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. Error-correcting codes can be useful in reducing decryption failure rate of several lattice-based and code-based public-key encryption schemes. Two schemes, namely LAC and HQC, in NIST's round 2 phase of its post-quantum cryptography standardisation project use the strong error-correcting BCH code. However, direct application of the BCH code in decryption algorithms of public-key schemes could open new avenues to the attacks. For example, a recent attack exploited non-constant-time execution of BCH code to reduce the security of LAC.
In this paper we analyse the BCH error-correcting code, identify computation steps that cause timing variations and design the first constant-time BCH algorithm. We implement our algorithm in software and evaluate its resistance against timing attacks by performing leakage detection tests. To study the computational overhead of the countermeasures, we integrated our constant-time BCH code in the reference and optimised implementations of the LAC scheme as a case study, and observed nearly 1.1 and 1.4 factor slowdown respectively for the CCA-secure primitives.

Keywords: Post-quantum cryptography · Decryption failures · Error-correcting codes · Constant-time implementation

1 Introduction

With large-scale quantum computers likely to be realised in the next 20 years [11], current widely implemented public-key schemes such as RSA and Elliptic-Curve Cryptography (ECC) are soon theorised to be obsolete as the computationally *hard* problems underpinning their security may be solvable in a post-quantum scenario [24], completely compromising the schemes. Lattice-based problems such as Learning with Errors (LWE) [22] or Learning with Rounding (LWR) [1], including their ring and module variants, have been very popular for constructing efficient public-key, post-quantum secure, cryptosystems. The LWE assumption underpinning their security states that it is computationally infeasible, theoretically even in a quantum scenario, to distinguish with non-negligible advantage between samples from a LWE distribution $A_{s,\chi}$ and samples drawn uniformly from $\mathbb{Z}_q^n \times \mathbb{Z}_q$. For secret $s \in \mathbb{Z}_q^n$, the LWE distribution $A_{s,\chi}$ consists of tuples $(a, a \cdot s + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, where $a \in \mathbb{Z}_q^n$ is uniformly-random and $e \in \mathbb{Z}$ is an error coefficient from the error distribution χ.

Through the process of encryption and then decryption, cryptosystems based on LWE or LWR problems have non-zero decryption failure rate. To reduce this failure rate, typically two mechanisms are used in the literature. Most schemes use 'reconciliation' mechanisms that involve transmission of additional information so that the two communicating parties can agree on a common key with higher success probability. An alternative approach is to use an error-correcting code to reduce the failure probability [6, 14] to a negligible level. Lattice-based schemes Round5 [23] and ThreeBears [12] use simple repetition codes for error correction, whereas LAC [16] uses strong error-correcting BCH code due to its aggressive design choices.

Error-correcting codes have applications in code-based public-key cryptography. For example, Hamming Quasi-Cyclic (HQC) [21] uses Tensor Product Codes (BCH codes and repetition codes) during its decryption operation.

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.

Having been studied for several decades in the context of communication, direct application to cryptography would result in the correction of decryption failures but also could introduce cryptographic weakness to otherwise theoretically strong and secure schemes. Recently D'Anvers et al. [7] devised a method for *failure boosting* decryption failures in LWE-based schemes, and by using a timing side-channel built a statistical model around the amount of information leaked by these failures. Subsequently, an estimation of the secret could be derived from the model which could be used to construct an easier problem for an attacker to solve to eventually break the entire secret. As a result the authors showed that an attacker could significantly reduce the security of LWE-based schemes that utilise error-correcting codes by using a timing side-channel found in BCH.

By eliminating the timing side-channel from BCH, both LAC and HQC can be made resistant to timing attacks. We propose the first constant-time implementation of the BCH error-correcting algorithm. We start with identifying the steps that lead to non-constant-time execution and design countermeasures by introducing algorithmic tweaks.

Contributions: The aim of this paper is to implement a constant-time variant of BCH, which can be placed as the error-correcting component of a cryptographic scheme to harden the error-correcting element from attacks which aim to exploit timing side-channels, and in turn strengthen the overall scheme.

This paper analyses the Simplified Inversion-less Berlekamp-Massey Algorithm (SiBMA) [6] for error-location polynomial computation often used during the decoding process of BCH. By analysing this algorithm for features which result in non-constant-time execution, and extrapolating observations to the entire encoding and decoding processes of BCH, we can develop countermeasures which will result in constant time execution.

The developed software implementation runs in a constant time and this claim is additionally supported by leakage-detection tests. BCH code has been applied in the LAC scheme and as such the LAC scheme is used in this research as a case study for analysing the performance overhead resulting from implementing the countermeasures. Source code is available from `https://github.com/mjw553/Constant_BCH`.

Organisation: The rest of the paper is organised as follows: Sec. 2 has the related mathematical background on BCH codes. Sec. 3 analyses the vulnerability of BCH codes to timing side-channel attacks and introduces methods to prevent against such vulnerabilities. Sec. 4 evaluates the proposed countermeasures. The performance overheads are discussed in Sec. 5. The final section concludes this work.

2 Background

In this section we describe the BCH error-correcting code. For their conciseness and clarity, the description of BCH codes below is heavily influenced by the Han lecture notes [13], themselves drawing from the explanations of Costello and Lin [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 [14]. 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].

2.1 Definition:

Due to the nature of our work, we only consider binary BCH codes. For any positive integers $m \geq 3$ 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.

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

2.2 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.

2.3 Decoding

Decoding of BCH codes occurs over three major steps as follows.

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.

These steps are described in detail in the following part of this section.

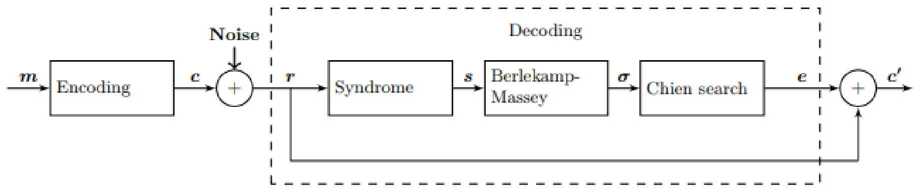


Figure 1: BCH error-correction using the Berlekamp-Massey Decoding algorithm [9]

2.3.1 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,

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

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 $\alpha^1, \alpha^2, \dots, \alpha^{2t}$ 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 $i = 1, 3, 5, \dots, 2t-1$, and then easily compute the even-numbered syndromes from the relation $S_{2i} = S_i^2$ for a more efficient approach to syndrome generation.

2.3.2 Error-location polynomial calculation

Any method for solving S is a decoding algorithm for BCH codes. Developed in 1967 [2, 19], the Berlekamp-Massey method is an efficient algorithm for solving the syndromes, S. 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

Algorithm 1 SiBMA for calculating error-locator polynomials [6]

```
INPUT: S[2t] = S[1],S[2],...,S[2t]        // array of calculated syndromes
INPUT: t                                  // error-correcting capability of the code
OUTPUT: $C^{t+1}(x)$                      // error-location polynomial (elp) after t iterations

/* Initialise arrays */
$C^{[t+2]}[t+1](x)$                       // current elps with each iteration, t+1 coefficients
                                          // must initialise $t+2$ of these
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 */
$C^0[0] = 1$, $C^1[0] = 1$, D[0] = 1, D[1] = S[1], L[0] = 0, L[1] = 0, UP[0] = -1
upMax = -1, p = -1, pVal = 0

/* Main Algorithm */
for i = 1; i <= t; 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], p = j, pVal = j
            end if
        end for
        if p == 0 then
            pVal = $\frac{1}{2}$
        end if
        $C^{i+1}(x) = C^i(x) + (D[i] \cdot (D[p])^{-1} \cdot x^{2\cdot((\text{i-1})-(\text{pVal-1}))} \cdot C^p(x))$
        L[i+1] = max( L[i] , L[p] + 2 · ((i - 1) - (pVal-1)) )
    end if
    if i != t then
        $D[i+1] = S[2i+1] + \sum_{j=1}^{\text{L[i+1]}}(C^{i+1}[j] \cdot S[2i+1-j]\ )$
    end if
end for
return $C^{t+1}(x)$
```

binary BCH codes, iterating for t instead of $2t$. All arrays are zero-indexed except for syndromes, S, which is one-indexed for consistency with definitions above.

Considering first the original Berlekamp Massey algorithm, it builds up the error-locator polynomial by requiring that its coefficients satisfy a set of equations known as the Newton identities [20]. In the initial (0th) state, we initialise all values in their respective arrays. We then start and iterate until we calculate the minimum-degree polynomial $C^2(x)$ which satisfies the first Newton identity. 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 $C^3(x)$. If it does (the discrepancy is 0), we precede iterating over the syndromes, otherwise we alter $C^3(x)$ 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 $C(x)$ (represented as $\Lambda(x)$ in the other algorithms) to be the state after the $2t$-th iteration.

Let the minimal-degree polynomial determined at the end of kth step of iteration, $1 \leq k \leq 2t$, whose coefficients satisfy the first k Newton identities for $L[k+1]$ presumed errors be

$$C^{k+1}(x) = C^{k+1}[0] + C^{k+1}[1] \cdot x + C^{k+1}[2] \cdot x^2 + \cdots + C^{k+1}[L[k+1]] \cdot x^{L[k+1]} \tag{12}$$

where $C^{k+1}[0] = 1$.

At the end of iteration k, excluding the last iteration, the discrepancy between the current representation $C^{k+1}(x)$ and the $k+1$'st Newton identity, represented by $C^{k+2}(x)$, is calculated as

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

When updating during iteration k, 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 iteration p prior to the kth iteration, $0 \leq p < k$, such that $D[p] \neq 0$ and $(p-1) - L[p]$ has the largest value. Then

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

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

and in either case, if $k \neq 2t$ (protecting against out-of-bounds access to S):

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

For SiBMA, we make minor alterations for updating with each successive iteration, now only iterating for $1 \leq k \leq t$. When updating during iteration k, 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 iteration p prior to the kth iteration, $0 \leq p < k$, such that $D[p] \neq 0$ and (if $p == 0$, set $pVal = \frac{1}{2}$, or $pVal = p$ otherwise) $2(pVal - 1) - L[p]$ has the largest value. Then

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

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

and in either case, if $k \neq t$ (protecting against out-of-bounds access to S):

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

2.3.3 Solving the error-locator polynomial and correcting errors

We must solve the error-location polynomial, $\mathbf{\Lambda(x)}$, 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 $\mathbf{\Lambda(x)}$ 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 and securing the SiBMA algorithm

In this section we explore the vulnerability of BCH codes to timing side-channel attacks. We start by introducing typical algorithmic changes which can translate a variable-time algorithm into a timing-attack resistant, constant-time equivalent. SiBMA is used as a case study for this work as it contains many of the algorithmic features which are typically attributed to variable-time algorithms and present in the other parts of the BCH process. This work can then be extrapolated to the entire encoding and decoding BCH process. Programming features which appear in common implementations which should be handled to ensure an effective, secure, implementation are then also noted. Finally, a timing side-channel resistant variant of SiBMA, *ConSiBMA*, is then presented.

3.1 Analysis and design

3.1.1 Constant-execution FOR loops

We consider the two `FOR` loops in the SiBMA algorithm:

```
17: for i = 1; i <= t; i++
...
24: for j = 0; j <  i; j++
```

As the first loop on line `17` iterates from 1 to t (the error-correcting capability of the code), this will iterate a constant number of times irrespective of input length. As the second loop on line `24` 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 a constant number of times.

Although these are presented as constant time in the SiBMA algorithm and there are no terminating statements, implementations [16] may attempt to reduce the iterations based on the input, such as by using a loop of the form `for(i=0;i<in_length;i++)` (iterating over an input), for efficiency reasons and as a result would be vulnerable to timing side-channel attacks.

To address this we must first fix the iteration length, setting it to the theoretical maximum value based on the algorithm, and ensure that there are no early-termination statements such as `break` or `exit`. 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 = x;`, where in a constant approach it becomes

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

where `v` evaluates to `x` if `determiner` evaluates to `1` (indicating the iteration should persist effects), and the original value *v* otherwise (no affect on the overall algorithm). The value of the determiner itself can be calculated in a constant time by checking whether the current iteration value is accepted by the loop condition which was 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).

3.1.2 Constant-execution branching statements

SiBMA has four branching statements which span the lines `19` to `37`. Many processors will perform optimisation methods such as *Branch Prediction* to improve execution performance of branching statements. Also, branches of unequal operations will execute differently, leaking execution information to an observer. Therefore these must be handled to ensure that the same number of operations are carried out and execution is consistent irrespective of the evaluation of a statement's condition.

Cross-Copying vs Conditional Assignment methods
Two methods for translating non-constant to constant-execution branching statements are considered, Cross-Copying and Conditional Assignment. Both approaches attempt to obfuscate to an observer how a branching statement executes. They were chosen as they formed part of the evaluation conducted by Mantel and Starostin on timing side-channel countermeasure techniques (of which Conditional Assignment evaluated the best) [18] and were later considered, and shown to be successful, by Mantel et al. as countermeasures against power side-channel attacks [17].

Cross-Copying balances out the branching statement to ensure that an equal number of operations are executed irrespective of which branch was taken. In the case of a one-branch statement (i.e. an `IF` statement with no `ELSE`), a branch can be created to balance out the original. This is normally implemented by using a *dummy* variable so that no effect of extra operations propagate through the function.

Unfortunately this method is susceptible to *fault attacks*, which have been contextualised in both hardware [3] and software [15] environments. An attacker may inject a program with 'faults' in an attempt to see how various parts of the program affect the output. If a fault were to be injected into a dummy variable the overall output would not be affected and as a result an attacker may determine the use of dummy variables. As a result we propose Conditional Assignment as a viable, tested [17, 18], alternative.

Conditional Assignment removes the branching statement entirely, and instead utilises a determiner similar to before to control whether a particular value gets updated or not.

Four key features to altering the branching statements of the SiBMA algorithm which must be considered can now be noted:

1. As the branching statement on line `19` 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]` or `A[i+1] = b`, such as the one on lines `18` and `21`, where the iterator (`i`) is increasing and indexing to unassigned elements, one can simply evaluate to the correct value or 0.
3. Branching statements (nested) inside of branching statements, such as the one starting on line `25` inside of the statement starting on line `19`, are conditional on the surrounding statement and should be adjusted.
4. When variables are being overwritten, such as those on line `26`, they must take either the new value or their existing value dependant on the execution determiner value.

3.1.3 Uniform array access

The SiBMA algorithm has many array accesses. Although arrays are stored in contiguous memory blocks, *data-dependent access* (accessing arrays based on the evaluation of input data resulting in different indexes) can result in timing variations.

For example, the array accesses on line `18`, `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 `36` 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 one which executes in a constant time we consider two methods - *blinded array access* and *full table scan.*

Blinded array access

Instead of accessing only one element in the array, we access all elements and perform arithmetic operations using the elements, but only return the element with which we were looking to access initially. This way there is no timing difference when a specific index is requested. Implementation wise, for every possible index we mask it to a value of all 0s if it isn't the required index, or all 1s if it is. We then AND this mask with the value read from the array at the current index. As a result, at the end of iterating, `sum` will contain the value of the array at the required index. This is given in Algorithm 2. Similarly for writing (Algorithm 3), we alter the array value access to instead write to the indexed position either the new value or original, controlled by a determiner. In the algorithms, $\{x\}$ denotes the setting all bits up to the highest currently set bit to the value x (*set_bits* function), whereas $[x]$ denotes the setting of all bits in the word to x.

Full table scan

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, a more efficient variant of BCH is implemented, but with resultant weaker security guarantees, by replacing the computationally taxing blinded array access

Algorithm 2 Blinded array reading

```
INPUT: index                              // index of array value
INPUT: size                               // length of array
INPUT: arr[ ]                             // array to access
OUTPUT: sum                               // value of array at index position

/* Initialise Variables */
xorVal, anyOnes, flipExpand, j, sum = 0
one = 1                                   // representation of 1 in array type
/* Main Algorithm */
for j = 0; j < size; j++ do
    xorVal = j $\oplus$ index                      // XOR potential index with required index
    anyOnes = set_bits(xorVal)            // anyOnes = {0} if j = index, {1} otherwise
    flipExpand = (anyOnes & 1) - one      // flipExpand = [1] if anyOnes = {0}, [0] otherwise
    sum = sum + (flipExpand & arr[j])     // flipExpand = [0] except if j = index
end for
return sum
```

Algorithm 3 Blinded array writing

```
INPUT: index                              // index of array value
INPUT: size                               // length of array
INPUT: arr[ ]                             // array to access
INPUT: val                                // value to place into array

/* Initialise Variables */
xorVal, anyOnes, flipExpand, j, sum = 0
one = 1                                   // representation of 1 in array type
/* Main Algorithm */
for j = 0; j < size; j++ do
    xorVal = j $\oplus$ index                      // XOR potential index with required index
    anyOnes = set_bits(xorVal)            // anyOnes = {0} if j = index, {1} otherwise
    flipExpand = (anyOnes & 1) - one      // flipExpand = [0] if anyOnes = {1}, [1] otherwise
    arr[j] = (arr[j] & $\sim$ flipExpand) + (val & flipExpand)
                                          // flipExpand = {0} except if j = index
end for
```

with full table scan array access. This method first accesses every element sequentially in an array and then reads/writes at the intended index. This limits access timing variations due to the table being completely read into the cache before accessing the desired index. This alternative algorithm is presented for reading arrays in Algorithm 4.

The weakness in security is attributed to an attacker being able to manipulate the cache. As an example, let A be the array we wish to access, and i be the index of the value we wish to retrieve from A. If we read the entirety of A into the cache and then access $A[i]$, time to access will be uniform for all possible i, provided the entirety of A can be read into the cache. However, if an attacker manages to replace what is in the cache in the location where $A[i]$ was stored, it will trigger a *cache miss* which results in the value of $A[i]$ needing to be fetched from other, slower, areas of memory, which will be noticeably slower than if another, non replaced, index was requested. Therefore an attacker can determine the specific indexes being read by the program and can as such learn information about the program input if these are input-dependant.

Algorithm 4 Full table scan read

```
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
```

3.2 Implementation considerations

The open-source BCH library by Parrot for Developers, made available on GitHub [8], was used as a base implementation to apply the devised countermeasures to and evaluate their success. It uses the SiBMA variant for error-locator polynomial calculation and is integrated into the LAC scheme as its method of error correction. This conveniently allows for LAC to be used as a case study for the security improvements to BCH codes and requires few adjustments to LAC-specific code. This section notes some considerations which had to be made to translate the library into a constant-time equivalent.

3.2.1 Structures

The reference base implementation utilises a large structure, `bch_control`, to store the current state of the program at various stages of execution. It stores key attributes (such as `elp` representing the error-locator polynomial once calculated, `ecc_bits` storing the size of the error-correcting code and `a_pow_tab/a_log_tab` storing table-lookups for Galois field values), and 'scratch' spaces (such as `cache` storing temporary values used during Chien Search and `poly_2t` representing temporary polynomials used in Berlekamp Massey).

For security, large structures are typically avoided due to reduced programmatic control. Constant integers such as `t` and `ecc_bits` are replaced with global constants `MAX_ERROR` and `ECC_BITS` respectively. Variables such as `elp` and `syn` are then locally, statically, allocated in the appropriate locations with appropriate scope to avoid the need to include them entirely in the structure. Once all of these changes have been appropriately made, the use of the `bch_control` structure can be removed entirely from the code and as a result greater control over the implementation is achieved.

3.2.2 Large data-dependent look-up arrays

During initialisation of the original `bch_control` structure, the large arrays `a_pow_tab` and `a_log_tab`, used to store values of α^i and $log_\alpha(x)$ respectively, were populated at run-time by the method `build_gf_tables`. Unlike other data-dependent array accesses, these arrays can be more efficiently handled then by just making them global and accessing with blinded access due to the values for each index already being known. This section explores two potential methods for handling these large look-up arrays more efficiently than standard blinded access - bit-sliced access, and blinded access with array packing.

Bit-sliced access

As all values (excluding the duplicate `1`'s in `a_pow_tab` and duplicate `0`'s in `a_log_tab`, which can be explicitly handled) are unique, a truth table can be generated to map how index bits contribute to output bits. Therefore the output can be presented as a function of the bit-represented index and this can be mapped into a set of bitwise operations which will run in constant time. However, as we treat the output as a function of input, it is difficult to parallelise and make more efficient than a basic implementation.

Blinded access with array packing

In the original implementation, the arrays `a_pow_tab` and `a_log_tab` both contain 512 elements. If using blinded array access this will take a considerable amount of time to cycle through and there will be significant overhead. As indexes are not dependent on one-another we can utilise *Bit Packing* to pack arrays. Modern 64-bit machines are capable of performing operations on values up to 64-bits, so programs can be optimised to make use of this, performing a single operation on a 64-bit value rather than lots of operations on values represented by few bits. Bit Packing provides a mechanism for this by using bit-shifting to represent multiple small-bit values in one 64-bit value.

As both arrays are indexed by a maximum value of 511 (as C is 0-indexed), all values stored are below 512, and these can be represented by 9 bits, we can therefore pack floor(64/9) = 7 indexed values into one larger 64-bit value, therefore representing the values of the original array at the 7 indexes in one value (see Figure 2). 512/7 = 73.1... means that we will have 73 full 64-bit values each representing the value of 7 indexes and one with only $512 - 73 * 7 = 1$ value represented.

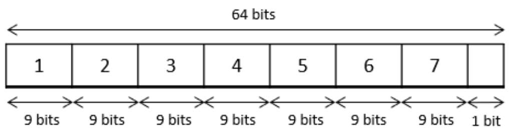


Figure 2: Packing 9-bit values into 64-bit value

For a section of seven indexes, starting at position j, we can therefore pack the values which they index in the array a in the following way:

$$\begin{aligned} &a[j] + (a[j+1] \ll 9) + (a[j+2] \ll (9*2)) + (a[j+3] \ll (9*3)) \\ &+(a[j+4] \ll (9*4)) + (a[j+5] \ll (9*5)) + (a[j+6] \ll (9*6)) \end{aligned} \tag{21}$$

We can then pack the index required in a similar manner to determine which index in the new array of 64-bit values the packed original array value should be extracted from. The algorithm then proceeds similar to normal blinded array access until the correct bit-packed value is located. It must then be unpacked to get the specific value. Before unpacking the result will only contain 1s in one of the seven 9-bit segments of the 64-bit value, representing the value to be extracted, the rest will be 0s. Therefore this 64-bit value can be successively split and OR'd with the opposite half to end up efficiently with the correct original value which is to be returned.

As seven chunks of 9-bits can be packed into one 64-bit value, the use of Bit Packing in this way on large arrays can therefore theoretically improve the efficiency of the blinded array access to read these look-up tables by seven times.

Comparison of techniques

In Figure 3 we compare the CPU cycles taken to return the value at random indexes in the `a_pow_tab` look-up table for all three implementations. Each method is tested with the same index for each test, repeating for 10,000 tests. As expected the packed variant of the blinded array access implementation performs seven times faster than the original. Although bit-slicing is faster than the original blinded implementation, it is approximately five times slower than the packed blinded implementation. Consequently, packed blinded array access will be used for large look-up arrays in the implementation. Notable are five accesses for the packed blinded implementation 'spiking' at around 7000 CPU cycles. We attribute this to the testing process being momentarily interrupted by other system processes and as such these anomalies should not be considered a weakness in the access method itself and we discuss mitigations in Section 4.2.1.

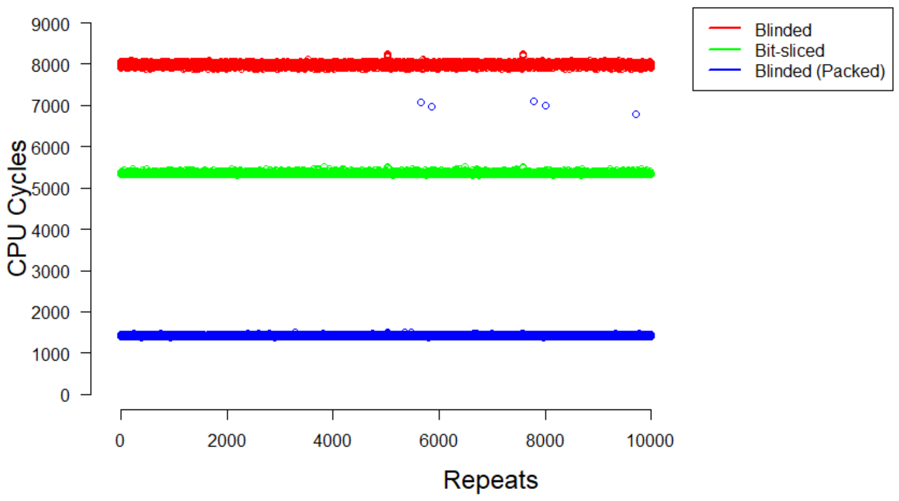


Figure 3: Comparison of blinded vs bit-sliced implementations

3.3 Constant-time SiBMA algorithm (ConSiBMA)

After analysing the SiBMA algorithm, a timing side-channel attack-resistant algorithm can now be proposed which runs in a constant time using Conditional Assignment and (Packed) Blinded Array Access. This algorithm, ConSiBMA, is presented in full in Algorithm 5.

Algorithm 5 Constant-time SiBMA (ConSiBMA)

```
INPUT: S[2t] = S[1], S[2], ..., S[2t]          // array of calculated syndromes
INPUT: t                                         // error-correcting capability of the code
OUTPUT: $C^{t+1}(x)$                             // error-location polynomial (elp) after t iterations
/* Initialise arrays */
$C^{[t+2]}[t+1](x)$                              // current elp with each iteration, t+1 coefficients
                                                 // must initialise $t+2$ of these
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 */
$C^0[0]$ = 1, $C^1[0]$ = 1, D[0] = 1, D[1] = S[1], L[0] = 0, L[1] = 0, UP[0] = -1
upMax = -1, p = -1, pVal = 0
mask = maskGen()                                 // function to generate mask based on data types
/* Main Algorithm */
for i = 1; i <= t; i++ do
    UP[i] = 2·(i-1) - L[i]
    flag1 = ((0 - D[i]) & mask) $\gg$ 31         // determiner equiv. to if D[i] == 0
    $C^{i+1}(x) = C^i(x) \cdot flag1$
    L[i+1] = L[i] · flag1                        // as L[i+1] is unassigned can set to value or 0
    /* Here we look for !flag1 to perform operation as we are in the Else condition */
    /* 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
        flag2 = ((0 - D[i]) & mask) $\gg$ 31;    // determine whether D[j]!=0
        flag2 = flag2 · ((upMax - UP[j]) & mask) $\gg$ 31
                                                 // determine whether D[j]!=0 && UP[j]>upMax
        flag2 = flag2 · !flag1                   // flag2 is dependent on flag1 condition
        upMax = UP[j] · flag2 + upMax · !flag2// set values if flag evaluates, otherwise set to current
        p = j · flag2 + p · !flag2
        pVal = j · flag2 + p · !flag2
    end for
    flag2 = !(((0 - p) & mask) $\gg$ 31)         // determine whether p==0
    pVal = $\frac{1}{2}$ · flag2 + p · !flag2    // set pVal = $\frac{1}{2}$ if p = 0
    $C^{i+1}(x) = (C^i(x) + (D[i] \cdot (blinded(p, t+1, D))^{-1} \cdot x^{2\cdot((\text{i-1})-(\text{pVal-1}))} \cdot C^p(x)))\cdot !flag1$
    L[i+1] = (max( L[i] , $blinded(p, t+1, L)$ + 2 · (i-1)-(pVal-1) )) · !flag1
    flag2 = (((i - t) & mask) $\gg$ 31)          // determine whether i != t
    D[i+1] = ($blinded(2(i \cdot flag2) + 1 - j, t, S)$ + $\sum_{j=1}^{\text{L[i+1]}}$ ( $blinded(j, t+1, C^{i+1}) \cdot blinded(2i + 1 - j, t, S)$) ) · flag2
                                                 // use flag2 to ensure S isnt out-of-bounds
end for
return $C^{t+1}(x)$
```

4 Evaluation

We evaluated the proposed countermeasures with BCH definition `BCH(511,264,59)` and $t = 29$. For each test, for each distinct number of fixed errors (limited by the error-correcting capability t of the code), the generation of a random message (seeded at the beginning for consistency and replication), encoding it into a codeword, adding the errors in random positions, and then decoding the codeword into the original message, was repeated 10,000 times. As encoding does not contain any data-dependent array accesses only one implementation was tested, although it does contain data-dependent operations. For simplicity encoding was tested in the same way as decoding, however as encoding is independent of the number of fixed errors in a codeword (it occurs before the addition of error) we are testing for the countering of timing variations caused by data-dependent operations which were counteracted similarly using the techniques described in Section 3. For decoding, both the (packed) blinded and full table scan methods of array access were tested. As with encoding, syndrome generation and Chien search decoding sub-processes were also made constant-time by altering their implementations according to the features identified by securing the SiBMA algorithm for error-locator polynomial calculation in Section 3. The total number of CPU cycles taken to execute the individual encode and decode processes from start to finish were recorded to provide a good representation of computational effort and execution timing. For each test this then gave $t+1$ sample classes each with 10,000 measurements for the processes, which *unpaired t-tests* on every possible pair of fixed errors and ANOVA could then be performed against. We set a maximum t-score of 4.5 to indicate a program runs in constant time [10]. We report the ANOVA result as

$$[F(df_{param}, df_{residuals}) = f_{value}, P = p_{value}] \tag{22}$$

where df_{param} are the degrees of freedom (d.f.) for the groups (number of groups -1), $df_{residuals}$ are the d.f. for the residuals (number of observations $-$ number of groups), f_{value} is the test statistic, and p_{value} is the probability of observing the test statistic.

Experiments were ran on an Intel i5-6500 desktop CPU running at 3.2 GHz with Linux operating system (OS). As raw CPU cycles are being measured there are noticeable 'spikes' in the data. This is explainable as being either when a machine is performing another user process or when our test program switches between operations, for example from input preparation to gathering measurements. These variances can lead to inaccurate results.

To limit the variance incurred from the first issue a machine was used with no other user process running, and to limit the second issue measurements were increased by 20% and the first 2,000 measurements were discarded before performing t-tests. Hyperthreading on the machine, designed to improve the instruction throughout by assigning virtual (logical) cores per CPU, can understandably result in unwanted effects on the test procedure, and as such was disabled on the test machine to ensure accurate results.

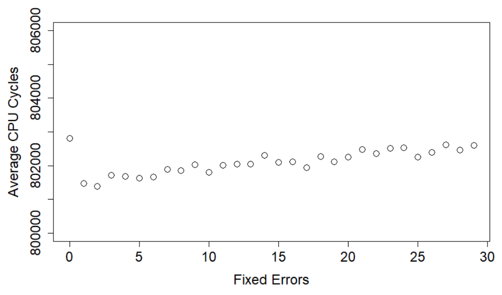
4.1 Evaluation of the original BCH implementation

In Table 1 minimum, average and maximum t-scores are provided, and in Figure 4 reference average CPU cycle and pairwise t-test graphs for the distinct errors tested are provided for the original BCH (reference) implementation. As all t-scores are below 4.5 for encoding we say that it is already a constant-time implementation. However, as most t-scores are above 4.5 for decoding we say that it is variable-time. ANOVA results are reported for encoding as `[F(29,299970) = 8.319, P = <2e-16]`, suggesting at a significant result and therefore variable-time implementation, and for decoding as `[F(29,299970) = 39972, P = <2e-16]`, suggesting at a significant result and therefore variable-time implementation.

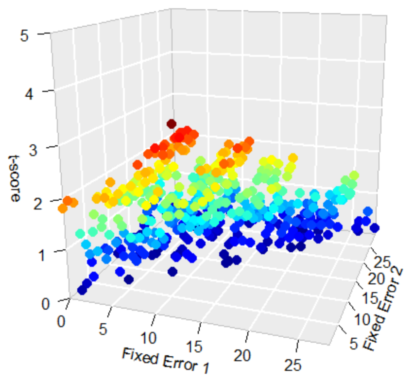
Table 1: t-score range for reference BCH encoding and decoding

	t-score		
Process	**Minimum**	**Average**	**Maximum**
Encoding	0.0032	0.9142	2.4520
Decoding	0.0274	18.8223	78.2366

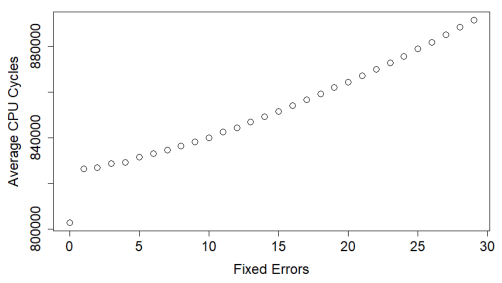
Although encoding appears to execute in constant time from its t-score, it fails ANOVA. Looking at the CPU cycle graph for encoding (a) in Figure 4, although there is little slope, the encoding algorithm does appear to be variable-time, indicating at the data-dependent operations referred to in Section 5.3. As such, t-test values are included in the subsequent evaluations for added certainty but conclusions are drawn only from ANOVA results.



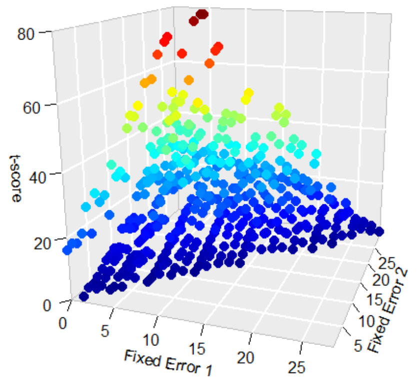
(a) Encoding: Average CPU cycles



(b) Encoding: Pairwise t-tests



(c) Decoding: Average CPU cycles



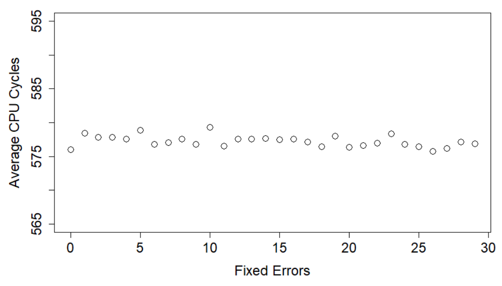
(d) Decoding: Pairwise t-tests

Figure 4: Statistical tests of BCH reference implementation

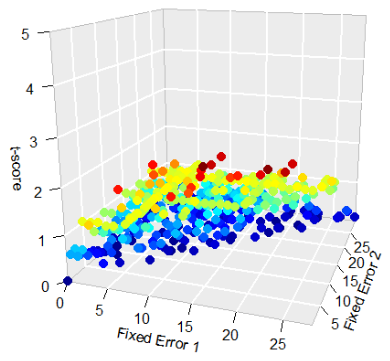
4.2 Evaluation of the proposed implementation

4.2.1 Evaluation of the proposed implementation on Desktop

BCH encoding: No pair of fixed errors has a t-score greater than 4.5 (Minimum = 0, Average = 0.6092, Maximum = 1.6590). ANOVA reports `[F(29,299970) = 1.023, P = 0.43]`. As $P > 0.05$, there is no significance. Encoding therefore runs in a constant time.



(a) Average CPU cycles



(b) Pairwise t-tests

Figure 5: Evaluation of BCH encoding on Desktop computer

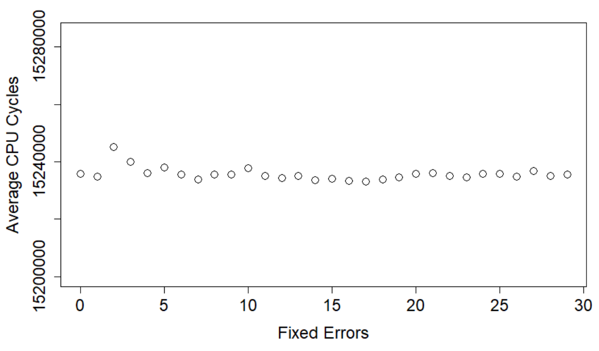
BCH decoding: Performing t-tests between all possible pairwise combinations of fixed errors for both the blinded and full table scan implementations of decoding no pair is found to have a t-score smaller than 4.5, indicating that both perform in constant time irrespective of the number of fixed errors present in a received codeword.

Table 2: t-score range for BCH decoding on Desktop

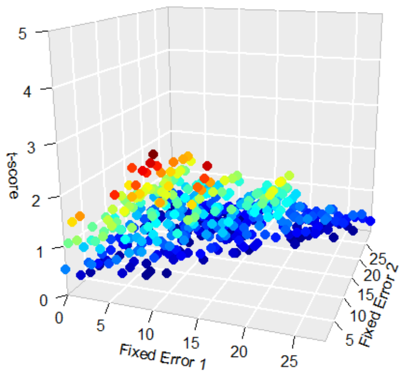
	t-score		
Implementation	**Minimum**	**Average**	**Maximum**
Blinded	0.0002	0.5748	2.0297
Full Table Scan	0.0025	0.8112	3.1881

ANOVA reports `[F(29,299970) = 0.747, P = 0.834]` for blinded and `[F(29,299970) = 0.938, P = 0.561]` for full table scan implementations. As P $>$ 0.05 for both, no significance between results, and as such both implementations perform in a constant time.

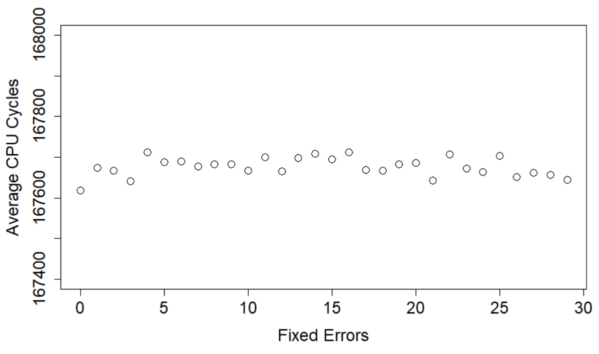
It is noticeable here that although the Full Table Scan CPU cycle graph appears constant (as much as any other graph in this evaluation), its t-score is quite high (although not above the 4.5 threshold) and its ANOVA P value is quite low (although not near the 0.05 threshold) compared to other results. This is most likely attributed to underlying variation caused by the OS despite the mitigation which was put in place, which would have a greater affect on this implementation causing greater cycle count variation than the Blinded (which is approximately 100 times slower). Although the encoding process does not seem to be affected by this variation, it is approximately 200 times faster and therefore it may be hypothesised that it executes fast enough to not be interrupted.



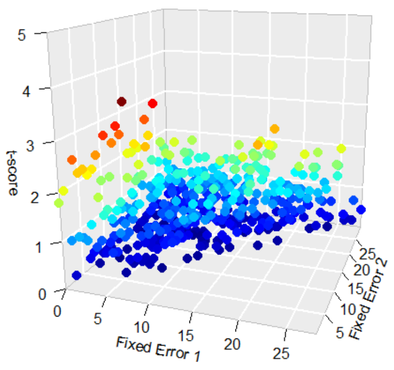
(a) Blinded: Average CPU cycles



(b) Blinded: Pairwise t-tests



(c) Full Table Scan: Average CPU cycles



(d) Full Table Scan: Pairwise t-tests

Figure 6: Evaluation of BCH decoding on Desktop computer

4.2.2 Evaluation of the proposed implementation on bare metal ARM microcontroller

To further support the claims of a constant time implementation of BCH decoding, experiments were performed on a bare metal ARM Cortex M4 processor residing on a STM32F4-discovery board from STMicroelectronics. With no overarching OS and data-cache, process and memory-access fluctuations are eliminated. Experiments were performed in a similar fashion to the ones previously, except that 100 measurements were taken 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. Experiments were only performed on the decoding process in this way due to the magnitude of CPU cycle values. As there is no data cache on the targeted ARM platform, there are no timing variations for array accesses for attackers to exploit. Therefore, an additional implementation with no table access countermeasures, i.e. with direct table access, was also evaluated on the ARM platform.

Table 3: t-score range for BCH decoding on ARM

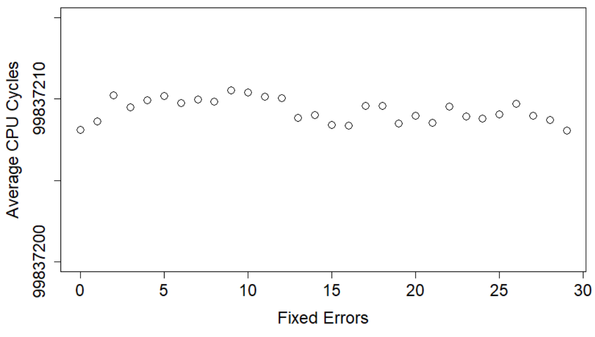
	t-score		
Implementation	**Minimum**	**Average**	**Maximum**
Blinded Table Scan	0.0042	0.3570	1.0363
Full Table Scan	0	0.1893	0.5147
Direct Table Access	0	0.2335	0.9857

Table 3 shows the results of the pairwise t-test scores for all three implementations. We find that no pair have a t-score greater than 4.5, indicating that all implementations execute in constant time on the ARM platform irrespective of the number of fixed errors present in a received codeword.

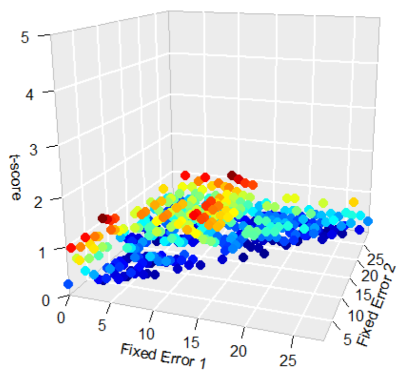
ANOVA reports `[F(29,2970) = 0.187, P = 1]` for blinded, `[F(29,2970) = 0.064, P = 1]` for full table scan and `[F(29,2970) = 0.097, P = 1]` for direct table access implementations. As `P` > 0.05, there is no significant difference between results, and as such the implementations execute in a constant time on the ARM Cortex M4 platform.

4.2.3 Conclusions from experimental results

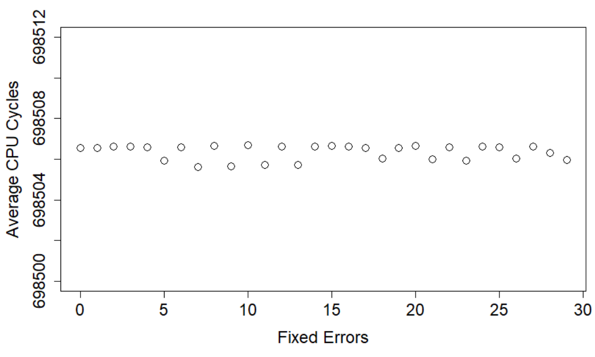
Statistical results from ANOVA, supported by visual average CPU cycle plots and indications from t-tests, confirm that the implementation of the entire BCH process, secured by the extrapolation of countermeasures to variable-time features identified in the SiBMA algorithm, performs in constant time. The use of ARM is evidence for the general correctness of the implementation (achieving very low t-scores and the highest P values possible for ANOVA) whilst Desktop tests show that this implementation is usable in real-world scenarios (achieving slightly higher t-scores and non-significant but not outstanding P values for ANOVA). A greater spread in CPU cycles for Desktop tests than ARM is observed and is a likely reason for less optimal statistics compared to ARM. It is hypothesised that this can be attributed to the underlying OS on the machine running Desktop tests. With other processes requiring usage of the CPU, the test process can be interrupted or otherwise adversely affected which as a result will cause fluctuations in measurements. Although steps were taken to mitigate the interference of User Processes and the OS when performing Desktop tests, this does not impact on the usability of this solution in real-world scenarios, as the mitigation was only performed so-that accurate cycle count results could be obtained, and induced variability in a real-world scenario would actually aid in obfuscation further, reducing the chances of a successful timing side-channel attack.



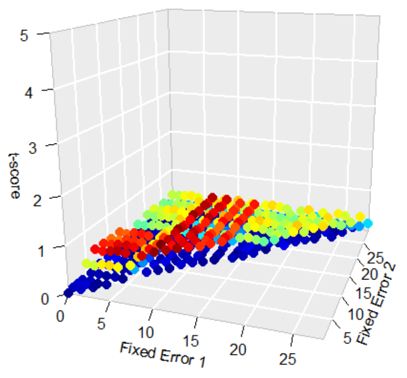
(a) Blinded: Average CPU cycles



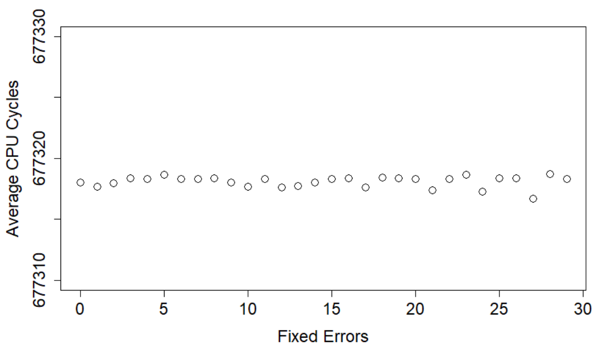
(b) Blinded: Pairwise t-tests



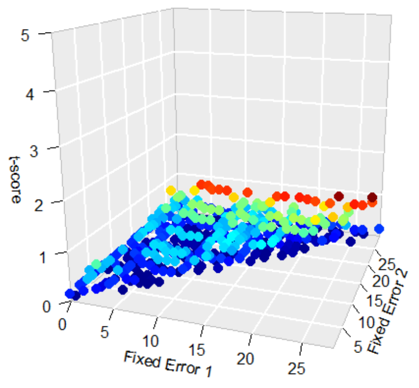
(c) Full Table Scan: Average CPU cycles



(d) Full Table Scan: Pairwise t-tests



(e) Direct Table Access: Average CPU cycles



(f) Direct Table Access: Pairwise t-tests

Figure 7: Evaluation of BCH decoding on ARM Cortex M4 microcontroller

4.3 Application to post-quantum public-key encryption

We now provide performance overhead as a result of the application of the proposed constant-time BCH error-correcting code in public-key cryptography. Two cryptosystems, HQC [21] and LAC [16], which reached the Round 2 phase of NIST's Post-Quantum Standardisation project, utilise BCH code for their error correction. In this research, we restrict to the application of the constant-time BCH error-correcting code to LAC.

CPU cycle counts for both the reference and optimised variants of the LAC decoding implementation were gathered and then the constant-time BCH implementation was integrated to determine the performance costs of the countermeasures implemented. LAC offers three choices of security parameter (128, 192, 256) which satisfy various security categories in the NIST documentation and are documented to use various different definitions of BCH. LAC-128, using the definition `BCH(511,385,29)` and an error-correction capability of 20 errors, was used for these tests.

Table 4 shows performance overhead across all four primitives of LAC. For the CCA-secure decapsulation operation, using the full table scan access implementation, we observed an average 1.1-factor slowdown against the reference implementation and an average 1.4-factor slowdown against the optimised implementation. With the blinded access implementation, the CCA-secure decapsulation operation becomes around 3.7-factor slower than the reference implementation and around 41-factor slower than the optimised implementation. The table details both the average CPU cycles taken to execute the respective LAC primitive with either the original or modified BCH implementation, and the relative slowdown factors incurred. Although these are averages, the modified BCH variant always runs in constant-time and variation may be attributed to the OS or other factors.

Given the experimentally confirmed constant-time performance of the full table scan implementation, 1.1 and 1.4 factor slowdowns for reference and optimised implementations respectively is a good result, showing that improved security can come with little overhead. The more secure blinded access implementation understandably induces greater overhead.

Code-based public-key scheme HQC [21] uses a variant of BCH and is a relatively slower scheme compared to LAC. Hence, we anticipate that the our constant-time BCH code will result in smaller performance overhead when integrated in HQC.

Table 4: Average cycle counts and slowdown factors when constant-time BCH is used in LAC

		Reference			**Optimised**		
		CPU Cycles			**CPU Cycles**		
Scan	**Primitives**	**Orig.**	**Mod.**	**Slow.**	**Orig.**	**Mod.**	**Slow.**
Full	*CPA.DEC*	1428429	1869943	1.3	91501	180244	2.0
	CCA.DEC	3501441	3954968	1.1	238632	329064	1.4
	KE.DEC	1436207	1878500	1.3	94037	184652	2.0
	AKE.DEC	3974442	5854306	1.5	349271	527466	1.5
Blinded	*CPA.DEC*	1428429	10974346	7.7	91501	9634475	105.3
	CCA.DEC	3501441	13081239	3.7	238632	9818436	41.1
	KE.DEC	1436207	10993750	7.7	94037	9644891	102.6
	AKE.DEC	3974442	24122748	6.1	349271	19514431	55.9

5 Conclusions

In this paper we analysed the BCH error-correcting code and proposed the first constant-time implementation of BCH decoding algorithm by introducing algorithmic tweaks in the low-level building blocks. Our constant-time BCH algorithm makes post-quantum public-key schemes that rely on powerful BCH code for error correction, such as LAC and HQC, more resistant against simple timing side-channel attacks. We have considered overhead as an issue with the constant-time transformation of existing implementations and have discussed methods to reduce these at the expensive of security.

Potential future works include a more in-depth algorithm analysis to further improve efficiency and reduce overhead, and an investigation into our implementation's security against Power Side-Channel Attacks [17].

References

[1] Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. pp. 719–737. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[2] Berlekamp, E.R.: Algebraic coding theory. McGraw-Hill series in systems science, World Scientific, New Jersey, revised edition edn. (2015)

[3] Carreira, J.V., Costa, D., Silva, J.G.: Fault injection spot-checks computer system dependability. IEEE Spectrum **36**(8), 50–55 (1999)

[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] Djelic, I.: Generic binary BCH encoding/decoding library (Jan 2019), https://github.com/Parrot-Developers/bch, original-date: 2017-03-10T13:33:59Z

[9] Fritzmann, T., Pöppelmann, T., Sepulveda, J.: Analysis of error-correcting codes for lattice-based key exchange. Cryptology ePrint Archive, Report 2018/150 (2018), https://eprint.iacr.org/2018/150

[10] Goodwill, G., Jun, B., Josh, J., Pankaj, R., et al.: A testing methodology for side-channel resistance validation. In: NIST non-invasive attack testing workshop. vol. 7, pp. 115–136 (2011)

[11] 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/

[12] Hamburg, M.: Supporting documentation: ThreeBears (2017) https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions

[13] Han, Y.: BCH Codes, http://web.ntpu.edu.tw/~yshan/BCH_code.pdf

[14] 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

[15] Kanawati, G.A., Kanawati, N.A., Abraham, J.A.: Ferrari: A flexible software-based fault and error injection system. IEEE Transactions on computers **44**(2), 248–260 (1995)

[16] 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

[17] 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)

[18] Mantel, H., Starostin, A.: Transforming Out Timing Leaks, More or Less. In: Pernul, G., Y A Ryan, P., Weippl, E. (eds.) Computer Security – ESORICS 2015, vol. 9326, pp. 447–467. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-24174-6_23, http://link.springer.com/10.1007/978-3-319-24174-6_23

[19] 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/

[20] 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

[21] Melchor, C.A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zemor, G.: Hamming quasi-cyclic (HQC). http://pqc-hqc.org/, (Accessed on 03/08/2019)

[22] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. ACM Symp. on Theory of Computing (STOC) (37), 84–93 (2005)

[23] Saarinen, M.J.O., Bhattacharya, S., Garcia-Morchon, O., Rietman, R., Tolhuizen, L., Zhang, Z.: Shorter messages and faster post-quantum encryption with Round5 on Cortex M. Cryptology ePrint Archive, Report 2018/723 (2018), https://eprint.iacr.org/2018/723

[24] 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), http://arxiv.org/abs/quant-ph/9508027, arXiv: quant-ph/9508027