Constant-time BCH Error-Correcting Code

Matthew Walters and Sujoy Sinha Roy

School of Computer Science
University of Birmingham, United Kingdom
mjw553@cs.bham.ac.uk, s.sinharoy@cs.bham.ac.uk

Abstract. Decryption failure is a common phenomenon in most lattice-based public-key schemes. To reduce the rate of decryption failure, application of error correction code can be helpful. However, the literature of error correcting codes typically addresses computational performances and error correcting capabilities of the codes; their security properties are yet to be investigated. Hence, direct porting of an error correcting code in a cryptographic scheme could open new avenues to the attacks. For example, a recent work has exploited non-constant-time execution of the BCH error correcting code to reduce the security of the lattice-based public-key scheme LAC.
In this work we analyze the decoding algorithm of the BCH code and design a constant-time version of the BCH decoding algorithm. To study the computational overhead of the countermeasures, we integrated our constant-time BCH code in the reference and optimized implementations of the LAC scheme and observed nearly 1.1 and 3.0 factor slowdown respectively for the CCA-secure primitives.

Keywords: Lattice-based cryptography · Decryption failures · Error-correcting codes · Constant-time implementation

1 Introduction

In 1976 Whitfield Diffie and Martin Hellman coined Public Key Cryptography and introduced the first public key exchange protocol – the Diffie-Hellman Key Exchange [8]. Since then, vast research has been carried out into various methods for securely communicating between two parties – two of the most popular schemes appearing from this research were RSA [20,39] and Elliptic Curve Cryptography (ECC) [21], the security proofs of which rely on the computational difficulty of Integer Factorization [3] and Elliptic Curve Discrete Logarithm Problem [40] respectively.

Though these two algorithms are currently understood as safe and protecting the digital world, by using a powerful quantum computer [24] an attacker can break these algorithms in polynomial time. At present only small, not overly threatening, quantum computers have been developed by labs such as at IBM [12] and Google [14]; yet it is predicted that suitably large devices can be realized within the next 10 to 20 years [15]. As a result, we need new public-key algorithms that are safe against the threat of quantum. Post-quantum cryptography [2] is a branch of cryptography that investigates new cryptographic algorithms that cannot be broken using quantum computers, of which lattice-based schemes have gained the most attention.

Unfortunately, through the process of encryption and then decryption using these schemes, there is a small chance of decryption error. However, error-correcting codes can reduce the probability of these [6,19]. Error correcting codes are of two main categories: block codes and convolution codes. When the block-size is fixed, which is the case in our research, block codes perform

very well as they can be hard-decoded in polynomial time. Convolution codes are more appropriate when the bit streams are of arbitrary length. Interested readers may follow the work by Fritzmann et al. [11] which analyzes error-correcting capabilities and performances of several error-correcting codes for lattice-based key exchange.

Having been studied for several decades in the context of communication, direct application to cryptography would result in the correction of decryption failures (where some bits were not decrypted correctly) but also could introduce cryptographic weakness. A recent paper by D'Anvers et al. [7] introduces a technique to increase the failure rate in several (Ring/Module)-Learning With Errors (LWE) based schemes. If a scheme uses non-constant-time error-correcting code, then the attack can exploit the variations in the execution time to reduce the security of the scheme. Using this technique, D'Anvers et al. have demonstrated an attack on the LAC [25] scheme which exploits non-constant-time BCH error-correcting code. Such an attack can be prevented by making the error-correcting code constant-time.

Contributions: In this paper we analyze the BCH algorithm, which is a strong error-correcting code and has been applied in the LAC scheme, and identify the computation-steps that cause variations in the execution time. Using this observation, we modify the BCH decoding algorithm to make it constant-time. Our software implementation runs in constant time and our claim is additionally supported by leakage-detection tests. Finally, to measure the performance overhead of the countermeasures, we integrate our constant-time BCH decoding algorithm in the source code of LAC. Our source code is available from: `https://github.com/mjw553/Constant_BCH`.

The rest of the paper is organized as follows:

- In Section 2 we discuss related preliminaries and introduce BCH codes.
- In Section 3 we analyse the vulnerabiity of BCH codes to timing side-channel attacks and introduce methods to prevent against such vulnerabilities, reporting results on our timing side-channel resistant implementation.
- In Section 4 we present our implementation integrated with the LAC cryptosystem and note overheads.
- In Section 5 we conclude our work and detail future work that can be carried out in the area.

2 Preliminaries

2.1 Lattice-Based Public Key Encryption (Learning With Errors)

In 2009, Regev introduced the *Learning With Errors* (LWE) problem and a Lattice-Based Public Key Cryptosystem centred around it [38]. Almost 10 years on, LWE is still the most popular lattice-based cryptosystem to date due to efficiency and strong theoretical proof of security, following several iterations of efficiency improvements from Kawachi et al. [22] and Peikert et al. [36]. In this section we introduce the LWE problem, the conjectured hardness of which underpins the security of this popular cryptosystem. We will also introduce the cryptosystem, which has been shown to be secure under chosen plaintext attacks based on the conjectured hardness of the LWE problem.

2.2 LWE-based cryptography (over ring)

The ring-LWE problem has been very popular for constructing public-key encryption, key exchange, digital signature and homomorphic encryption schemes. Lyubashevsky, Peikert, and

Regev constructed the first public-key encryption scheme, well-known as the LPR scheme, based on the ring-LWE problem in the full version of [27]. In the LPR scheme, a discrete error distribution, typically a discrete Gaussian distribution χ_σ with a small standard deviation σ, is used to generate the secret and error polynomials. A global polynomial $a \in R_q$ is sampled from the uniform distribution. The key generation, encryption and decryption operations are as follows.

1. $\texttt{LPR.KeyGen}(a)$: Sample polynomials $r_1, r_2 \in R_q$ from $\mathcal{X}_\sigma$, calculate $p = r_1 - a \cdot r_2 \in R_q$ and output (a, p) as the public-key and r_2 as the private-key.
2. $\texttt{LPR.Encrypt}(a, p, m)$: Message m, which is a binary polynomial, is first encoded to $\bar{m} \in R_q$ by multiplying the coefficients by $(q-1)/2$. Next, three error polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext is the pair of polynomials $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + \bar{m} \in R_q$.
3. $\texttt{LPR.Decrypt}(c_1, c_2, r_2)$: Intermediate message polynomial $m' = c_1 \cdot r_2 + c_2 \in R_q$ is computed and then the original message m is recovered using a threshold decoder. The decoder simply compares the coefficients of m' with $(q/4, 3q/4)$ and outputs 1 if they are in the interval $(q/4, 3q/4)$, and as 0 otherwise.

Although efficient and the subject of much research, most LWE-based schemes have nonzero probability of decryption errors due to the presence of noise in the system. Even though appropriate setting of domain parameters can improve upon the inherent error, the effective utilization of an error-correcting code can reduce the error probability to a negligible level.

2.3 Overview of Error-Correcting Codes

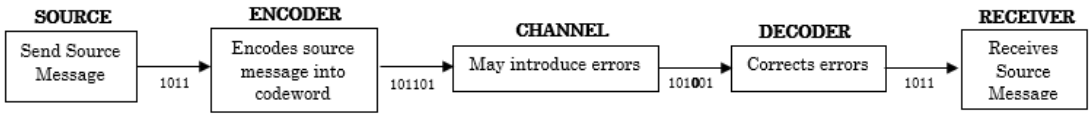


Fig. 1: Communications Channel with Error-Correcting Code (adapted from [16])

Pioneered in the 1940s by Richard Hamming and inventing the first, the *Hamming (7,4) Code* [17], in 1950, error-correcting codes have vast applications in computing, telecommunications, information theory and coding theory to moderate errors induced by a noisy communication medium. Error-correcting codes are encoded into a transmitted message as a redundant (information that is non-message critical, in this instance providing error-resistance assurances), allowing the receiver to detect and correct a limited number of errors that may occur within the message often without the need for retransmission.

Wang et al. [42] suggests that although this adds the extra cost of a fixed, higher forward channel bandwidth, this therefore means that error-correcting codes are ideal for use when retransmissions are costly. For example, Reed-Solomon codes [37] were used in the Voyager's space explorations [33].

The design of the error-correcting code determines the threshold on the number of errors that can be corrected, its *error-correcting capability*, and therefore different codes should be used in different situations. Generally, *strong* codes induce greater redundancy in data that must be transmitted using the bandwidth available, in-turn reducing the rate at which message bits are transmitted whilst improving the received ratio of signal to noise.

Types of Error-Correcting Code Error-correcting codes can be split into two main categories, *block codes* and *convolutional codes*:

- Block codes [6,28] work on blocks of fixed length and are generally decodedable in polynomial time due to their block length. Notable examples are Reed-Solomon, Bose-Chaudhuri-Hocquenghem (BCH) and Low-Density Parity-Check (LDPC). In the next subsection we explain these in further detail.

- Convolutional codes [6,9] work on bit streams of arbitrary length and are normally decoded using the Viterbi algorithm [10]. There is also a convolutional variation of LDPC.

Classical block codes can normally be decoded using *hard-decision algorithms*. These algorithms decide for every input/output signal whether it should correspond to a 1 or 0 bit. Contrastingly, convolutional codes can normally be decoded using *soft-decision algorithms*. These take analogue signals as input and as a result offer greater error-correcting capability [31].

In comparison to classical block codes, modern variations (such as LDPC) lack specific error-correcting guarantees and are more often evaluated by their bit-error rates.

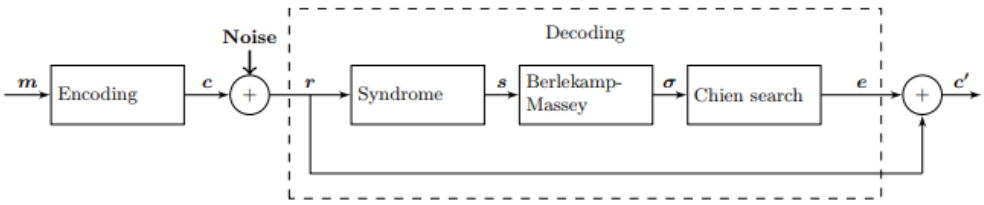


Fig. 2: BCH error correction using the Berlekamp-Massey Decoding algorithm [11]

2.4 BCH codes

For their conciseness and clarity, the description of BCH codes below is heavily influenced by the Han lecture notes [18], themselves drawing from the explanations of Costello et al. [6]. For a greater and well-presented description of the mathematical primitives referenced (such as Finite Fields or Primitive Elements), we draw the attention of the reader to the work of Huffman and Pless [19]. For work that has been done on the efficient implementation of software-based BCH, we draw the attention of the reader to the work of Cho and Sung [5].

Definition: Due to the nature of our work, we only consider binary BCH codes. For any positive integers $m \leq 3$ and $t < 2^{m-1}$ (error-correcting capability), there exists a binary *t-error-correcting* BCH code, denoted $BCH(n, k, d)$, with the following parameters:

Block length: $n = 2^m - 1$
Number of parity-check bits: $n - k \leq mt$
Minimum distance: $d_{min} \geq 2t + 1$

where n is the size of the codeword, k is the length contribution in the codeword of the message, and d is the minimum distance (d_{min}) between codewords.

Let α be a *primitive element* in $GF(2^m)$. The *generator polynomial* $g(x)$ of the code of length 2^{m-1} is the lowest-degree polynomial over $GF(2)$ which has $\alpha, \alpha^2, \ldots, \alpha^{2t}$ as its roots. Therefore, $g(\alpha^i) = 0$ for $1 \leq i \leq 2t$ and $g(x)$ has $\alpha, \alpha^2, \ldots, \alpha^{2t}$ and their conjugates as roots.

Let $\phi_i(x)$ be the *minimal polynomial* of α^i, where $\phi_1(x)$ is the *primitive polynomial* from which all elements can be generated. Then $g(x)$ must be the least common multiple of $\phi_1(x), \phi_2(x), \ldots, \phi_{2t}(x)$, i.e.

$$g(x) = LCM\{\phi_1(x), \phi_2(x), \ldots, \phi_{2t}(x)\} \tag{1}$$

However, if i is an even integer, it can be expressed as $i = i^{'}2^l$ where i' is odd and $l > 1$. Then $\alpha^i = (\alpha^{i^{'}})^{2l}$ is a conjugate of $\alpha^{i^{'}}$. Hence $\phi_i(x) = \phi_{i'}(x)$, and therefore

$$g(x) = LCM\{\phi_1(x), \phi_3(x), \ldots, \phi_{2t-1}(x)\} \tag{2}$$

This property is useful for efficiency, reducing the number of calculations required for BCH decoding, which we will see in a later subsection.

Since α is a primitive element, the BCH codes defined are usually called primitive (or 'narrow-sense') BCH codes.

Table 1: Representations of $GF(2^4)$ [34]

Power	**Calculation**	**Numeric value**
$a = x$	x	0010(2)
$a^2 = x \times x$	x^2	0100(4)
$a^3 = x \times x \times x$	x^3	1000(8)
$a^4 = a \times a^3$	$x^4 = x + 1$(using the GP$x^4 + x + 1 = 0$)	0011(3)
$a^5 = a \times a^4$	$x^5 = x^2 + x$	0110(6)
$a^6 = a \times a^5$	$x^6 = x^3 + x^2$	1100(12)
$a^7 = a \times a^7$	$x^7 = x^4 + x^3 = x^3 + x + 1$	1011(11)
$a^8 = a \times a^8$	$x^8 = x^4 + x^2 + x = x^2 + 1$	0101(5)
$a^9 = a \times a^9$	$x^9 = x^3 + x$	1010(10)
$a^{10} = a \times a^9$	$x^{10} = x^4 + x^2 = x^2 = x^2 + x + 1$	0111(7)
$a^{11} = a \times a^{10}$	$x^{11} = x^3 + x^2 + x$	1110(14)
$a^{12} = a \times a^{11}$	$x^{12} = x^4 + x^3 + x^2 = x^3 + x^2 + x + 1$	1111(15)
$a^{13} = a \times a^{12}$	$x^{13} = x^4 + x^3 + x^2 + x = x^3 + x^2 + 1$	1101(13)
$a^{14} = a \times a^{13}$	$x^{14} = x^4 + x^3 + x = x^3 + 1$	1001(9)
$a^{15} = a \times a^{14}$	$x^{15} = x^4 + x = 1$	0001(1)

Encoding: To transmit the whole codeword, data symbols are first transmitted and then followed by the parity symbols. This can be represented as

$$C(x) = x^{n-k}M(x) + P(x), \tag{3}$$

where $M(x)$ is the original source message and $P(x)$ is the parity, given

$$P(x) = x^{n-k}M(x) \bmod g(x) \tag{4}$$

where $g(x)$ is the generator polynomial.

Decoding: Decoding of BCH codes occurs over three steps:

1. Syndrome generation of $2t$ syndromes from the received codeword.
2. Error-locator polynomial, Λ, calculation.
3. Solve Λ, the roots of which point to the error locations, and correct the received codeword.

Syndrome Generation: Let

$$\mathbf{r}(x) = r_0 + r_1 x + r_2 x^2 + \cdots + r_{n-1} x^{n-1} \tag{5}$$

be the *received codeword*. We expect $\mathbf{r}(x)$ to be a combination of the actual codeword $\mathbf{v}(x)$ and error $\mathbf{e}(x)$ from a noisy channel,

$$\mathbf{r}(x) = \mathbf{v}(x) + \mathbf{e}(x). \tag{6}$$

The *syndrome* is a $2t$-tuple,

$$\mathbf{S} = (S_1, S_2, \ldots, S_{2t}), \tag{7}$$

where

$$S_i = \mathbf{r}(\alpha^i) = r_0 + r_1 \alpha^i + \cdots + r_{n-1} \alpha^{(n-1)i} \tag{8}$$

for $1 \leq i \leq 2t$.
Dividing $\mathbf{r}(x)$ by the minimal polynomial $\phi_i(x)$ of α^i, we have

$$\mathbf{r}(x) = \alpha^{\mathbf{i}}(x)\phi_i(x) + \mathbf{b}_i(x), \tag{9}$$

where $\mathbf{b}_i(x)$ is the remainder,

$$\mathbf{b}_i(x) = \mathbf{r}(x) \bmod \phi_i(x). \tag{10}$$

Since $\phi_i(\alpha^i) = 0$, we have

$$S_i = \mathbf{r}(\alpha^i) = \mathbf{b}_i(\alpha^i). \tag{11}$$

Finally, as $\alpha^1, \alpha^2, \ldots, \alpha^{2t}$ are roots of each code polynomial, $\mathbf{v}(\alpha^i) = 0$ for $1 \leq i \leq 2t$.

If all elements of S are 0, there are no errors and therefore decoding can stop here.
As described in the definition of BCH codes, if i is an even integer it can be expressed as $i = i'2^l$ where i' is odd and $l > 1$. We can therefore compute the odd-numbered syndromes by calculating S_i for $i = 1, 3, 5, \ldots, 2t-1$, and then easily compute the even-numbered syndromes from the relation $S_{2i} = S_i^2$ for a more efficient approach to syndrome generation.

Error-Location Polynomial Calculation: Any method for solving S is a decoding algorithm for BCH codes. Developed in 1967 [1,30], the Berlekamp-Massey method is an efficient algorithm for such a purpose. In this section we will describe both the original Berlekamp Massey algorithm and a more efficient variant, the Simplified Inversion-less Berlekamp-Massey Algorithm (SiBMA), as described by Costello et al. [6]. The simplified version is outlined in Algorithm 1 and is considered advantageous because of its reduction in iterations by one-half when considering binary BCH codes, iterating for t instead of $2t$.

We first describe the original Berlekamp Massey algorithm. The algorithm builds up the error-locator polynomial by requiring that its coefficients satisfy a set of equations known as the

Algorithm 1 SiBMA for Calculating Error-Locator Polynomials

INPUT: **S[t]** = S[1],S[2],...,S[t] // array of calculated syndromes
INPUT: t // total number of syndromes
OUTPUT: C[ ](x) // error-location polynomial (elp)

/ Initialise arrays */*
C[t+1](x) // current elp with each iteration
D[t+1] // current discrepancy value with each iteration
L[t+1] // corresponding degree of C[ ](x) with each iteration
UP[t+1] // value of 2(i-1)-L[i] with each iteration

/ Assign Initial Values */*
C[0](x) = 1, D[0] = 1, D[1] = 1, L[0] = 0, L[1] = 0, UP[0] = -1
dpVal = 1, lpVal = 0, upVal = -1, upMax = -1, p = -1

/ Main Algorithm */*
for `i = 1; i <= t+1; i++` **do**
  UP[i] = 2·(i-1) - L[i]
  if `D[i] == 0` **then**
    C[i+1](x) = C[i](x)
    L[i+1] = L[i]
  else
    / Find another row, p, prior to ith row such that D[p] != 0 and UP[p] has largest value */*
    for `j = 0; j < i; j++` **do**
      if `D[j] != 0 && UP[j] > upMax` **then**
        upMax = UP[j], lpVal = L[j], dpVal = D[j], upVal = UP[j], p = j
      end if
    end for
    C[i+1](x) = C[i](x) + D[i] · $(\text{dpVal})^{-1} \cdot x^{2\cdot((\text{i-1})-\text{p})}\cdot$ C[p](x)
    L[i+1] = max(L[i] , L[p] + 2 · ((i-1)-p))
  end if
  D[i+1] = S[2i + 1] + $\sum_{j=1}^{\text{L[i+1]}}$ C[j](x) · S[2i+1-j]
end for
return C[ ](x)

Newton identities [32].

In the 0th state, we initialise all variables for their respective arrays. We then start with the minimum-degree polynomial $C[1](x)$ which satisfies the first Newton identity [32]. We then check if the coefficients also satisfy the second Newton identity by calculating the *discrepancy*, representing the difference between the identity and the current representation $C[2](x)$. If it does (the discrepancy is 0), we precede iterating over all syndromes, otherwise we alter $C[2](x)$ so that it does, and then precede until $C[2t+1](x)$ is obtained and all $2t$ Newton Identities are satisfied. We then take the error-location polynomial $C(x)$ to be the state of $C[\,](x)$ after the $2t+1$st iteration.

Let the minimal-degree polynomial determined at the kth step of iteration, $1 \leq k \leq 2t+1$, whose coeffients satisfy the first $k-1$ Newton's identities for $L[k]$ presumed errors be

$$C[\,](x) = 1 + C[1](x) + C[2](x^2) + \cdots + C[L[k]-1](x^{L[k]-1}) + C[L[k]](x^{L[k]}) \tag{12}$$

At iteration k the discrepancy between the next representation, $k+1$, and the respective Newton identity is calculated as

$$D[k] = S[k-1] + \Sigma_{j=1}^{L[k]} C[j](x) \cdot S[k-1-j] \tag{13}$$

When updating for iteration $k+1$, we consider:

1. If $D[k] == 0$, then $C[k+1](x) = C[k](x)$ and $L[k+1] = L[k]$.
2. If $D[k] \neq 0$, find another iteration p prior to the kth iteration such that $D[k] \neq 0$ and the number $p - L[p]$ has the largest value. Then

$$C[k+1](x) = C[k](x) + D[k] \cdot D[p]^{-1} \cdot X^{k-1-p} \cdot C[p](x) \tag{14}$$

$$L[k+1] = max(L[k], L[p] + k - 1 - p) \tag{15}$$

and in either case:

$$D[k+1] = S[k] + \Sigma_{j=1}^{L[k+1]} C[j](x) \cdot S[k-j] \tag{16}$$

For SiBMA, we make minor alterations for updating with each successive iteration: When updating for iteration $k+1$, we consider:

1. If $D[k] == 0$, then $C[k+1](x) = C[k](x)$ and $L[k+1] = L[k]$.
2. If $D[k] \neq 0$, find another iteration p prior to the kth iteration such that $D[k] \neq 0$ and the number $2p - L[p]$ has the largest value. Then

$$C[k+1](x) = C[k](x) + D[k] \cdot D[p]^{-1} \cdot X^{2((k-1)-p))} \cdot C[p](x) \tag{17}$$

$$L[k+1] = max(L[k], L[p] + 2(k-1) - p) \tag{18}$$

and in either case:

$$D[k+1] = S[2k+1] + \Sigma_{j=1}^{L[k+1]} C[j](x) \cdot S[2k+1-j] \tag{19}$$

Solving the error-locator polynomial and correcting errors: We must solve the error-location polynomial, $\mathbf{\Lambda}(\mathbf{x})$, for its roots to determine the locations of errors in the received codeword. These roots are the inverse of the error-locations, and as we are considering the binary case, we simply flip the bits in these positions.

It is common to use Chien Search [4] to solve the polynomial. A brute-force substitution method, the algorithm examines whether α^i is a root of $\mathbf{\Lambda}(\mathbf{x})$ for $i = 1, \ldots, n-1$. That is to say that for all primitive elements $\alpha^1, \alpha^2, \ldots, \alpha^{n-1}$, we select all α^i for which

$$\mathbf{\Lambda}(\alpha^{\mathbf{i}}) = 0 \tag{20}$$

3 Analysing the vulnerability of BCH codes to timing attacks

In this section we explore the vulnerability of BCH codes to timing side-channel attacks. A variable-time algorithm executes variably with different inputs, and may be exploited by an attacker to gleam information about either the input provided or the specifics of a program. If an variable-time decoding algorithm is coupled with even a strong cryptographic method (such as LWE Lattice-Based cryptoschemes) it will weaken the cryptographic security of the resulting scheme. We start this section by introducing some algorithmic changes that can be made to translate a variable-time algorithm into a timing-attack resistant equivalent which executes in a constant number of CPU cycles, specifically looking at SiBMA. We will consider the algorithm itself and also note features that may appear in implementations. We then present our experimental results into the translation of the SiBMA algorithm into a constant-time equivalent.

3.1 Translating SiBMA

Constant-execution FOR Loops We consider the two `FOR` loops in the SiBMA algorithm:

```
16: for i = 1; i <= t+1; i++
...
23: for j = 0; j <  i; j++
```

As the first loop on line `16` iterates from 1 to $t+1$ (the error correcting capability of the code), this will iterate constant number of times irrespective of input length. As the second loop on line `23` iterates from 0 to the iterator of the first loop, i, this will also run irrespective of input length, and given there are no terminating statements inside such as `break` or `exit`, the algorithm already has loops which iterate constant number of times.

Although these are presented as constant-time in the algorithm given and there are no terminating statements in the algorithm, implementations of the BCH decode algorithm [41,25] may attempt to reduce the iterations based on input for efficiency reasons and as a result would be vulnerable to timing side-channel attacks.

More precisely, many loops are of the form `for(i=0;i<in_length;i++)`, iterating over a specific input. Algorithms (and programs) often only iterate the minimum number of iterations necessary, so if input is variable this would result in the loop iterating a variable amount dependant on input, therefore allowing an attacker to learn about the input provided.

To address this we must first fix the iteration length, usually the maximum number of iterations the loop could ever be and ensure that there are no early-termination statements such as `break` or `exit`.

As a cautionary note, loops that need to be altered to always run their maximum number of iterations must be careful to not perform any actions such as variable assignment with a variable value whilst executing an iteration of the loop that would otherwise not be executed to ensure program correctness is maintained. This can be done in a constant-time approach using a *bounds determiner* to indicate whether a variable assignment should persist through the given iteration (i.e. whether there should be an effect to running the iteration). For example, an assignment to a variable in a non-constant approach may be `v = 2;`, where in a constant approach it becomes

```
v = 2 * determiner + v * !determiner;
```

where v evaluates to 2 if *determiner* evaluates to 1 (indicating the iteration should persist effects), and the original value v otherwise (no effect on the overall algorithm). The value of the determiner itself is calculated in constant time by checking whether the current iteration value is accepted by the loop condition initially in place, for example:

```
int j = 4;
for(i = 0; i < j; i++)
...
```

will become

```
int max_value = 6;
int j = 4;
for(int i = 0; i < max_value; i++)
    determiner = ((i - j) & mask) >> 31;
    ...
```

where the difference between the current iteration i and the original bound j is determined, and then bitwise AND'd with a *mask* representing the largest negative number for the representation of numbers being used inside the loop, finally bit-shifting to the most significant bit to determine if the difference is positive or negative, representing whether the condition is met (with a value of 1) or not (with a value of 0).

Constant-time Branching Statements SiBMA has two IF statements which span the lines 18 to 30:

```
18: if D[i] == 0 then
...
21: else
...
24:     if D[j] != 0 && UP[j] > upMax then
        ...
26:     end if
...
30: end if
```

Many processors will perform optimisation methods such as *Branch Prediction* to improve execution performance of branching statements. On top of this, branches of unequal operation counts

will leak timing, and therefore execution, information, opening the algorithm to timing attacks and resulting in crytographic weakness. Therefore these must be handled to ensure that the same number of operations are carried out irrespective of the evaluation of a statement's condition.

Similar to constant `FOR` loops, the operations inside an `IF` statement should always be executed, however their effect's persistence can be controlled using an execution determiner. In the case of SiBMA, the two statements and can controlled by two determiners which evaluate in a similar way to before.

We now note four key features to altering the branching statements of the SiBMA algorithm which must be considered:

1. As the branching statement on line `18` is an `IF-ELSE` statement, care must be taken to evaluate the execution determiner to `True` before the `Else` if the condition was met, but after the `Else` if it was not.
2. Assignments of the form `A[i+1] = A[i]`, such as those on lines `19` and `20`, and of the form `A[i+1] = b`, such as those on lines `28` and `29`, where the iterator (`i`) is increasing and indexing is to unassigned elements, one can simply evaluate to the correct value or 0.
3. `IF` statements inside of `IF` statements, such as the one starting on line `24` inside of the statement starting on line `18`, are conditional on the surrounding statement and should be adjusted.
4. When variables are being overridden, such as those on line `25`, they must take either the new value or their existing value dependant on the execution determiner value.

In Algorithm 17SiBMA for Calculating Error-Locator Polynomialsalgorithm.1_if_constantSiBMA_-if_constantesent a variant of the SiBMA algorithm with constant-time branching statements.

Uniform Array Access The SiBMA algorithm has many array accesses. Although arrays are stored in contiguous memory blocks, accessing arrays based on input data evaluations resulting in different indexes being accessed (*data-dependent access*) can result in timing variations.

For example, the array accesses on line `17`, `UP[i] = 2· (i-1) - L[i]`, is only dependent on the overarching iterator i and as such executes in a predictable manner which would not leak information to an attacker and therefore does not need to be altered. However, the array access `S[2i+1-j]` must be altered on line `31` as it is dependant on `j`, itself dependent on `L[i+1]`, itself dependent on the value of `D[i]`, itself dependent on the values of `S[ ]`, which is data-dependent.

To translate a variable-time array access into a constant execution time array access we use *blinded array access* and *full table scan*. Abstractly, instead of accessing only one element in the array, we access all elements but only return the element with which we were looking to access initially. This way there is no timing difference with every access to the array. Implementation wise, we cycle through each possible index and determine (through XOR'ing current iteration and required index, then OR'ing the bits of the result) whether we want to return the specific value. We then perform a bitwise AND operation with the array value and add it to a sum. At the end of the iteration, the sum value will store the correct required value. This is given in Algorithm **??**_accessblind_access

Algorithm 2 SiBMA with constant-time branching statements

```
14: ...
15: /* Main Algorithm */
16: for i = 1; i <= t+1; i++ do
17:    UP[i] = 2·(i-1) - L[i]
18:    flag1 = ((0 - D[i]) & mask) >> 31;          // determiner equiv. to if D[i] == 0
19:    C[i+1](x) = C[i](x) · flag1;                // as C[i+1] is unassigned can set to value or 0
20:    L[i+1] = L[i] · flag1;                      // as L[i+1] is unassigned can set to value or 0
21:    /* Here we look for !flag1 to perform operation as we are in the Else condition */
22:    /* Find another row, p, prior to ith row such that D[p] != 0 and UP[p] has largest value */
23:    for j = 0; j < i; j++ do
24:       flag2 = ((0 - D[i]) & mask) >> 31;       // determine whether D[j]!=0
25:       flag2 = flag2 · ((upMax - UP[j]) & mask) >> 31;
26:                                                // determine whether D[j]!=0 && UP[j]>upMax
27:       flag2 = flag2 · !flag1                   // flag2 is dependent on flag1 condition
28:       upMax = UP[j] · flag2 + upMax · !flag2   // set values if flag evaluates, otherwise set to current
29:       lpVal = L[j] * flag2 + lpVal · !flag2
30:       dpVal = D[j] · flag2 + dpVal · !flag2
31:       upVal = UP[j] · flag2 + upVal · !flag2
32:       p = j * flag2 + p · !flag2
33:    end for
34:    C[i+1](x) = (C[i](x) + D[i] · (dpVal)^{-1} · x^{2·((i-1)-p)}· C[p](x)) · !flag1;
35:                                                // as C[i+1] is unassigned can set to value or 0
36:    L[i+1] = (max( L[i] , L[p] + 2 · ((i-1)-p) )) · !flag1
37:                                                // as L[i+1] is unassigned can set to value or 0
38:    D[i+1] = S[2i+1] + ∑_{j=1}^{L[i+1]} C[j](x) · S[2i+1-j]
39: end for
40: ...
```

Algorithm 3 Blinded Array Access

```
1: INPUT: index                                  // index of array value
2: INPUT: size                                   // length of array
3: INPUT: arr[ ]                                 // array to access
4: OUTPUT: sum                                   // value of array at index position
5:
6: /* Initialise Variables */
7: temp1, temp2, j, sum = 0
8: one = 1                                       // representation of 1 in array type
9: /* Main Algorithm */
10: for j = 0; j < size; j++ do
11:    temp1 = j ⊕ index                         // XOR potential index with required index
12:    temp2 = check_bits(temp1)                 // function to determine if any bits are 1
13:    temp1 = (temp2 & 1)- one                  // temp1=(0) if temp2 = 1, otherwise temp1 = (1)
14:    sum = sum + (temp1 & arr[j])              // temp1 is 0 in all cases except j = index
15: end for
```

3.2 Brief Comments on Overhead

It is easy to see that running an algorithm for the maximum number of operations for every input will induce overhead. Whilst some techniques can be implemented to reduce the factor of overhead, there is inherently more computational effort required for a constant-time implementation. This therefore represents a fine trade-off between computational efficiency and computational security.

As a consequence we also implement BCH with weaker security but a more efficient approach by replacing the computationally taxing Blinded array access for data-dependent lookup table accesses with Full Table Scan array access, first accessing every element sequentially in the array and then reading the intended index. This limits access timing variations due to the table being completely read into the cache before accessing the desired index. We present this alternative method in Algorithm **??**_accesscache_access

Algorithm 4 Full Table Scan

```
1: INPUT: index                          // index of array value
2: INPUT: size                           // length of array
3: INPUT: arr[ ]                         // array to access
4: OUTPUT: val                           // value of array at index position
5:
6: /* Initialise Variables */
7: val = 0, i;
8: /* Main Algorithm */
9: for i = 0; i < size; i++ do
10:    val = arr[i]                      // Read all elements into cache
11: end for
12: val = arr[index]                     // Access required element from cached array
13: return val
```

3.3 Evaluating Using T-Tests

To evaluate our implementations we will utilise the statistical *t-test*. Proposed by Welch in 1947 [43], the test is used to determine the statistical degree of equivalence of two classes' means, and therefore failure of such a test implies that two groups of data are susceptible to timing information leakage. We set a upper limit t-score of 4.5 [13] to indicate a program runs in constant execution cycles for distinct input parameter (i.e. number of fixed errors) and any t-test value exceeding this indicates variable-time.

For each distinct potential number of errors, limited by the error-correcting capability t of the code, we repeat the process of generating a random message and adding the appropriate number of errors in random positions 10,000 times. This then gives us $t+1$ samples of 10,000 measurements, which we can perform pairwise t-tests on to determine whether there is a significant difference in means between any of the fixed-error classes. If there is not, this indicates that the execution timings for the decode process run suitably equivalent independent of the number of errors in the codeword, and therefore a successful constant-time implementation.

However, as t-tests are statistical tests they work on the principle of confidence intervals. For example, $\alpha = 95\%$ indicates that we want to be at least 95% confident in our result to

suggest that there is a significant result to our test (in this case, that there *is* a significant difference between the two different classes' means, and as such the procedure is not a constant-time implementation). If run independent t-tests between two classes of data, this is acceptable. However, if we wish to compare three classes of data (for example, 1, 2 and 3 fixed errors) and run three t-tests at 95% confidence, our confidence level actually becomes $\alpha = 0.95*0.95*0.95 = 0.857$ and therefore we would incorrectly suggest that there is a significant difference 1 in 7 times, rather than 1 in 20 times. To remedy this and compare all of our independent results together we can run an ANalysis Of VAriance (ANOVA) test [23] to compare all means simultaneously and not reduce the confidence level, looking for a resulting `P` value > 0.05 to confirm a successful constant-time implementation.

3.4 Experimental Results

Performing our experiments on an Intel i5-6500 desktop CPU we run our implementation of BCH[1] using parameters `BCH(511,264,59)` and with $t = 29$. We test both the Blinded data-dependent lookup table access and Full Table Scan data-dependent lookup table access implementations, collecting 10,000 CPU timings for performing the BCH decoding for each amount of fixed error, $e = 0, 1, 2, ..., 29$, on encoded randomly generated messages with randomly placed errors. As we are measuring raw CPU cycles we notice 'spikes' in our data either when a machine is performing another user process in parallel or when our test program switches from one operation to another, for example from input preparation to gathering measurements. To limit the variance incurred from the first issue we run on a machine with no other user process running, and to limit the second issue we increase our measurements by 20% and discard the first 2,000 measurements pre t-test.

Performing t-tests between all possible pairwise combinations of fixed errors for both the blinded and full table scan implementations we find that no pair has a t-score greater then 4.5, indicating that both implementations perform in constant-time irrespective of the number of fixed errors present in a received codeword.

Furthermore, performing ANOVA on both sets of data we report test results of `[F(29,299970) = 0.747, P = 0.834]` for Blinded and `[F(29,299970) = 0.429, P = 0.997]` for Full Table Scan implementations. As for both results `P` > 0.05, we conclude that there is no significant difference between our results, and as such our implementations perform in a constant-time.

To ensure logical correctness we also perform our experiments on a bare metal ARM Cortex M4 processor residing on a STM32F4-discovery board from STMicroelectronics. With no over-arching OS and data-cache, process and memory-access fluctuations are eliminated. We perform our experiment in a similar fashion to the ones previously, except that we take 100 measurements per fixed error instead of 10,000. This is due to the consistency offered by ARM and also the slowdown as a result of less hardware resources and slower UART communication.

Similar to before, performing t-tests between all possible pairwise combinations of fixed errors for both the blinded and full table scan implementations we find that no pair has a t-score greater then 4.5, indicating that both implementations perform in constant-time on ARM irrespective of the number of fixed errors present in a received codeword.

Performing ANOVA on both sets of data we report test results of `[F(29,2970) = 0.187, P = 1]` for Blinded and `[F(29,2970) = 0.015, P = 1]` for Full Table Scan implementations. As

[1] Source code available from: `https://github.com/mjw553/Constant_BCH`

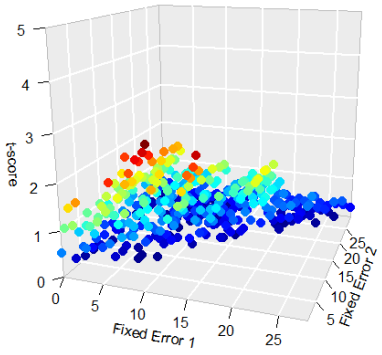
for both results $P > 0.05$, we conclude that there is no significant difference between our results, and as such our implementations perform in a constant-time on ARM.

Finally, as there is no data-cache on the targeted ARM platform, there are no timing variations for array accesses for attackers to exploit. Therefore we also test on ARM our implementation by removing all table access countermeasures that we implemented. As with all other tests, all t-tests fall below 4.5 and we obtain a non-significant ANOVA result of `[F(29,2970) = 0.097, P = 1]`.

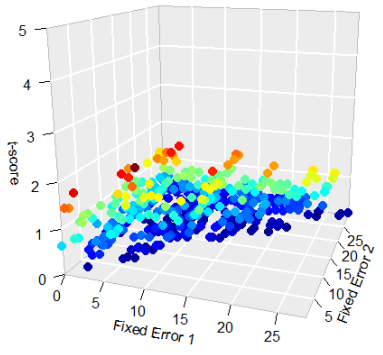
Table 2: T-score range for BCH implementations

		T-Score		
Device	**Implementation**	**Minimum**	**Average**	**Maximum**
Desktop	*Blinded*	0.0002	0.5748	2.0297
	Full Table Scan	0.0017	0.4844	1.7474
Arm	*Blinded*	0.0042	0.3570	1.0363
	Full Table Scan	0	0.0980	0.3644
	No Table Countermeasures	0	0.2335	0.9857

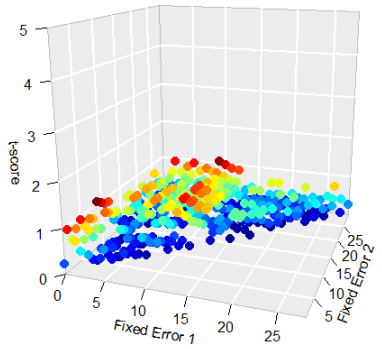
Fig. 3: T-tests for BCH implementations



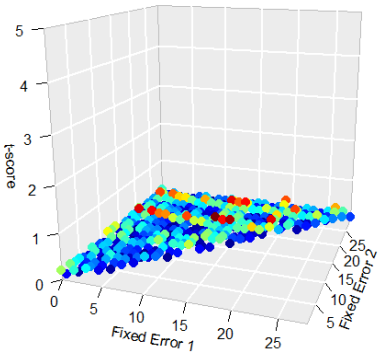
(a) Desktop Blinded



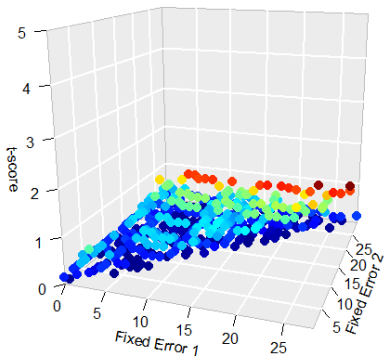
(b) Desktop Full Table Scan



(c) ARM Blinded

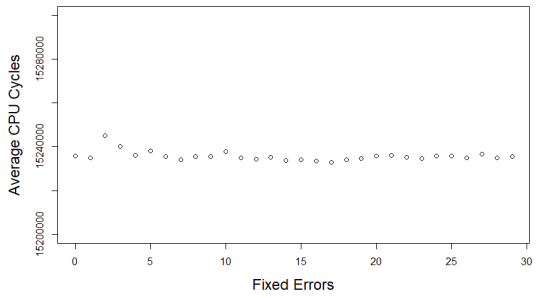


(d) ARM Full Table Scan

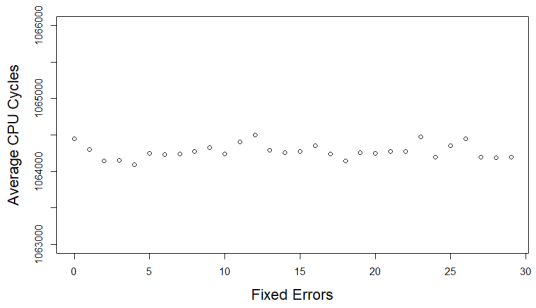


(e) ARM with direct table access

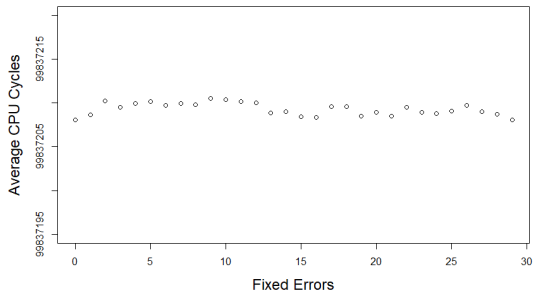
Fig. 4: Average CPU Cycles repeats of BCH implementations



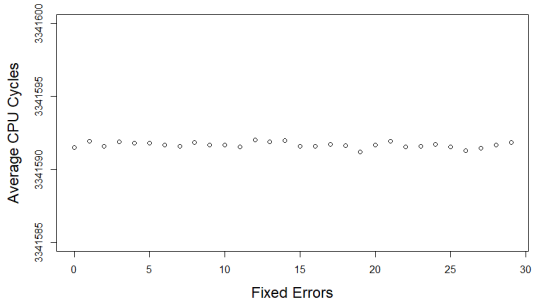
(a) Desktop blinded table scan



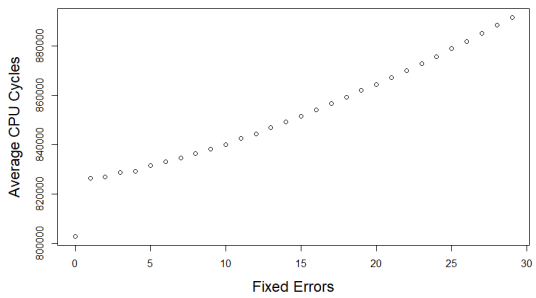
(b) Desktop full table scan



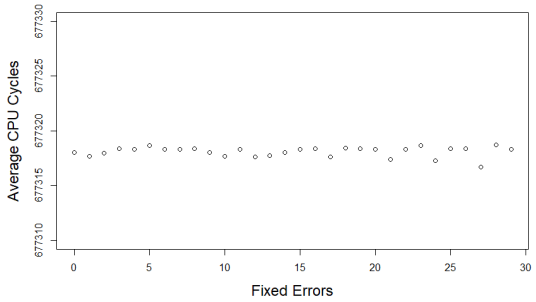
(c) ARM blinded table scan



(d) ARM full table scan



(e) Reference with no countermeasure



(f) ARM with direct table access

4 Case Study: LAC

A submission for NIST's Round Two Post-Quantum Cryptography (PQC) Standardization [35], LAC [25,26][2] has a C implementation of Lattice-Based Cryptography using four related LWE-based public-key cryptographic primitives (LAC.CPA, LAC.KE, LAC.CCA, LAC.AKE). We gather CPU cycle counts for both the Reference and Optimised variants of the LAC implementation and then integrate our constant-time BCH implementation to determine the performance costs of our changes to secure BCH. LAC offers three choices of security parameter (128, 192, 256) which satisfy various security categories in the NIST documentation and use various different parameters for BCH. We test our implementation of BCH using LAC-128 which uses the definition `BCH(511,385,29)` and an error-correction capability of 20 errors. We test both our security-focused blinded-access and our weaker but more efficiency full table scan access implementations with the provided Reference and Optimised variants of LAC. Across all four primitives with the blinded-access implementation, we observe an average 6-factor slowdown against the Reference implementation and an average 77-factor slowdown against the Optimised implementation. Across all four primitives with the full table scan access implementation, we observe an average 1-factor slowdown against the Reference implementation and an average 4.5-factor slowdown against the Optimised implementation.

Table 3: Slowdown factors for LAC primitives with blinded and full table scan implementations

		Reference			**Optimised**		
	Primitives	**Minimum**	**Average**	**Maximum**	**Minimum**	**Average**	**Maximum**
Blinded	*CPA.DEC*	7.6	7.7	7.7	92.3	108.1	122.0
	CCA.DEC	3.7	3.7	3.7	38.8	41.1	43.0
	KE.DEC	7.6	7.7	7.7	89.0	103.4	115.6
	AKE.DEC	4.7	4.8	4.9	52.0	56.5	60.0
Full Table Scan	*CPA.DEC*	1.3	1.3	1.3	5.0	5.9	6.6
	CCA.DEC	1.1	1.1	1.1	2.7	2.9	3.0
	KE.DEC	1.3	1.3	1.3	4.9	5.7	6.3
	AKE.DEC	1.1	1.2	1.2	3.3	3.6	3.8

5 Conclusion and Future Work

We have shown that constant time implementations of BCH decoding are possible, securing schemes such as LAC which have been shown to be weakened by the inclusion of BCH. We have considered overhead as an issue with the constant-time transformation of existing implementations and have discussed methods to reduce these, unfortunately often at the expensive of security.

Potential future works include a more in-depth algorithm analysis to further improve efficiency and reduce overhead as this is a first study on the implementation of constant-time BCH codes further optimizations may be possible, and an investigation into our implementation's security against Power Side-Channel Attacks [29].

[2] At the time of writing we use version 2 of the submitted code, originally downloaded 06/01/2019 and checked 11/02/2019.

References

1. Berlekamp, E.R.: Algebraic coding theory. McGraw-Hill series in systems science, World Scientific, New Jersey, revised edition edn. (2015)
2. Bernstein, D.J., Dahmen, E., Buchmann, J. (eds.): Post-quantum cryptography. Springer (2009), oCLC: 551314023
3. Chang, S.Y.: Prime Factorization Problem - RSA Algorithm, https://www.coursera.org/lecture/asymmetric-crypto/prime-factorization-problem-ITnwE
4. Chien, R.: Cyclic decoding procedures for Bose- Chaudhuri-Hocquenghem codes. IEEE Transactions on Information Theory **10**(4), 357–363 (Oct 1964). https://doi.org/10.1109/TIT.1964.1053699
5. Cho, J., Sung, W.: Efficient Software-Based Encoding and Decoding of BCH Codes. IEEE Transactions on Computers **58**(7), 878–889 (Jul 2009). https://doi.org/10.1109/TC.2009.27, http://ieeexplore.ieee.org/document/4782950/
6. Costello, D.J., Lin, S.: Error Control Coding: Fundamentals and Applications. Prentice Hall (1982)
7. D'Anvers, J.P., Vercauteren, F., Verbauwhede, I.: On the impact of decryption failures on the security of lwe/lwr based schemes. Cryptology ePrint Archive, Report 2018/1089 (2018), https://eprint.iacr.org/2018/1089
8. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (Nov 1976). https://doi.org/10.1109/TIT.1976.1055638, http://ieeexplore.ieee.org/document/1055638/
9. Felstrom, A.J., Zigangirov, K.S.: Time-varying periodic convolutional codes with low-density parity-check matrix. IEEE Transactions on Information Theory **45**(6), 2181–2191 (Sep 1999). https://doi.org/10.1109/18.782171
10. Forney, G.D.: The viterbi algorithm. Proceedings of the IEEE **61**(3), 268–278 (1973)
11. Fritzmann, T., Poppelmann, T., Sepulveda, J.: Analysis of Error-Correcting Codes for Lattice-Based Key Exchange p. 22
12. Galeon, D.: IBM just announced a 50-qubit quantum computer (Nov 2017), https://futurism.com/ibm-announced-50-qubit-quantum-computer
13. Gilbert Goodwill, B.J., Jaffe, J., Rohatgi, P., et al.: A testing methodology for side-channel resistance validation. In: NIST non-invasive attack testing workshop. vol. 7, pp. 115–136 (2011)
14. Greene, T.: Google reclaims quantum computer crown with 72 qubit processor (Mar 2018), https://thenextweb.com/artificial-intelligence/2018/03/06/google-reclaims-quantum-computer-crown-with-72-qubit-processor/
15. Greenemeier, L.: How Close Are We–Really–to Building a Quantum Computer?, https://www.scientificamerican.com/article/how-close-are-we-really-to-building-a-quantum-computer/
16. Hamilton, K.: Polynomial Codes Over Certain Finite Fields (2000), http://epubs.siam.org/doi/10.1137/0108018
17. Hamming, R.W.: Error detecting and error correcting codes. Bell System technical journal **29**(2), 147–160 (1950)
18. Han, Y.: BCH Codes, http://web.ntpu.edu.tw/~yshan/BCH_code.pdf
19. Huffman, W.C., Pless, V.: Fundamentals of Error-Correcting Codes. Cambridge University Press, Cambridge (2003). https://doi.org/10.1017/CBO9780511807077, http://ebooks.cambridge.org/ref/id/CBO9780511807077
20. Jonsson, J., Kaliski, B.: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. Tech. Rep. RFC3447, RFC Editor (Feb 2003). https://doi.org/10.17487/rfc3447, https://www.rfc-editor.org/info/rfc3447
21. Kapoor, V., Abraham, V.S., Singh, R.: Elliptic curve cryptography. Ubiquity **2008**(May), 7 (2008)
22. Kawachi, A., Tanaka, K., Xagawa, K.: Multi-bit Cryptosystems Based on Lattice Problems. In: Okamoto, T., Wang, X. (eds.) Public Key Cryptography – PKC 2007, vol. 4450, pp. 315–329. Springer

Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71677-8_21, http://link.springer.com/10.1007/978-3-540-71677-8_21
23. Kim, H.Y.: Analysis of variance (ANOVA) comparing means of more than two groups. Restorative Dentistry & Endodontics **39**(1), 74–77 (Feb 2014). https://doi.org/10.5395/rde.2014.39.1.74, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3916511/
24. Ladd, T.D., Jelezko, F., Laflamme, R., Nakamura, Y., Monroe, C., O'Brien, J.L.: Quantum Computing. Nature **464**(7285), 45–53 (Mar 2010). https://doi.org/10.1038/nature08812, http://arxiv.org/abs/1009.2267, arXiv: 1009.2267
25. Lu, X., Liu, Y., Zhang, Z., Jia, D., Xue, H., He, J., Li, B.: LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. Tech. Rep. 1009 (2018), http://eprint.iacr.org/2018/1009
26. luxianhui007: Lattice based pke and ke scheme. Contribute to luxianhui007/LAC development by creating an account on GitHub (Oct 2018), https://github.com/luxianhui007/LAC, original-date: 2018-10-12T00:45:52Z
27. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Advances in Cryptology – EUROCRYPT 2010. Lecture Notes in Computer Science, vol. 6110, pp. 1–23. Springer Berlin Heidelberg (2010)
28. MacWilliams, F.J., Sloane, N.J.A.: The theory of error correcting codes. North-Holland mathematical library ; v. 16, North-Holland Pub. Co. ; sole distributors for the U.S.A. and Canada, Elsevier/North-Holland, Amsterdam ; New York : New York (1977)
29. Mantel, H., Schickel, J., Weber, A., Weber, F.: How secure is green it? the case of software-based energy side channels. In: European Symposium on Research in Computer Security. pp. 218–239. Springer (2018)
30. Massey, J.: Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory **15**(1), 122–127 (Jan 1969). https://doi.org/10.1109/TIT.1969.1054260, http://ieeexplore.ieee.org/document/1054260/
31. Mathuranathan: Hard and Soft decision decoding (Dec 2009), https://www.gaussianwaves.com/2009/12/hard-and-soft-decision-decoding-2/
32. Mead, D.G.: Newton's Identities. The American Mathematical Monthly **99**(8), 749–751 (1992). https://doi.org/10.2307/2324242, https://www.jstor.org/stable/2324242
33. NASA: Voyager - Two Voyagers Taught Us How to Listen to Space (Aug 2017), https://voyager.jpl.nasa.gov/news/details.php?article_id=50
34. Neuberger, G., Kastensmidt, F.G.d.L., Reis, R.: An Automatic Technique for Optimizing Reed-Solomon Codes to Improve Fault Tolerance in Memories. IEEE Design & Test of Computers **22**, 50–58 (2005). https://doi.org/10.1109/MDT.2005.2, doi.ieeecomputersociety.org/10.1109/MDT.2005.2
35. NIST: Post-Quantum-Computing Round 1 Submissions (Jun 2019), https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions
36. Peikert, C., Vaikuntanathan, V., Waters, B.: A Framework for Efficient and Composable Oblivious Transfer. In: Wagner, D. (ed.) Advances in Cryptology – CRYPTO 2008. pp. 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
37. Reed, I.S., Solomon, G.: Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics **8**(2), 300–304 (1960), https://www.jstor.org/stable/2098968
38. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 34 (2009)
39. Rivest, R.L., Shamir, A., Adleman, L.M.: Cryptographic communications system and method (Sep 1983), https://patents.google.com/patent/US4405829/en
40. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing **26**(5), 1484–1509 (Oct 1997). https://doi.org/10.1137/S0097539795293172, http://arxiv.org/abs/quant-ph/9508027, arXiv: quant-ph/9508027

41. Torvalds, L.: Linux kernel source tree. Contribute to torvalds/linux development by creating an account on GitHub (Feb 2019), https://github.com/torvalds/linux, original-date: 2011-09-04T22:48:12Z
42. Wang, C., Sklar, D., Johnson, D.: Forward error-correction coding. Crosslink **3**(1), 26–29 (2001)
43. Welch, B.L.: The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. Biometrika **34**(1/2), 28–35 (1947). https://doi.org/10.2307/2332510, https://www.jstor.org/stable/2332510