi-cyclic moderate-density parity-check (QC-MDPC) codes. Quasi-cyclic codes are characterized by quasi-cyclic generator and parity check matrices, i.e., they are composed by circulant, square sub-matrices, where all the rows are obtained by cyclically shifting the first one. The arithmetics of such matrices with size p is isomorphic to the one of the polynomials modulo $x^p - 1$ over the same field as the coefficients of the circulant matrices. In particular, in the case of binary linear block codes, the arithmetics of $p \times p$ circulant matrices over $\mathbb{Z}_2$ can be substituted with the arithmetics of polynomials in $\mathbb{Z}_2[x]/(x^p - 1)$. This, in turn, implies a reduced size of the keypairs and faster arithmetic operations.

Contributions. We analyze four efficient algorithms for polynomial inversion over $\mathbb{Z}_2[x]/(x^p - 1)$, which represents the most time consuming operation in the key-generation of LEDAcrypt [1–4], a current second round candidate to the U.S.A. NIST standardization effort. We describe in detail the state-of-the-art of the inversion techniques relying on Euclid's algorithm, highlighting the non straightforward similarities between the approaches, and providing a detailed version of the strategy introduced in [5], tailored to inverses over $\mathbb{Z}_2[x]/(x^p - 1)$. We also tailor the approach based on Fermat's little theorem, describing an optimal square and multiply chain to compute the required exponentiation. We benchmark our implementation of all four algorithms, employing the features offered by the extensions of the Instruction Set Architecture (ISA) available in Intel Haswell CPUs, and validate the constant time nature of the algorithms which are expected to enjoy this property.

Organization of the paper. In Section 2, we provide an overview of the LEDAcrypt primitives and introduce the schoolbook Euclid's algorithm for the computation of polynomial multiplicative

*Both authors contributed equally to the paper
[1] www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm

[2] www.nist.gov/pqcrypto

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF '20, May 11–13, 2020, Catania, Italy
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7956-4/20/05…$15.00
https://doi.org/10.1145/3387902.3397224

inverses. In Section 3, we present our systematization of the optimized algorithms to compute polynomial multiplicative inverses reporting a description of the approaches employed by Brunner *et al.* [7], Kobayashi *et al.* [11] and Bernstein *et al.* [5]. The section ends with the presentation of our approach to optimize the inversion algorithm based on the Fermat's Little Theorem. In Section 4 we report the implementation details of the presented algorithms to end of comparing both their performance and their behavior w.r.t. the information leakage (possibly) arising from the timing side channel. Finally, Section 5 reports our concluding remarks.

2 PRELIMINARIES

The current LEDAcrypt specification [3] describes an IND-CPA and an IND-CCA2 Key Encapsulation Methods (KEMs) relying on the Niederreiter cryptoscheme [13], named LEDAcrypt-KEM-CPA and LEDAcrypt-KEM, respectively. Moreover, a Public Key Encryption (PKC) system, named LEDAcrypt-PKC, relying on the McEliece cryptoscheme [12] plus the IND-CCA2 Kobara-Imai γ-construction is reported as well. The three systems employ a binary QC-MDPC code having a systematic public parity-check matrix representation. The KEY-GENERATION algorithms of all mentioned primitives consider a QC-MDPC error corection code $C(n,k)$, with codeword length $n{=}pn_0$ and information word length $k{=}p(n_0{-}1)$, where $n_0{\in}\{2,3,4\}$, p is a large prime number such that $\mathtt{ord}_p(2) = p-1$ (i.e., 2 is a primitive element of the Galois field GF(p)$\cong \mathbb{Z}/p\mathbb{Z}$). The private key is composed by the quasi-cyclic $p{\times}pn_0$ parity-check matrix of the code $C(n,k)$, i.e.: $sk{=}\{H\}$, which is in turn structured as $1{\times}n_0$ circulant blocks, each of which with size $p{\times}p$ and with v non-null elements per row/column, where v is an odd number to guarantee full-rank blocks: $H{=}[H_0, H_1, \ldots, H_{n_0-1}]$. The matrix H is generated ensuring that each one of the full-rank n_0 circulant blocks $H{=}[H_0, H_1, \ldots, H_{n_0-1}]$ has a number of asserted bits in the first row equal to v ($n_0 \cdot v \approx \sqrt{pn_0}$). Subsequently, starting from the multiplicative inverse of H_{n_0-1}, the public key pk of the KEM primitive is obtained as: $M = H^{-1}_{n_0-1}H = \left[M_0|M_1|\ldots|M_{n_0-2}|I\right]$, where I is a $p{\times}p$ identity matrix. The public key of the LEDAcrypt-PKC is obtained by converting M into the corresponding systematic generator matrix of the same code as $pk = \left[D \mid [M_0 \mid \ldots \mid M_{n_0-2}]^T\right]$, where D is a block matrix with $n_0{-}1$ replicas of I on its diagonal.

2.1 Polynomial Multiplicative Inverses

The KEY-GENERATION algorithm of all LEDAcrypt primitives require $n_0 - 2$ multiplications between $p \times p$ binary blocks, each of which represented as an invertible polynomial in the ring $\mathbb{Z}_2[x]/(x^p-1)$ with an odd and small number of asserted coefficients $v \approx \sqrt{p/n_0}$. In particular, the first factor of each multiplication is given by the multiplicative inverse of the polynomial $a(x) \in \mathbb{Z}_2[x]/(x^p-1)$ corresponding to the first row of the last circulant-block of the secret parity-check matrix $H{=}[H_0, H_1, \ldots, H_{n_0-1}]$.

Although the computation of a multiplicative inverse polynomial is required only once during the execution of the KEY-GENERATION algorithm for the IND-CCA2 LEDAcrypt-KEM and the IND-CCA2 LEDAcrypt-PKC system, the length/degree of the involved operand $a(x)$, $7{\cdot}10^3{<}\mathtt{deg}(a(x)){<}9{\cdot}10^4$, requires to implement carefully this operation by establishing the inversion strategy that fits best the length of the involved operand. Furthermore, the LEDAcrypt-KEM-CPA system generates ephemeral private/public key pairs to transmit a session key at each run, requiring the computation of a multiplicative inverse polynomial each time. Finally, another feature steering the choice of the best polynomial inversion algorithm for a given primitive and operand size is the posssibility to exhibit a constant-time implementation to prevent timing based side-channel analyses aimed at the recovering of the secret key $sk = \{H\}$.

Algorithm 1: Inversion using the Euclid's gcd Algorithm

Input: $f(x)$ irreducible polynomial of GF(2^m), $m \geq 2$,
$a(x)$ invertible element of GF(2^m)

Output: $\mathsf{V}(x)$, polynomial such that $a(x)^{-1} \equiv \mathsf{V}(x) \bmod f(x)$

```
1  begin
2      S(x) ← f(x), R(x) ← a(x)
3      V(x) ← 0, U(x) ← 1
4      repeat
5          Q(x) ← ⌊S(x)/R(x)⌋
6          tmp(x) ← S(x) − Q(x) · R(x), S(x) ← R(x), R(x) ← tmp(x)
7          tmp(x) ← V(x) − Q(x) · U(x), V(x) ← U(x), U(x) ← tmp(x)
8      until (R(x) = 0)
9      return V(x)
10 end
```

2.2 SchoolBook Euclid's Algorithm

The Euclid's algorithm to compute the greated common divisor, gcd, between two polynomials is commonly employed to derive a polynomial time algorithm yielding the multiplicative inverse of an element in the multiplicative group of a Galois field represented in polynomial basis, e.g., the multiplicative group of GF(2^m), $m \geq 2$ with GF(2^m) $\cong \mathbb{Z}_2[x]/(f(x))$, where $f(x)$ is an irreducible polynomial with $\mathtt{deg}(f(x)) = m$.
A schoolbook analysis considers a polynomial $a(x)$ and the irreducible polynomial $f(x)$, employed to represent the field elements ($\mathtt{deg}(a(x)) < \mathtt{deg}(f(x))$), to iteratively apply the equality $\mathtt{gcd}(f(x), a(x))$= $\mathtt{gcd}(a(x), f(x) \bmod a(x))$ and derive the computation path leading to the non-null constant term r epresenting the greatest common divisor d (e.g., $d = 1$, when $\mathbb{Z}_2[x]$ is considered).

$$
\begin{array}{ll}
\mathtt{deg}(f(x)) > \mathtt{deg}(a(x)) > 0 & d{=}\mathtt{gcd}(f(x), a(x)) \\
r_0{=}f(x),\ r_1{=}a(x) & d{=}\mathtt{gcd}(r_0(x), r_1(x)) \\
r_2{=}r_0 \bmod r_1,\quad 0{\leq}\mathtt{deg}(r_2){<}\mathtt{deg}(r_1) & d(x){=}\mathtt{gcd}(r_1(x), r_2(x)) \\
\ldots & \ldots \\
r_i{=}r_{i-2} \bmod r_{i-1},\quad 0{\leq}\mathtt{deg}(r_i){<}\mathtt{deg}(r_{i-1}) & d = \mathtt{gcd}(r_{i-1}(x), r_i(x)) \\
\ldots & \ldots \\
r_z{=}r_{z-2} \bmod r_{z-1},\quad r_z(x){=}0 & d{=}\mathtt{gcd}(r_{z-1}(x), 0){=}r_{z-1}(x)
\end{array}
$$

for a proper number of iterations, $z \geq 2$. A close look to the previous derivation shows that two series of auxiliary polynomials $w_i(x)$ and $u_i(x)$, $0 \leq i \leq z{-}1$, can be defined to derive the series of remainders $r_i(x)$ as: $r_i(x) = r_{i-2} - q_i(x) \cdot r_{i-1}(x) = w_i(x) \cdot f(x) + u_i(x) \cdot a(x)$, where $q_i(x) = \left\lfloor \frac{r_{i-2}(x)}{r_{i-1}(x)} \right\rfloor$, with $r_0(x) = f(x)$ and $r_1(x) = a(x)$. Specifically, the computations shown above can be rewritten as follows.

$\mathsf{deg}(f(x)) > \mathsf{deg}(a(x)) > 0$
$r_0 = 1 \cdot f(x) + 0 \cdot a(x) = w_0(x) \cdot f(x) + u_0(x) \cdot a(x)$
$r_1 = 0 \cdot f(x) + 1 \cdot a(x) = w_1(x) \cdot f(x) + u_1(x) \cdot a(x)$
$r_2 = r_0 \bmod r_1 =$
$= (w_0(x) \cdot f(x) + u_0(x) \cdot a(x)) - \left\lfloor \frac{r_0}{r_1} \right\rfloor (w_1(x) \cdot f(x) + u_1(x) \cdot a(x)) =$
$= w_2(x) \cdot f(x) + u_2(x) \cdot a(x)$
$\cdots$
$r_i = r_{i-2} \bmod r_{i-1} = w_i(x) \cdot f(x) + u_i(x) \cdot a(x)$
$\cdots$
$r_{z-1} = r_{z-3} \bmod r_{z-2} = w_{z-1}(x) \cdot f(x) + u_{z-1}(x) \cdot a(x)$
$r_z = r_{z-2} \bmod r_{z-1} = 0$

Finally, the multiplicative inverse of $a(x)$ is obtained from the equality $d = w_{z-1}(x) \cdot f(x) + u_{z-1}(x) \cdot a(x)$, by computing:

$$a(x)^{-1} \equiv \left(d^{-1} \cdot u_{z-1}(x) \bmod f(x)\right).$$

In the last derivation of remainder polynomials, it is worth noting that $w_i(x) = w_{i-2}(x) - q_i(x) \cdot w_{i-1}(x)$, and $u_i(x) = u_{i-2}(x) - q_i(x) \cdot u_{i-1}(x)$, with $w_0(x) = 1$, $u_0(x) = 0$ and $w_1(x) = 0$, $u_1(x) = 1$.

Restricting ourselves to the case of binary polynomials, Algorithm 1 shows the pseudo-code for computing an inverse employing only the strictly needed operations. Note that the execution of an actual division would prevent the efficient implementation of the algorithm, therefore in the following section we consider the optimizations to this algorithm that circumvent its computation. The naming conventions adopted in Algorithm 1 denote as $\mathsf{R}(x)$ the i-th remainder of the gcd procedure described before, and as $\mathsf{S}(x)$ the $(i-1)$-th remainder, with $i \geq 1$. Furthermore, the i-th $u(x)$ value is denoted as $\mathsf{U}(x)$, while the value it took at the previous iteration (i.e., the $(i-1)$-th $u(x)$ value) is denoted as $\mathsf{V}(x)$, with $i \geq 1$.
Note that in the last iteration (the z-th one, counting the first as 1) $\mathsf{R}(x) = r_z(x) = 0$, while $\mathsf{S}(x) = r_{z-1}(x) = 1$.
The rationale to keep the notation of the pseudo-code variables as polynomials in x lies on the willingness of not specifying the implementation dependent choice for the endianness of values stored in array variables, i.e., if the least significant coefficient of a polynomial is stored in the first or last cell of an array.

In the case of LEDAcrypt where the arithmetic operations are performed in $\mathbb{Z}_2[x]/(x^p - 1)$, with $\mathsf{ord}_p(2) = p - 1$, it is worth noting that the Euclid's algorithm can still be applied to compute the inverse of invertible elements as the factorization of $f(x) = x^p - 1$ in irreducible factors, i.e., $f(x) = (x+1) \cdot (\sum_{i=0}^{p-1} x^i)$, shows that every binary polinomial $a(x)$ with degree less than p and an odd number of asserted coefficients, that is also different from the said factors, (i.e., any polynomial representing a circular block in the LEDAcrypt) always admits a greater common divisor equal to one (i.e., $\mathsf{gcd}(a(x), f(x)) = 1$). As a consequence, the steps of the Euclid's algorithm to compute inverses remain the same shown before.

3 OPTIMIZED POLYNOMIAL INVERSIONS

In [7] Brunner *et al.* optimized the computation performed by Algorithm 1 embedding into it the evaluation of the quotient and remainder of the polynomial division $\lfloor \mathsf{S}(x)/\mathsf{R}(x) \rfloor$. Indeed, Algorithm 2 performs the division operation by repeated shifts and subtractions, while keeping track of the difference δ between the degrees of polynomials in $\mathsf{S}(x)$ (which is the dividend and starts by containing the highest degree polynomial, that is, the modulus $f(x)$) and $\mathsf{R}(x)$ (which is the divisor, and is initialized to the polynomial of which the inverse should be computed, $a(x)$). The difference between the degrees, $\delta = \mathsf{deg}(\mathsf{S}) - \mathsf{deg}(\mathsf{R})$ is updated each time the degree of either $\mathsf{S}(x)$ or $\mathsf{R}(x)$ is altered, via shifting.

Algorithm 2: Brunner *et al.* (BCH) Inversion Algorithm [7]

Input: $f(x)$ irreducible polynomial of $\mathrm{GF}(2^m)$, $m \geq 2$,
$a(x)$ invertible element of $\mathrm{GF}(2^m)$

Output: $\mathsf{V}(x)$, such that $a(x)^{-1} \equiv \mathsf{V}(x) \in \mathrm{GF}(2^m)$

Data: polynomial variables $\mathsf{R}(x)$, $\mathsf{S}(x)$, $\mathsf{U}(x)$, $\mathsf{V}(x)$ are assumed to have at most $m+1$ coefficients each (e.g., $\mathsf{R}(x) = \mathsf{R}_m \cdot x^m + \mathsf{R}_{m-1} \cdot x^{m-1} + \cdots + \mathsf{R}_1 \cdot x + \mathsf{R}_0$, with $\mathsf{R}_i \in \mathbb{Z}_2$, $0 \leq i \leq m$) to prevent reduction operations (i.e., mod $f(x)$) among the intermediate values of the computation; δ stores a signed integer number

```
1  begin
2      S(x) ← f(x), R(x) ← a(x)
3      V(x) ← 0, U(x) ← 1, δ ← 0
4      for i ← 1 to 2·m do
5          if (R_m = 0) then
6              R(x) ← x · R(x)
7              U(x) ← x · U(x)
8              δ ← δ + 1
9          else
10             if (S_m = 1) then
11                 S(x) ← S(x) − R(x)
12                 V(x) ← V(x) − U(x)
13             else
14                 S(x) ← x · S(x)
15                 if (δ = 0) then
16                     tmp(x) ← R(x), R(x) ← S(x), S(x) ← tmp(x)
17                     tmp(x) ← U(x), U(x) ← V(x), V(x) ← tmp(x)
18                     U(x) ← x · U(x)
19                     δ ← 1
20                 else
21                     U(x) ← U(x)/x
22                     δ ← δ − 1
23     return V(x)
```

Brunner *et al.* in [7] observed that, since either a single bit-shift or a subtraction is performed at each iteration, the number of iterations equals $2m$, that is the number m of single bit-shifts needed to consider every bit of the element to be inverted, plus the number of substractions to be performed after each alignment. The last iteration of the Algorithm 2 computes $\mathsf{R}(x) = 0$, $\mathsf{S}(x) = 1$ (i.e., $\mathsf{deg}(\mathsf{R}) = \mathsf{deg}(\mathsf{S}) = 0$), and outputs the result in $\mathsf{V}(x)$. The part of Algorithm 2 dealing with computations on $\mathsf{U}(x)$ and $\mathsf{V}(x)$ mimics the same operations, in the same order, that are applied to $\mathsf{R}(x)$ and $\mathsf{S}(x)$, respectively, as the schoolbook algorithm does (Alg. 1).

3.1 Kobayashi-Takagi-Takagi Algorithm

The target implementation of Algorithm 2 is a dedicated hardware one, therefore the algorithm simplifies the computations to be performed in it, reducing them to single bit-shifts, bitwise xors and single bit comparison. While this approach effectively shortens the combinatorial cones of a hardware circuit, yielding advantages in timing, when directly transposed in a software implementation it may fail to exploit the wide computation units that are present in a CPU to the utmost. In particular, modern desktop, and high-end mobile CPUs are endowed with a so-called carryless multiplier computation unit. Such a unit computes the product of two binary polynomials with degree lower than the architecture word size, w with a latency which is definitely smaller than computing the

same operation with repeated shifts and xors (3–7 cycles on Intel CPUs [9]). To maximize the exploitation of the available carryless multipliers, Kobayashi *et al.* [11] optimized further the gcd based Algorithm 2. The main assumption in this sense is that a w-bit input carryless multiplier is available on the target CPU architecture. Algorithm 3 reports the result of the optimization proposed in [11], which still takes as input an irreducible polynomial of $\mathrm{GF}(2^m)$, employed to build a representation for all its elements, and one of its invertible elements. However, these polynomials and the ones computed as intermediate values of the algorithm, are now thought as polynomials of degree at most $M-1$, with $M = \lceil (m+1)/w \rceil$, having as coefficients binary polynomials with degree $w-1$. For example,the polynomial $S(x) = s_m \cdot x^m + s_{m-1} \cdot x^{m-1} + \cdots + s_1 \cdot x + s_0$, with $s_j \in \{0,1\}$ and $0 \le j \le m$, is processed by considering it as $S(x) = S_{M-1}(x) \cdot (x^w)^{M-1} + \cdots + S_1(x) \cdot (x^w)^1 + S_0(x)$, $S_i(x) = s_{iw+w-1} \cdot x^{w-1} + \cdots + s_{iw+1} \cdot x + s_{iw}$, where $s_{iw+l} = 0$ if $iw + l > m$, with $0 \le l \le w-1$, and $0 \le i \le M-1$.

Employing this approach, the loop at lines 5–22 of Algorithm 2 is rewritten to perform fewer iterations, each one of them corresponding to the computation made in w iterations of Algorithm 2. The crucial observation is that the computations performed by Algorithm 2 on $\mathsf{U}(x), \mathsf{V}(x), \mathsf{R}(x)$,and $\mathsf{S}(x)$ at each iteration can be represented as a linear transformation of the vectors $\begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix}$ and $\begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix}$. Indeed, the authors represent the computation portions of Algorithm 2, driven by the values of $\mathsf{S}(x)$ and $\mathsf{R}(x)$, present at lines 6–7 as $\begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix} = \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix}$, $\begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix} = \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix}$, and rewrite similarly also lines 11–12, line 14, lines 16–18 and line 21. With this representation of the computation, Algorithm 3 selects a portion of $\mathsf{R}(x)$ and $\mathsf{S}(x)$ as large as an architectural word (line 5 of Algorithm 3) and makes a copy of them (variables $\mathsf{C}(x)$ and $\mathsf{D}(x)$). Employing these copies, Algorithm 3 computes a single linear transformation, $\mathcal{H}$ cumulating the effects of w computations of the main loop of Algorithm 2, to apply them all at once (line 33, Algorithm 3). While this approach appears more expensive, as polynomial multiplications are employed to compute the values of $\mathsf{U}(x), \mathsf{V}(x), \mathsf{R}(x)$,and $\mathsf{S}(x)$ at the end of each iteration of the loop at lines 5–33, we recall that such multiplications can be carried out at the cost of a handful of xors by the carryless multipliers present on modern CPUs, effectively resulting in a favourable tradeoff for the strategy described by Algorithm 3. To provide a further speedup, Algorithm 3 has a shortcut in case entire word-sized portions of $\mathsf{R}(x)$ (which is initialized with the input polynomial to be inverted) are filled with zeroes, a fact which can be efficiently checked in one or a few CPU operations in software. Indeed, in that case, the resulting transformation to be applied is given by $\mathcal{H} = \begin{pmatrix}x^w & 0\\ 0 & 1\end{pmatrix}$; therefore the authors of Algorithm 3 insert a shortcut (lines 6–9) to skip the expensive execution of the loop body computing a trivial transformation, simply applying the aforementioned $\mathcal{H}$, which boils down to two word-sized operand shifts (line 7). We note that this efficiency improvement trick, while effectively making the inversion faster, it also provides an information leakage via the timing side channel. Indeed, due to this shortcut, the inversion

Algorithm 3: Kobayashi *et al.* (KTT) Inversion Algorithm [11]

Input: $f(x)$ irreducible polynomial of $\mathrm{GF}(2^m)$, $m \ge 2$,
$a(x)$ invertible element of $\mathrm{GF}(2^m)$

Output: $\mathsf{V}(x)$, such that $a(x)^{-1} \equiv \mathsf{V}(x) \in \mathrm{GF}(2^m)$

Data: Polynomials $\mathsf{R}(x), \mathsf{S}(x), \mathsf{U}(x), \mathsf{V}(x)$ have at most m+1 binary coefficients to prevent the execution of modulo operations in the intermediate computations. Furthermore, they are assumed to be processed a w-bit chunk at a time, i.e., $M = \lceil (m+1)/w \rceil$ and $\mathsf{R}(x) = \mathsf{R}_{M-1}(x) \cdot (x^w)^{M-1} + \cdots + \mathsf{R}_1(x) \cdot (x^w)^1 + \mathsf{R}_0(x)$, with $0 \le \deg(\mathsf{R}_i(x)) < w$, $0 \le i \le M-1$;
Polynomials $\mathsf{C}(x)$ and $\mathsf{D}(x)$ are such that $0 \le \deg(\mathsf{C}), \deg(\mathsf{D}) < w$.

1 **begin**
2 $\quad \mathsf{S}(x) \leftarrow f(x)$, $\mathsf{R}(x) \leftarrow a(x)$
3 $\quad \mathsf{V}(x) \leftarrow 0$, $\mathsf{U}(x) \leftarrow x$, $\mathsf{degR} \leftarrow M \cdot w - 1$, $\mathsf{degS} \leftarrow M \cdot w - 1$
4 $\quad$ **while** $\mathsf{degR} > 0$ **do**
5 $\quad\quad \mathsf{C}(x) \leftarrow \mathsf{R}_{M-1}(x)$, $\mathsf{D}(x) \leftarrow \mathsf{S}_{M-1}(x)$
6 $\quad\quad$ **if** $\mathsf{C}(x) = 0$ **then**
7 $\quad\quad\quad \mathsf{R}(x) \leftarrow x^w \cdot \mathsf{R}(x)$, $\mathsf{U}(x) \leftarrow x^w \cdot \mathsf{U}(x)$
8 $\quad\quad\quad \mathsf{degR} \leftarrow \mathsf{degS} - w$
9 $\quad\quad\quad$ **goto** line **4**
10 $\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$, $j \leftarrow 1$
11 $\quad\quad$ **while** $j < w$ **and** $\mathsf{degR} > 0$ **do**
12 $\quad\quad\quad j \leftarrow j + 1$
13 $\quad\quad\quad$ **if** $\mathsf{C}_{w-1} = 0$ **then**
14 $\quad\quad\quad\quad \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix} \leftarrow \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix}$
15 $\quad\quad\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \mathcal{H}$
16 $\quad\quad\quad\quad \mathsf{degR} \leftarrow \mathsf{degR} - 1$
17 $\quad\quad\quad$ **else if** $(\mathsf{degR} = \mathsf{degS})$ **then**
18 $\quad\quad\quad\quad \mathsf{degR} \leftarrow \mathsf{degR} - 1$
19 $\quad\quad\quad\quad$ **if** $\mathsf{D}_{w-1} = 1$ **then**
20 $\quad\quad\quad\quad\quad \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix} \leftarrow \begin{pmatrix}x & -x\\ 1 & 0\end{pmatrix} \times \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix}$
21 $\quad\quad\quad\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}x & -x\\ 1 & 0\end{pmatrix} \times \mathcal{H}$
22 $\quad\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad\quad \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix} \leftarrow \begin{pmatrix}0 & x\\ 1 & 0\end{pmatrix} \times \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix}$
24 $\quad\quad\quad\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}0 & x\\ 1 & 0\end{pmatrix} \times \mathcal{H}$
25 $\quad\quad\quad$ **else**
26 $\quad\quad\quad\quad \mathsf{degS} \leftarrow \mathsf{degS} - 1$
27 $\quad\quad\quad\quad$ **if** $\mathsf{D}_{w-1} = 1$ **then**
28 $\quad\quad\quad\quad\quad \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix} \leftarrow \begin{pmatrix}1 & 0\\ x & -x\end{pmatrix} \times \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix}$
29 $\quad\quad\quad\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}1 & 0\\ x & -x\end{pmatrix} \times \mathcal{H}$
30 $\quad\quad\quad\quad$ **else**
31 $\quad\quad\quad\quad\quad \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix} \leftarrow \begin{pmatrix}1 & 0\\ 0 & x\end{pmatrix} \times \begin{pmatrix}\mathsf{C}(x)\\ \mathsf{D}(x)\end{pmatrix}$
32 $\quad\quad\quad\quad\quad \mathcal{H} \leftarrow \begin{pmatrix}1 & 0\\ 0 & x\end{pmatrix} \times \mathcal{H}$
33 $\quad\quad \begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix} \leftarrow \mathcal{H} \times \begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix}$; $\begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix} \leftarrow \mathcal{H} \times \begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix}$
34 $\quad$ **if** $\deg(\mathsf{R}) = 0$ **then**
35 $\quad\quad$ **return** $\mathsf{U}(x)/x^{wM}$
36 $\quad$ **return** $\mathsf{V}(x)/x^{wM}$

algorithm will be running faster if it is consuming words of the polynomial to be inverted which contain only zeroes. As a result, some information on the length of the zero runs of the operand is encoded in the timing side channel. We note that, many code-based cryptoschemes, the position of the asserted coefficients of the polynomial to be inverted represent the private key of the scheme.

As a final step, Algorithm 3 returns either the value of $\mathsf{U}(x)$ or the value of $\mathsf{V}(x)$, detecting which is the variable containing the actual pseudo inverse (i.e., $a(x)^{-1}x^{wM}$), and performing an exact division by x^{wM} (i.e., a shift by M words).

3.2 Bernstein-Yang Algorithm

The last approach to polynomial inversion via improved Euclid algorithms, is the one recently proposed in [5]. Bernstein *et al.* provide a comprehensive study [5] of modular inversion and greatest common divisor (gcd) computation both for integer and polynomial Euclidean domains. In this work, we specialize the divide-et-impera strategy devised for the computation of gcds and modular inverses for polynomial rings having coefficients over a generic GF(p) onto one which only performs the computation of modular inverses of binary polynomials. We report the said specialized algorithm as Algorithm 4. A full proof of the correctness of the algorithmic approach is provided in [5], and we will omit it from the current work for space reasons. However, to provide an intuition of the inner working of Algorithm 4, we note that it is possible to obtain Algorithm 3 as a special case of it,highlighting the conceptual similarities. The key observation made in [5] is that it is possible to split the operands recursively up to a point where the operands become as small as the designer deems useful (a machine word, in our case) and then proceed to compute a portion of the linear transform over GF(2^m) which operates on the inputs to provide the modular inverse. Once such a transform is computed for the operand fragments, the transform can be recombined by means of a multiplication of matrices over GF(2^m) yielding the modular inverse. Algorithm 4 performs the operand splitting phase in the `jumpdivstep` function (defined at lines 17–25), taking two polynomials $f(x), g(x)$ of maximum degree n and the difference between their degrees δ, and picking a splitting point j (line 20). The splitting point can be chosen as any non null portion of the maximum degree n, although splitting the operands in half is likely to be optimal. We report that the intuition of optimality of splitting the operands in half was verified to be the optimal choice in our implementations. Two subsequent recursive calls to the `jumpdivstep` function are made, splitting the input polynomials at their j-th degree term (lines 21 and 34), until the maximum degree n is equal or smaller than the machine word size (w in the algorithm). When this happens ("branch taken" at line 18 of Algorithm 4), the function handling the base case of the recursion, `divstep` is invoked. `divstep` can be seen as a clever reformulation of n iterations, of the loop body of Algorithm 2, with n being the parameter taken as input by `divstep`. The computation of the loop body is decomposed into a conditional swap (lines 5–9), depending only on the value of the difference between the operand degrees δ being positive and the constant term of $g(x)$ being equal to 1, and a sequence of steps (lines 10–13) performing the correct operation between the two portions of the source operands $f(x), g(x)$ and the auxiliary values.

This approach allows a constant-time implementation using for the `if` construct Boolean-predicate operations (lines 5–9).

To compute the polynomial inverse, Algorithm 4 invokes the `jumpdivstep` function on the reflected representation of both the modulus and the polynomial to be inverted, that is it considers the polynomials S, R of the same degree of $f(x)$ and $a(x)$, respectively, obtained swapping the coefficient of the x^i term with the one of the $x^{\deg(\mathsf{S})-i}$ (resp., $x^{\deg(\mathsf{S})-i}$) one. Both polynomials, represented as degree $2m-1$ ones, adding the appropriate null terms, are processed by the `jumpdivstep` call, which returns the final difference in degrees (expected to be null), and the reflected representation of the polynomial inverse in the second element of the first row of $\mathcal{H}$.

Algorithm 4: Bernstein *et al.* (BY) Inverse [5] specialized for GF(2^m)

Input: $f(x)$ irreducible polynomial of GF(2^m), $m \geq 2$,
$a(x)$ invertible element of GF(2^m)

Output: $\mathsf{V}(x)$, such that $a(x)^{-1} \equiv \mathsf{V}(x) \in$ GF(2^m)

Data: w: machine word size

```
                                        // Base case, solved iteratively on n ≤ w
1  function divstep(n, δ, f(x), g(x)):
2      U(x) ← x^(n−1); V(x) ← 0
3      Q(x) ← 0; R(x) ← x^(n−1)
4      for i ← 0 to n − 1 do
5          if (δ > 0 ∧ g_0 = 1) then
6              δ ← −δ
7              Swap(f(x), g(x))
8              Swap(U(x), Q(x))
9              Swap(R(x), V(x))
10         δ = δ + 1
11         Q(x) ← (f_0 · Q(x) − g_0 · U(x))/x      // dropping the remainder
12         R(x) ← (f_0 · R(x) − g_0 · V(x))/x      // dropping the remainder
13         g(x) ← (f_0 · g(x) − g_0 · f(x))/x      // dropping the remainder
14     H ← ( U(x)  V(x) )
           ( Q(x)  R(x) )
15     return δ, H
16
                               // Splitting case, shortens operands until n ≤ w
17 function jumpdivstep(n, δ, f(x), g(x)):
18     if n ≤ w then
19         return divstep(n, δ, f(x), g(x))
            // any j > 0 is admissible, intuitive optimum at ⌊n/2⌉
20     j ← ⌊n/2⌉                                   // integer part of n/2
21     δ, P ← jumpdivstep(j, δ, f(x) mod x^j, g(x) mod x^j)
22     f'(x) ← P_{0,0} · f(x) + P_{0,1} · g(x)
23     g'(x) ← P_{1,0} · f(x) + P_{1,1} · g(x)
24     δ, Q ← jumpdivstep(n − j, δ, f'(x)/x^j, g'(x)/x^j)  // dropping remainders
25     return δ, (P × Q)
26
                          // Main inverse function calling recursion splitting case
27 S(x) ← mirror(f(x)), R(x) ← mirror(a(x))
28 δ, H ← jumpdivstep(2m − 1, 1, S(x), R(x))
29 V(x) ← mirror(H_{0,1})
30 return V(x)
```

3.3 Inversion with Fermat's Little Theorem

Finally, an approach to perform a constant-time implementation of the binary polynomial inversion is to employ the Fermat's little theorem. While this procedure is usually slower on a polynomial ring with a generic modulus, we obtain an efficient implementation of Fermat's method to compute inverses in $\mathbb{Z}_2[x]/(x^p-1)$, in case p is prime and $\mathrm{ord}_p(2) = p-1$ (i.e., 2 is a primitive element of GF(p)), as it is the case in the LEDAcrypt parameters. Indeed, $\mathbb{Z}_2[x]/(x^m-1), m \geq 1$ can be seen as $\prod_i \mathbb{Z}_2[x]/((f^{(i)}(x))^{\lambda_i})$, where $f^{(i)}(x)$ are the irreducible factors of $x^m - 1 \in \mathbb{Z}_2[x]$, each with its own multiplicity $\lambda_i \geq 0$. As a consequence, the number

of elements in each $\mathbb{Z}_2[x]/\left(\left(f^{(i)}(x)\right)^{\lambda_i}\right)$ admitting an inverse is: $\left|\left(\mathbb{Z}_2[x]/(f^{(i)}(x))^{\lambda_i}\right)^*\right| = 2^{\deg(f^{(i)})\cdot\lambda_i} - 2^{\deg(f^{(i)})(\lambda_i-1)}$, and the multiplicative inverse of a unitary element in $\mathbb{Z}_2[x]/(x^m - 1)$ can be obtained by raising it to the least common multiple (`lcm`) of the said quantities:
$\texttt{lcm}(2^{\deg(f^{(1)})\cdot\lambda_1}-2^{\deg(f^{(1)})(\lambda_1-1)}, 2^{\deg(f^{(2)})\cdot\lambda_2}-2^{\deg(f^{(2)})(\lambda_2-1)}, \ldots)-1$.

In our case, where $m = p$ and p is a prime number such that $\texttt{ord}_p(2)=p-1$, the polynomial x^p-1 factors as the product $(x + 1) \cdot (\sum_{i=0}^{p-1} x^i)$, and consequentially the number of invertible elements is $\left(2^{1\cdot1} - 2^{1\cdot(1-1)}\right)\cdot\left(2^{(p-1)\cdot1} - 2^{(p-1)\cdot(1-1)}\right) = 2^{p-1}-1$, while the computation of the inverse of $a(x) \in \left((\mathbb{Z}_2[x]/(x^p - 1)^*, \cdot\right)$ is obtained raising it to $2^{p-1} - 2$, i.e., $a(x)^{-1} = a(x)^{2^{p-1}-2}$.

Noting that the binary representation of $2^{p-1} - 2$ is obtained as a sequence of $p - 2$ set bits followed by a null bit (i.e., $2^{p-1} - 2 = (11, \ldots, 10)_{\texttt{bin}}$), a simple (right-to-left) *square & multiply* strategy would compute the inverse employing $p - 2$ modular squarings and $p - 3$ multiplications (i.e., $a(x)^{-1} = a(x)^2 \cdot a(x)^{2^2} \cdots a(x)^{2^{p-1}}$). However, as reported first in [8], it is possible to devise a more efficient square and multiply chain, tailoring it to the specific value of the exponent. Indeed, we are able to obtain a dedicated algorithm, reported as Algorithm 5, which computes the inversion with only $\lceil\log_2(p - 2)\rceil + \texttt{HammingWeigth}(p - 2)$ multiplications and $p - 1$ squarings. Since the squaring operations are significantly more frequent than the multiplications, it is useful to exploit the fact that polynomial squaring can be computed very efficiently in $\mathbb{Z}_2[x]/(x^p - 1)$. Indeed, two approaches are possible.

The first approach observes the fact that computing a square of an element of $\mathbb{Z}_2[x]$ is equivalent to the interleaving of its (binary) coefficients with zeroes, an operation which has linear complexity in the number of polynomial terms (as opposed to the quadratic complexity of a multiplication).

The second approach builds on the observation made in [10]: given an element $a(x) \in \mathbb{Z}_2[x]/(x^p-1)$, considering the set of exponents of its non-zero coefficient monomials S allows to rewrite $a(x)$ as $\sum_{j\in S} x^j$. It is known that, on characteristic 2 polynomial rings we have that $\forall i \in \mathbb{N}, \left(\sum_{j\in S}(x^j)\right)^{2^i} \equiv \sum_{j\in S}(x^j)^{2^i}$, thus allowing us to obtain the 2^i-th power of $a(x)$ by computing the 2^i-th powers of a set of monomials, and then adding them together. To efficiently compute the 2^i-th power of a monomial, observe that the order of x in $\mathbb{Z}_2[x]/(x^p-1)$ is p (indeed, $x^p - 1 \equiv 0$ in our ring, therefore $x^p \equiv 1$). We therefore have that $(x^j)^{2^i} = (x^j)^{2^i \bmod p}$ in $\mathbb{Z}_2[x]/(x^p-1)$. This in turn implies that it is possible to compute the 2^i-th power of an element $a(x) \in \mathbb{Z}_2[x]/(x^p - 1)$ simply permuting its coefficients according to the following permutation: the j-th coefficient of the polynomial $a(x)$, $0{\leq}j{\leq}p{-}1$, becomes the $((j \cdot 2^i) \bmod p)$-th coefficient of the polynomial $a(x)^{2^i}$. This observation allows to compute the 2^i-th power of an element in $\mathbb{Z}_2[x]/(x^p - 1)$, again, at a linear cost (indeed, the one of permuting the coefficients); moreover the permutation which must be computed is fixed, and depends only on the values p and 2^i, which are both public and fixed, thus avoiding any meaningful information leakage via the timing side channel. The authors of [10] observe that, since the required permutations, one for each value of $(2^i \bmod p)$, with $i \in \{1, \ldots, \lceil\log_2(p - 2)\rceil\}$ are fixed, they can be precomputed, and stored in lookup tables.

We note that the precomputation of such tables, while feasible, is likely to take a non-negligible amount of memory. Indeed, such tables require $p\cdot(\lceil\log_2(p{-}2)\rceil{-}1)\cdot\lceil\log_2(p)\rceil$ bits to be stored, assuming optimal packing in memory. This translates into tables ranging between 136 kiB and 2,856 kiB for the recommended prime sizes in LEDAcrypt [3]. While these table sizes are not problematic for an implementation targeting a desktop platform, more constrained execution environments, such as microcontrollers, may not have enough memory to store the full tables.

To this end, we introduce and examine a computational tradeoff, where only a small lookup table comprising the $(\lceil\log_2(p - 2)\rceil - 1)$ values obtained as $(2^i \bmod p)$, with $i \in \{1, \ldots, \lceil\log_2(p - 2)\rceil\}$, is precomputed and stored, while the position of the j-th coefficient, $0{\leq}j{\leq}p{-}1$, of $a(x)^{2^i}$ is computed via a multiplication and a (single-precision) modulus operation online, as $(j \cdot (2^i \bmod p)) \bmod p$.

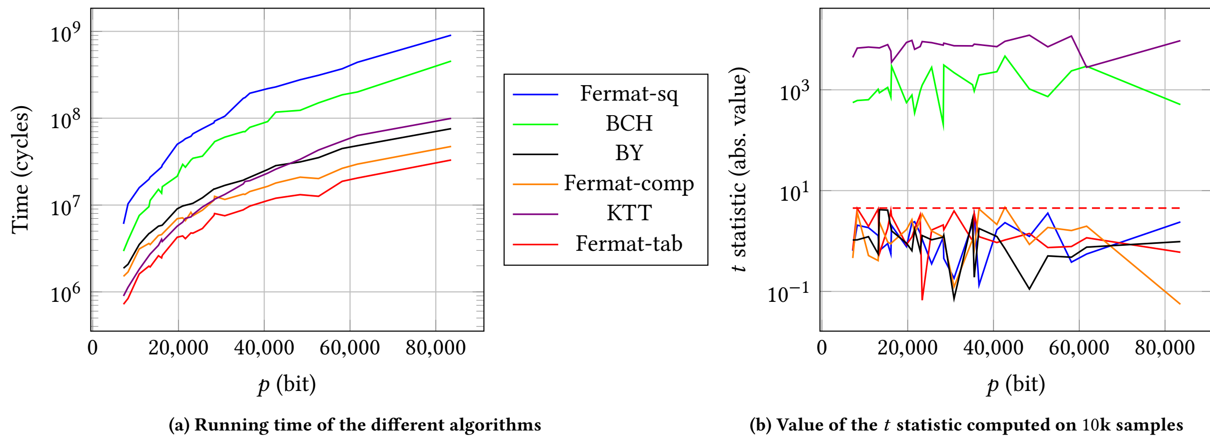
This effectively reduces the tables to a size equal to $(\lceil\log_2(p - 2)\rceil - 1) \cdot \lceil\log_2(p)\rceil$ bits, which corresponds to less than 1 kiB for all the LEDAcrypt parameters.

Finally, the authors of [10] also note that, on `x86_64` platforms, it is faster to precompute the inverse permutation with respect to the one corresponding to raising $a(x)$ to 2^i, as it allows the collection of binary coefficients of the result $a(x)^{2^i}$ that are contiguous in their memory representation. This is done determining, for each coefficient of $a(x)^{2^i}$, which was its corresponding one in $a(x)$ through the inverse permutation. The said inverse permutation maps the coefficient of the l-th degree term in $a(x)^{2^i}$, $0{\leq}l{\leq}p{-}1$, back to the coefficient of the $(l \cdot (2^{-i} \bmod p) \bmod p)$-th term in $a(x)$. We note that, also in this case, it is possible to either tabulate the entire set of inverse permutations, as suggested by [10], or simply tabulate the values of $(2^{-i} \bmod p)$ and determine each position of the permutation at hand partially online.

Algorithm 5: Inverse based on Fermat's Little Theorem

Input: $a(x)$: element of $\mathbb{Z}_2[x]/(x^p - 1)$ with a multiplicative inverse.
Output: $c(x) = (a(x)^e)^2$, and $e{=}2^{p-2}{-}1{=}(11\ldots1)_{\texttt{bin}}$. The binary encoding of e is a sequence of $p{-}2$ set bits.
Data: p: a prime such that $\mathrm{ord}_p(2) = p - 1$ (i.e., 2 is a primitive element of GF(p)).
The algorithm is instrinsically constant-time w.r.t. to the value of $a(x)$, as the control flow depends only on the value of p, which is not a secret.
Computational cost: $\lceil\log_2(p - 2)\rceil - 1 + \texttt{HammingWeight}(p - 2)$ polynomial multiplications, plus $p - 1$ squarings

1 $\texttt{exp} \leftarrow \textsc{BinEncoding}(p - 2)$
2 $\texttt{expLength} \leftarrow \lceil\log_2(p - 2)\rceil$
// scan of exp from right to left; i=0 ↔ LSB
3 $b(x) \leftarrow a(x)$ // $\texttt{exp}_0$=1 as $p{-}2$ is an odd number
4 $c(x) \leftarrow a(x)$
5 **for** $i \leftarrow 1$ **to** $\texttt{expLength} - 1$ **do**
6 $\quad c(x) \leftarrow c(x)^{2^{2^{i-1}}} \cdot c(x)$ // 2^{i-1} squarings, 1 mul.
7 $\quad$ **if** $\texttt{exp}_i = 1$ **then**
8 $\quad\quad b(x) \leftarrow b(x)^{2^{2^i}}$ // 2^i squarings
9 $\quad\quad b(x) \leftarrow b(x) \cdot c(x)$ // 1 multiplication
10 $c(x) \leftarrow (b(x))^2$ // 1 squaring
11 **return** $c(x)$



(a) Running time of the different algorithms

(b) Value of the t statistic computed on 10k samples

Figure 1: Experimental evaluation of the four modular inverse algorithms. (a) average running time in clock cycles over 10^3 computations. (b) absolute value of the t-statistic for a Student's t with two populations of 10^3 execution times for each algorithm. The dashed horizontal line highlights the upper bound of the range $[-4.5, 4.5]$, pointing out no data-dependent changes in the timing behaviors of the algorithms below it with a confidence of 99.999% ($\alpha = 10^{-5}$)

4 EXPERIMENTAL EVALUATION

We realized a self-contained implementation in C11 of all the aforementioned inversion algorithms, employing appropriate compiler intrinsics to exploit the features of the AVX2 ISA extensions available on the Intel Haswell microarchitecture, selected by the U.S.A. National Institute of Standards and Technology as the standard evaluation platform (source code available at https://doi.org/10.5281/zenodo.3760589). In particular, we employed vector Boolean instructions to speed up the required `xor` and `shift` operations which make up a significant portion of the reported algorithms.

We exploited the presence of the *carryless multiplication instruction* (`pclmulqdq`) which performs a polynomial multiplication of two 64-bit elements in a 128-bit one. We implemented the polynomial multiplication primitive applying the Toom-Cook (TC) optimized interpolation proposed in [6] until the recursive operand-splitting computation path yields operands below 50, 64-bit machine words. Then we employ a Karatsuba multiplication technique, with all multiplications involving operands below 9 machine words performed picking the optimal sequence of additions & multiplications according to [15]. We determined the number of machine words where the multiplication strategy changes (between the TC and Karatsuba ones) via exhaustive exploration of the tradeoff points.

Concerning the exponentiations to 2^i, as required by the optimized Fermat's little based inverse, we investigated the effects of the tradeoff reported in [10], where it is noted that, for small values of i, it can be faster to compute the exponentiation to 2^i by repeated squaring, instead of resorting to a bitwise permutation. Indeed, it is possible to obtain a fast squaring exploiting the `pclmulqdq` instruction which performs a binary polynomial multiplication of two, 64 terms, polynomials in a 128 terms one. While it may appear counterintuitive, this approach is faster than the use of the dedicated *Parallel Bits Deposit* (`pdep`) instruction available in the AVX2 instruction set. Indeed, the throughput obtained with the parallel bit deposit instruction is lower than the one obtained via `pclmulqdq`. This is due to the possibility of performing two `pclmulqdq` bit-interleaving starting from a 128-bit operand which can be loaded in a single instruction, as opposed to two `pdep` which need to expand a 64-bit wide operand only. This fact, combined with the low latency of the `pclmulqdq` instruction (5 to 7 cycles, depending on the microarchitecture, with a CPI of 1, while the `pdep` instruction has a 3 cycles latency, to acts on operands having half the size) provides a fast zero interleaving strategy. We also recall that, the implementation of the `pdep` instruction on AMD CPUs is quite slow (50+ cycles), and non constant-time, as it is implemented in microcode. As a consequence, the use of the `pdep` instruction would lead to a scenario where the non constant-time execution on a (AVX2 ISA extension compliant) AMD CPU can compromise the security of the implementation. Finally, we note that the `pclmulqdq` bit-interleaving strategy performs better than the one relying on interleaving by Morton numbers [14], employing the 256-bit registers available via the AVX2 ISA extension, plus the corresponding vector shift and or instructions.

We ran our experimental campaign on two AMD64 machines, an Intel Core i5-6600, and an AMD Ryzen 5 1600, both running Debian GNU/Linux 10, compiling the implementations with GCC ver. 8.3.0, enabling the architecture-specific code emission (`-march=native` compilation option) and the maximum optimization level (`-O3` option). We measured the running time of our implementations employing the `rtdscp` instruction available in the AMD64 ISA to obtain the number of clock cycles taken by a run of each algorithm, taking care of pre-heating the instruction cache running another non-timed execution of the same algorithm before. To obtain practical results, we computed the polynomial inversions picking the modulus $f(x)=x^p-1 \in \mathrm{GF}(2^p)[x]$ for all the values of p indicated in the LEDAcrypt specification [3] for use in the ephemeral key exchange LEDAcrypt-KEM-CPA, and in the IND-CCA2 LEDAcrypt-KEM with their two-iterations out-of-place decoder. Figure 1 (a) provides a depiction of the average number of clock cycles taken to

compute each polynomial inversion algorithm, computed over 10^3 runs of it, acting on a randomly drawn, invertible, polynomial on the Intel based machine. As it can be seen, the inverse computation strategy exploiting large tables and Fermat's little theorem proves to be the most efficient one for the entire range of primes which are employed in the LEDAcrypt specification. Analogous results were obtained on the AMD based machine, and are omitted for the sake of brevity. A further noteworthy point is that, if a compact memory footprint is desired, and the constant-time property is not strictly required, as it is the case for ephemeral keypairs, Algorithm 3 (KTT) provides a valid alternative to the method relying on Fermat's little theorem, with compact lookup tables (Fermat-comp in the legend) for the low range of prime sizes. We also note that an implementation of Fermat's little polynomial inverse, employing only repeated squarings via bit interleaving, instead of a permutation based computation of the exponentiation to 2^i is significantly slower (Fermat-sq in Figure 1) than all other methods. Finally, we note that, if the modular inverse is not being computed on a polynomial ring having a modulus with a peculiar structure, such as $\mathbb{Z}_2[x]/(x^p - 1)$, the benefits of performing the inversion via Fermat's little theorem may be smaller, or nullified, with respect to a completely general purpose inversion algorithm such as the one by Bernstein and Yang, which, from our analysis provides competitive performance and is constant time.

We note that the implementation of the algorithms expected to be running in constant time (Algorithm 4 (BY in the legend) and Algorithm 5 (Fermat-sq, Fermat-comp, and Fermat-tab in the legend)) relies on their implementation not containing any secret-dependent branches, nor any memory lookups whose address depends on a secret value. To provide an experimental validation of the said fact Figure 1 (b) reports the result of the validation of the constant-time property by means of a Student t-test, for each value of p, done on two timing populations of 10^3 samples each taken, one when computing inverses of random invertible polynomials, and the other when computing the inverse of the $x^2 + x + 1$ trinomial. The choice of a trinomial is intended to elicit the leakage of the position of the (very sparse) set coefficients, which represents the secret not to be disclosed in cryptoschemes such as LEDAcrypt, as the position of the few set coefficients is clustered at one end of the polynomial.

The values of the t-statistic, represented in absolute value in Figure 1 (b), for Algorithm 4 (BY in the legend) and Algorithm 5 (All Fermat variants in the legend) are within the range $[-4.5, 4.5]$ (dashed horizontal lines), in turn implying that the t-test is not detecting data-dependent changes in the timing behaviors with a confidence of 99.999% ($\alpha = 10^{-5}$). By contrast both the method by Brunner *et al.* (BCH in the legend), and the method by Kobayashi *et al.* (KTT in the legend) show t statistic values far out of the interval $[-4.5, 4.5]$, exposing (as expected) their non constant time nature.

Finally, we note that on a desktop platform, given a polynomial to be inverted $a(x)$, making the KTT and BCH algorithms immune to the timing side channel by replacing the computation of $a(x)^{-1}$ with $\lambda(x)\cdot(\lambda(x)\cdot a(x))^{-1}$, where $\lambda(x)$ is a randomly chosen invertible polynomial, will increase further their performance penalty w.r.t. the Fermat-tab solution. Indeed, this solution (known as operand blinding) requires $\lambda(x)$ to be a large, random polynomial in our case, since $a(x)$ is highly sparse, and a value of $\lambda(x)$ having few coefficients would not hide its structure.

5 CONCLUDING REMARKS

We analyzed, implemented and benchmarked a set of polynomial inversion algorithms, taking as our case study the binary polynomials over $\mathbb{Z}_2[x]/(x^p - 1)$, which are of interest due to their use in current post-quantum cryptosystems such as LEDAcrypt. Our analysis and experimental evaluation shows that, on platforms providing AVX2 ISA extensions, an optimized implementation of the Fermat's Little theorem turns out to be the best performing strategy overall, in addition to exhibiting a constant-time behavior. Furthermore, we observe that, even selecting a tradeoff between computation and memory geared towards a smaller fingerprint, the aforementioned method is still competitive in performance with variable time algorithms when the prime p sizes exceed $\approx 30,000$ bits. Finally, we provided the general inversion algorithms introduced in [5] adapted for the case of binary fields and verified its competitive performance and strong constant-time guarantees.

REFERENCES

[1] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2018. LEDAkem: A post-quantum key encapsulation mechanism based on QC-LDPC codes. In *Post-Quantum Cryptography - 9th Int.'l Conference, PQCrypto 2018, Fort Lauderdale, April 9-11, 2018 (Lecture Notes in Computer Science)*, Tanja Lange and Rainer Steinwandt (Eds.), Vol. 10786. Springer, 3–24.

[2] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2019. LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate. In *Code-Based Cryptography - 7th International Workshop, CBC 2019, Darmstadt, May 18-19, 2019 (Lecture Notes in Computer Science)*, Marco Baldi, Edoardo Persichetti, and Paolo Santini (Eds.), Vol. 11666. Springer, 11–43. DOI: http://dx.doi.org/10.1007/978-3-030-25922-8_2

[3] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2020. LEDAcrypt website. https://www.ledacrypt.org/. (2020).

[4] Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. 2019. Evaluating the Trade-offs in the Hardware Design of the LEDAcrypt Encryption Functions. In *26th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2019, Genoa, Italy, November 27-29, 2019*. IEEE, 739–742. DOI: http://dx.doi.org/10.1109/ICECS46596.2019.8964882

[5] Daniel J. Bernstein and Bo-Yin Yang. 2019. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 3 (2019), 340–398. DOI: http://dx.doi.org/10.13154/tches.v2019.i3.340-398

[6] Marco Bodrato. 2007. Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0. In *WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings (Lecture Notes in Computer Science)*, Claude Carlet and Berk Sunar (Eds.), Vol. 4547. Springer, 116–133. DOI: http://dx.doi.org/10.1007/978-3-540-73074-3_10

[7] Hannes Brunner, Andreas Curiger, and Max Hofstetter. 1993. On Computing Multiplicative Inverses in GF(2ˆm). *IEEE Trans. Computers* 42, 8 (1993), 1010–1015.

[8] Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee. 2000. Fast Implementation of Elliptic Curve Defined over GF(p^m) on CalmRISC with MAC2424 Coprocessor. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings (Lecture Notes in Computer Science)*, Çetin Kaya Koç and Christof Paar (Eds.), Vol. 1965. Springer, 57–70. DOI: http://dx.doi.org/10.1007/3-540-44499-8_4

[9] Intel Corp. 2020. Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (2020).

[10] Nir Drucker, Shay Gueron, and Dusan Kostic. 2020. Fast polynomial inversion for post quantum QC-MDPC cryptography. *IACR Cryptology ePrint Archive* 2020 (2020), 298. https://eprint.iacr.org/2020/298

[11] Katsuki Kobayashi, Naofumi Takagi, and Kazuyoshi Takagi. 2012. Fast inversion algorithm in GF(2^m) suitable for implementation with a polynomial multiply instruction on GF(2). *IET Computers & Digital Techniques* 6, 3 (2012), 180–185. DOI: http://dx.doi.org/10.1049/iet-cdt.2010.0006

[12] Robert James McEliece. 1978. A Public-Key Cryptosystem Based on Algebraic Coding Theory. *JPL Deep Space Network* 44, 42 (1978), 114–116.

[13] Harald Niederreiter. 1986. Knapsack-type cryptosystems and algebraic coding theory. *Probl. Contr. and Inf. Theory* 15 (1986), 159–166.

[14] Sean Eron Anderson. 2020. Bit Twiddling Hacks. https://graphics.stanford.edu/~seander/bithacks.html, last accessed 7 April 2020. (2020).

[15] Chen Su and Haining Fan. 2012. Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication. *Inf. Process. Lett.* 112, 12 (2012), 497–502. DOI: http://dx.doi.org/10.1016/j.ipl.2012.03.012