IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 8, 2020, accepted May 20, 2020, date of publication June 1, 2020, date of current version June 11, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.2998950*

# Table-Free Multiple Bit-Error Correction Using the CRC Syndrome

**VIVIEN BOUSSARD[1,2], (Graduate Student Member, IEEE), STÉPHANE COULOMBE[1], (Senior Member, IEEE), FRANÇOIS-XAVIER COUDOUX[2], (Senior Member, IEEE), AND PATRICK CORLAY[2]**

[1]Department of Software and IT Engineering, École de technologie supérieure, Université du Québec, Montreal, QC H3C 1K3, Canada
[2]University Polytechnique Hauts-de-France, CNRS, University Lille, ISEN, Centrale Lille, UMR 8520–IEMN–Institut d'Électronique de Microélectronique et de Nanotechnologie, DOAE–Département d'Opto-Acousto-Électronique, 59313 Valenciennes, France

Corresponding author: Vivien Boussard (vivien.boussard.1@etsmtl.net)

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant. The authors also wish to thank UPHF for its financial support of this Ph.D. in international joint supervision.

**ABSTRACT** In this paper, we propose a novel method for correcting multiple errors in data packets, using the Cyclic Redundancy Check (CRC) syndrome present in low layers of protocol stacks. The proposed method generates the whole list of error patterns, leading to a received syndrome containing up to a given maximum number of errors. Our approach is table-free, is computationally efficient, and can instantly correct erroneous packets when the output list contains a single element. A performance study is conducted, and shows that the proposed approach outperforms existing ones in Bluetooth Low Energy (BLE) as it can correct all single- and double-error patterns as well as most triple-error cases when considering small payloads used in Internet of Things (IoT) applications.

**INDEX TERMS** Data communication, error correction, cyclic redundancy check (CRC), Internet of Things (IoT), Bluetooth low energy (BLE).

## I. INTRODUCTION

Cyclic Redundancy Check (CRC) codes constitute a well-known special case of checksum functions, which are typically used for packet error detection in a wide variety of low-layer protocols [1]. Their main purpose is to validate the integrity of received packets. If an error is detected by such codes, the corrupted packet is normally discarded and a data recovery mechanism can be set, as implemented in protocols such as the Transmission Control Protocol (TCP) [2], where reliability is ensured through retransmission of the corrupted data. In order to avoid systematic retransmission, which would lead to an increased amount of data and extra delays within the network, error correction methods have been proposed at the receiver side. In addition, error detection codes such as CRCs and Checksums [3] have also been demonstrated to allow error correction [4]–[11]. The principle of CRC error detection is based on the computation of a so-called CRC field at the transmitter side. The value of this field is the remainder of the long division of the protected bit sequence, the data, which we will refer to as the *payload*, denoted $d(x)$, by a generator polynomial (a binary polynomial of degree $n$ defined by the protocol used, denoted $g(x)$). The payload is left-shifted by $n$ positions before the division. In Eq.(1), the remainder is denoted $r(x)$ and $q(x)$ represents the quotient of the long division [1]:

$$d(x).x^n = q(x).g(x) + r(x) \Rightarrow \text{ CRC} = r(x) \qquad (1)$$

The computed CRC field $r(x)$ is then appended to the payload and sent to the receiver. The transmitted packet, comprising the payload and its associated remainder, is denoted $p_T(x) = d(x).x^n + r(x)$.

At the receiver, a long division by $g(x)$ is performed on the received packet, denoted $p_R(x)$, in order to check it's integrity. An error-free packet (i.e., $p_R(x) = p_T(x)$) is thus a multiple of $g(x)$ and the remainder is zero. In the contrary case, an error will modify $p_T(x)$ and produce a non-zero value as the remainder. The result is called the syndrome of the CRC, denoted $s(x)$. The standard management here consists in automatically discarding a received packet with a non-null syndrome. However, such management leads to a waste of information. In real-time applications such as

The associate editor coordinating the review of this manuscript and approving it for publication was Maurizio Murroni.

video conferencing, packet retransmission is unavailable. One would then benefit from extracting as much information as possible from a received corrupted packet. Our approach is to propose algorithms to attempt to repair such corrupted data using the actual syndrome value.

CRC-based error correction techniques have been explored in previous works, and can be divided into two main categories:

1) **Estimator approaches [11]–[13]**: These approaches use statistical estimators, such as the Maximum A Posteriori (MAP) estimator, and aim at finding the most probable binary sequence that has been sent, considering the received erroneous sequence. The CRC is used to check the validity of the MAP sequence or can be part of the estimation process. Such methods can use optimization techniques such as the Alternating Direction Method of Multipliers (ADMM) [14] or Belief Propagation (BP) [15]. These approaches to MAP are costly, and generally use Log Likelihood Ratios (LLR) [13] and provide information on the confidence of the received bit, expressed as a real value between $-\infty$ and $+\infty$, based on the received soft values. Unfortunately, today's TCP/IP and User Datagram Protocol UDP/IP protocol stacks are essentially designed to deal with hard values (decoded bits), and consequently, such approaches cannot be implemented in current architectures without great effort.
2) **Lookup table approaches [4]–[8]**: These approaches implement lookup tables prior to the communication, in which each entry contains the syndrome resulting from one [6] or two [7] errors at specific positions. Upon reception, when a CRC check results in a non-null syndrome, the table is scanned. If a match is found, the corresponding bit positions are flipped to correct the packet. By definition, the CRC codes are designed such that each single error leads to a unique syndrome within the period of the generator polynomial used. The period of a generator polynomial is thus the number of different syndromes it can output for single errors. If the packet length surpasses the period of the generator polynomial, several single-error positions could lead to the same syndrome, thereby introducing ambiguity. Recently, some CRC-aided error correction methods have implemented a lookup table approach to increase their correction capacities [5]. Besides high memory requirements for storing the table, such approaches raise two main issues:
   - Lack of flexibility: lookup tables must be generated prior to the transmission, and cannot be dynamically modified to support multiple generator polynomials and larger packet sizes than those for which they were designed.
   - Memory constraints: memory requirements for lookup table-based approaches rapidly increase with the number of errors to consider. In fact, such methods must store the entire set of possible error patterns and their associated syndrome.

In this paper, we propose a novel approach to error correction that outputs the exhaustive list of CRC-compliant binary sequences containing up to $N$ errors. This method does not need any lookup table, which thus reduces the memory resources needed and allows the algorithm to be flexible as it can be used for any number of errors and any payload length without the need to rebuild a lookup table. Whereas CRC-aided Maximum Likelihood (ML) methods [4] typically use CRC to check the validity of the candidates at the end of the MAP process, our method uses the CRC syndrome itself to produce the list of candidates, thus ensuring the CRC integrity of every candidate. The output list can be used to instantly correct the packet if it contains a single element or it can be used along with error correction or validation methods from upper layers of the protocol stack in order to reduce the list of candidates.

The paper is organized as follows. In section II, we give a detailed description of the proposed method in three distinct parts. The first one describes the concept of the approach and its application to single-error correction. Challenges encountered with the double-error correction are then introduced and generalization to any number of errors is explained. In section III, we present the proposed algorithm's performance as compared to state-of-the-art approaches. Tests are conducted according to different standards used in targeted applications, such as Wi-Fi [16] and Bluetooth Low Energy [17]. Simulation results demonstrate the superiority of the proposed solution in terms of error correction rate, computational complexity and memory usage. They show that for small-sized packets such as those found in IoT, we can achieve a 100% correction rate for corrupted packets containing two errors or less, as well as high correction rates for three errors. In section IV, we conclude and give an overview of future research works.

## II. PROPOSED METHOD

The proposed method uses the CRC syndrome value $s(x)$ computed at the receiver to list all the possible error patterns that lead to such a specific syndrome, considering a maximum number of errors. The resulting list can contain one or several entries at the end of the process. Each entry represents the positions of the bits to be flipped to recover a CRC-valid packet, i.e., it reveals the error positions. When the list contains only one element, we can instantly correct the packet, but when it contains several entries, additional information is required in order to identify the actual error pattern among the candidates. The proposed approach is flexible, and lists the whole set of possible error patterns with up to $N$ errors, where the parameter $N$ can be set according to the observed channel conditions, for instance. In this section, we first introduce the basic theoretical concepts of the proposed method for the single-error case. Then, we extend the method to double-error patterns, followed by multiple-error patterns.

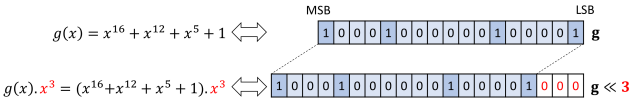


**FIGURE 1.** Illustration of the binary vector representation: each polynomial $g(x)$ with binary coefficients can be seen as a binary vector g. Multiplying $g(x)$ by $x^n$ corresponds to a left shift of g by $n$ positions.

### A. FUNDAMENTALS

For convenience, it is common to use a binary vector representation of binary polynomials as described in [19] and illustrated in Fig. 1. Using the vector representation, the degree of a coefficient corresponds to the bit position of the associated element in the vector. The length, in bits, of a vector is equal to the degree of the polynomial increased by 1, due to the existence of degree 0 in the polynomial (i.e., a polynomial of highest degree $x^{15}$ will be represented as a 16-bit vector). Vectors allow a better understanding of operators such as *exclusive or* (XOR) and *binary left shifts*. Throughout this paper, specific notations will be used. The following is a list of such notations based on [18] and their definitions:

- $\mathbf{a}$: binary vector $[a_k, \ldots, a_0]$ of length $k+1$ associated with the binary polynomial $a(x)$ of degree $k$
- $a_i$: $i^{th}$ bit (entry) of binary vector $\mathbf{a}$, starting from least significant bit (LSB)
- $m$: payload length in bits
- $n$: syndrome length in bits
- $M$: total packet length in bits ($M = m + n$)
- $N$: number of errors searched
- $P_1$: error position obtained from the single-error correction algorithm (Algorithm 1)
- $\mathcal{F}$: sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$
- len($\mathcal{F}$): number of elements in the list $\mathcal{F}$
- $E_i$: set of valid error patterns containing $i$ errors or less
- sum($\mathbf{a}$): number of non-zero elements in a binary vector $\mathbf{a}$; also denoted $\sum$ when the context is clear
- $\oplus$: XOR operator between binary vectors
- $+$ : XOR operator between polynomials
- $\ll$: left shift operator
- $\leftarrow$: affectation operator
- $t_i$: $i^{th}$ step

We will frequently use the following binary vectors:

- $\mathbf{0}$: null vector (the length depends on the context)
- $\mathbf{g}$: generator polynomial vector of length $n+1$ with $g_0 = 1$ and $g_n = 1$, given its definition [20]
- $\mathbf{s}$: syndrome vector of length $n$
- $\mathbf{e}$: error vector of length $M = n + m$

According to the definition of the CRC [1], we know that the syndrome $s(x)$ is computed at the receiver as the remainder of the division of the received packet by the generator polynomial, which can be expressed as:

$$s(x) = p_R(x) \bmod g(x) \tag{2}$$

When no error occurs, the syndrome $s(x)$ is equal to a null polynomial. If we consider an error pattern $e(x)$, the syndrome of the received packet can be expressed as:

$$s(x) = (p_T(x) + e(x)) \bmod g(x) \tag{3}$$

where $s(x)$ is a non-null syndrome. A given syndrome value can be the result of several different error patterns $e(x)$, containing different numbers of errors. We denote $E_M(s(x))$ the set of all valid error patterns leading to the syndrome $s(x)$. In order to lighten the notation, we will use $E_M$ since we are interested in a single syndrome value throughout the process. The error patterns in $E_M$ contain between 1 and $M$ errors (all bits of the packet are erroneous in the latter case). We denote $E_i$ the subset of $E_M$ comprising error patterns with $i$ errors or less ($1 \leq i \leq M$). We thus have:

$$E_i \in E_{i+1} \quad \forall\, 1 \leq i \leq M-1 \tag{4}$$

where the $E_i$ are not disjoint sets. The number of elements in $E_M$ and in each subset $E_i$ depends on the syndrome, the generator polynomial used and the packet length. We aim at finding the actual error pattern $e_A(x)$ among the set of all error patterns leading to the computed syndrome value $s(x)$. Given the definition of the modulo operator and Eq.(3), all error patterns of $E_M$ are defined as:

$$E_M = \{e(x) \in \mathrm{GF}(2^M) \mid e(x) = s(x) + q(x).g(x) \quad \text{with} \quad q(x) \in \mathrm{GF}(2^m)\} \tag{5}$$

where $m$ is the payload length and GF($2^m$) is the Galois Field of order $2^m$ (i.e., the set of binary polynomials of length $m$ [19]). In other words, the error pattern corresponding to the syndrome can be any binary polynomial of highest degree $m-1$ (that we denoted as $q(x)$) multiplied by the generator polynomial, with $s(x)$ added. The set $E_M$ is called the equivalence class containing $s(x)$. Each element is equivalent under mod $g(x)$ operation since adding any multiple of $g(x)$ to $s(x)$ does not affect the result. Every possible value of $q(x)$ in this equation will produce a CRC-compliant error pattern $e(x)$ (i.e., an element of $E_M$). The degrees of the non-zero coefficients in the resulting $e(x)$ correspond to the erroneous positions in the corrupted packet. Assuming that packets are not too damaged, the straightforward approach to identifying candidates having a maximum number of errors would be to test every possible value of $q(x)$ and to count the number of non-zero coefficients in the resulting error polynomial $e(x)$. If this number, denoted sum($\mathbf{e}$), is greater than a fixed threshold, the candidate is discarded. Otherwise, it is appended to the list of valid candidates. This method is computationally complex, and would require $2^m$ tests to consider all the possible values of $q(x)$. Such a complex process is therefore prohibitive to conduct in real-time scenarios, such as videoconferencing, for instance.

It can be verified that most of the possible values of $q(x)$ produce error polynomials $e(x)$ containing many errors (i.e., corresponding to highly corrupted packets cases, where $e(x)$ and its associated vector $\mathbf{e}$ contain a significant amount of non-null values). Considering the whole set of possibilities would only increase the complexity of the method. We make

the hypothesis that highly corrupted packets are too damaged to be recovered. Thus, in the rest of this paper, we focus on recovering slightly corrupted packets that are worth extracting information from. Some indicators such as the Receiver Signal Strength Indicator (RSSI), included in the 802.11 standard [16], can be used to indicate the degree of corruption of a received packet. In the remainder of this section, we first describe the single-error correction method (the search for all elements in $E_1$), and then we introduce the double-error correction and extend it to any number $N$ of errors (i.e., we determine the elements of $E_N$).

### B. SINGLE-ERROR CORRECTION

We exploit the knowledge on both the generator polynomial and the way the syndrome is computed to reversely find the position of the single error at the receiver side. With such an approach, we are not testing possible values of $q(x)$, but rather, are gradually building a specific polynomial $q(x)$, one coefficient at a time. If a single candidate is identified at the end of the process, the packet can be corrected. If not, some additional processes must be used to determine the only candidate to consider.

We now demonstrate that the proposed approach is guaranteed to identify single errors. Suppose that the error is at position $P_1$ (i.e., $e(x) = x^{P_1}$). We know from the definition of Eq.(5) that:

$$x^{P_1} = s(x) + q(x).g(x) \text{ for a } q(x) \in \text{GF}(2^m) \tag{6}$$

It is clear that $q(x)$ must be constructed such that $s(x) + q(x).g(x)$ has zero coefficients for all positions $i \neq P_1$. Having coefficients at positions $i < P_1$ is ensured by successively determining, from LSB to MSB, the coefficient values of $q(x)$ meeting this condition. For simplicity, in the following derivations, we can consider $s(x)$ of degree $m-1$ with $s_i = 0, i > n-1$. We have:

$$\begin{aligned}
x^{P_1} &= \sum_{i=0}^{m-1} s_i x^i + \left(\sum_{i=0}^{m-1} q_i x^i\right)\left(\sum_{j=0}^{n} g_j x^j\right) \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} q_i . \sum_{j=0}^{n} g_j x^{i+j} \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} q_i . \sum_{r=i}^{i+n} g_{r-i} x^r \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} \left(q_i . g_0 x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right) \\
&= \sum_{i=0}^{m-1} \left((s_i + q_i . g_0) x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right) \\
&= \sum_{i=0}^{m-1} \left((s_i + q_i) x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right), \text{since } g_0 = 1
\end{aligned} \tag{7}$$

From Eq.(7), it is clear that for every value of $i$ in the main summation, $(s_i + q_i.g_0)x^i$ is of a lower degree than $q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r$. Thus, for $i = 0$ and each successive value of $i$, we can easily determine the $q_i$ value resulting in the desired result, namely, zero coefficients for positions $i < P_1$. Of course, setting $q_i$ to 1 creates terms that must be considered in subsequent positions. If $s(x)$ was generated by a single error, performing the process on increasing values of $i$ would eventually lead to a monomial (i.e., $x^{P_1}$ for a certain value of $P_1$). This must happen, otherwise, after position $i = P_1$, adding $\sum_{r=i+1}^{i+n} g_{r-i} x^r$ (i.e., $q_i \neq 0$) would add a coefficient at position $i + n$ (the MSB) that cannot be canceled without adding a coefficient of even higher degree.

**Algorithm 1** SingleErrorCorrection(**s**,**g**,$n$,$m$)

**Inputs:**

**s**: the syndrome vector
**g**: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector

**Output:**

$E_1$ the list of valid error patterns for a single bit error

```
1: E1 ← {}
2: Let e be a vector of length m + n
3: e ← 0 ⊕ s
4: if sum(e) = 1 then
5:     Add e to E1
6: end if
7: for j = 0 to m − 1 do
8:     if e_j = 1 then
9:         e ← e ⊕ (g ≪ j)
10:        if sum(e) = 1 then
11:            Add e to E1
12:        end if
13:    end if
14: end for
15: Return E1
```

The search for single-error patterns is illustrated in Algorithm 1 and the corresponding flowchart is given in Fig. 2. Each step of the algorithm is identified in Fig. 2 using binary notations. We provide further details in the following steps:

**3**: We first initialize the error **e** to a zero vector of length $M = m + n$ and replace the $n$ LSB values with the computed syndrome **s**, as shown in Fig. 3. We can note that it corresponds in Eq.(5) to $e(x) = q(x).g(x) + s(x)$, where $q(x)$ is equal to zero. Such initialization allows to comply with Eq.(5) and maintains its equivalence relation as we are adding shifted versions of $g(x)$ to build $q(x)$ in step 9.

**4-5**: At this point, we compute the sum of non-zero elements in $\mathbf{e} = \mathbf{s}$, denoted sum(**e**), equivalent to the number of

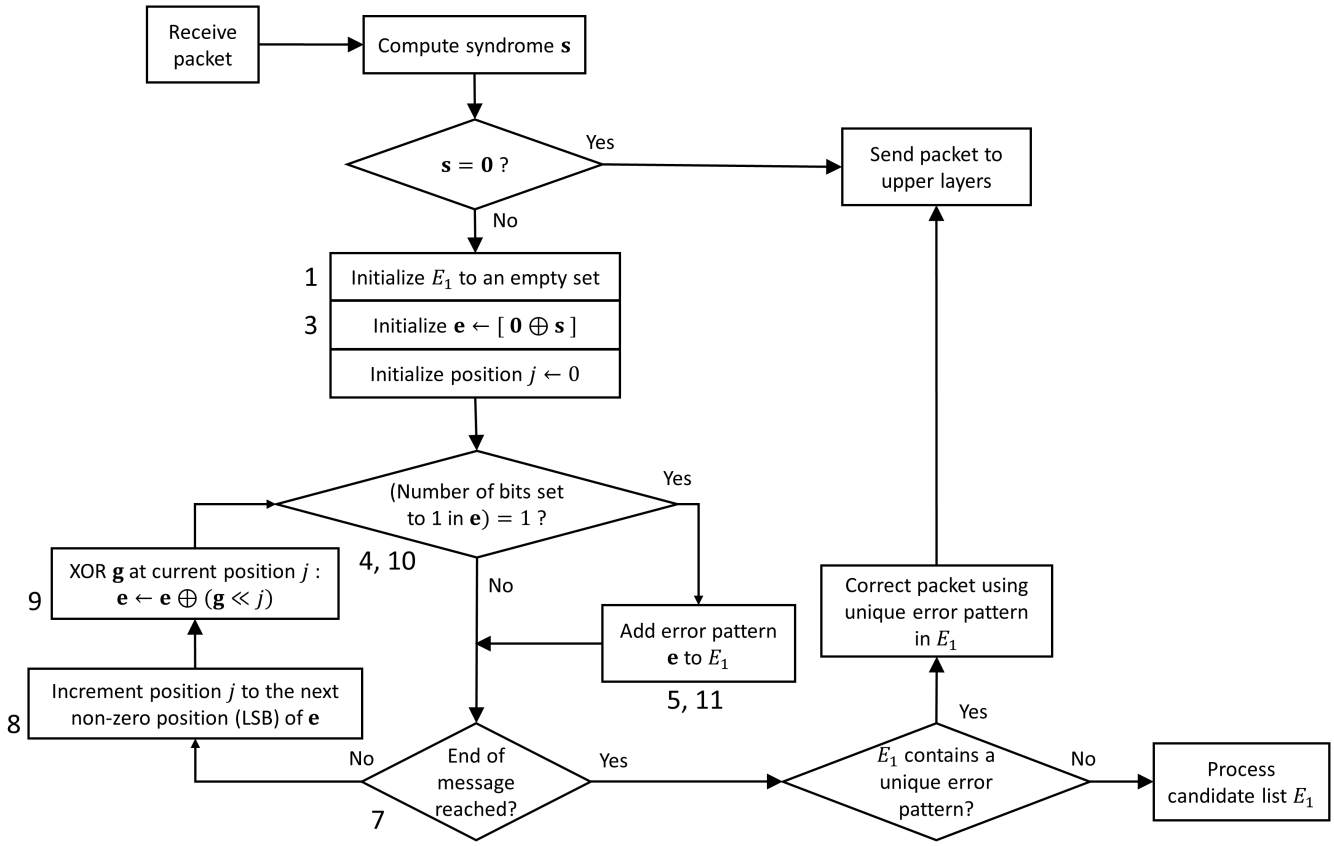


**FIGURE 2.** Flowchart of the proposed method's algorithm to correct a single error in the packet. Numbers show the corresponding steps in Algorithm 1.

errors in the computed syndrome. If it contains only one element to 1, then it is itself a suitable candidate as a single-error pattern.

**7**: We scan the $m$ first payload positions from 0 to $m-1$. We do not consider the last $n$ positions since they correspond to the range of the XOR operation to perform. Hence, it reaches the end of the payload at position $m-1$ and beyond this position would be out of the payload range.

**8-9**: For each scanned position, we check the $j^{th}$ bit value of the current error vector. If this value is 0, we simply jump to the next element. If it is 1, we cancel the non-zero value by performing an XOR operation with **g** at this position, as its LSB is 1 (i.e., $g_0 = 1$). Note that for clarity, we simplified this step in the figures and flowcharts by directly incrementing the current position to the next element set to 1. Each time we perform an XOR operation at position $j$, a 1 is added at MSB position $j+n$ since $g_n = 1$. If the error pattern is a single error at position $k$, the proposed method will reveal this since the XOR operations will be able to cancel all bits at positions $j < k$, and all bits at positions $j > k$ are already set to zero. The strategy is to cancel every LSB non-zero element until the end of the packet is reached, and thus not miss any single-error candidate.

**10-11**: After each cancelation, we check the number of non-zero coefficients in the error vector **e**. If this number is equal to 1, a valid single-error candidate is identified and its position is appended to the list.

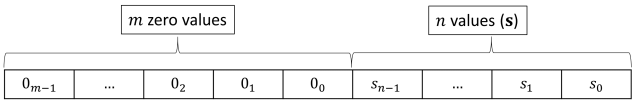


**FIGURE 3.** Structure of the initial error vector $\mathbf{e} = \mathbf{0} \oplus \mathbf{s}$.

At the end of the whole process, if the algorithm does not provide any candidate, it means the syndrome was caused by multiple errors in the packet.

So, depending on the packet size and syndrome, there can be zero, one or multiple candidates. This latter case occurs in long enough packets due to the periodic aspect of generator polynomials, as discussed in the introduction. The whole single-error search process is illustrated in Fig. 4. In the present case, the payload consists of 10 data bits and the CRC-4-ITU where a generator polynomial $g(x) = x^4 + x + 1$ is applied. At the receiver, the computed syndrome is $s(x) = x^2 + 1$, represented in dark grey boxes at step $t_0$.

At step $t_0$, the error vector **e** is initialized to $m$ zeros, $m$ being the length of the protected data, and the syndrome **s** is appended. We first check the number of non-zero values in the error vector to verify if the syndrome itself is a valid candidate

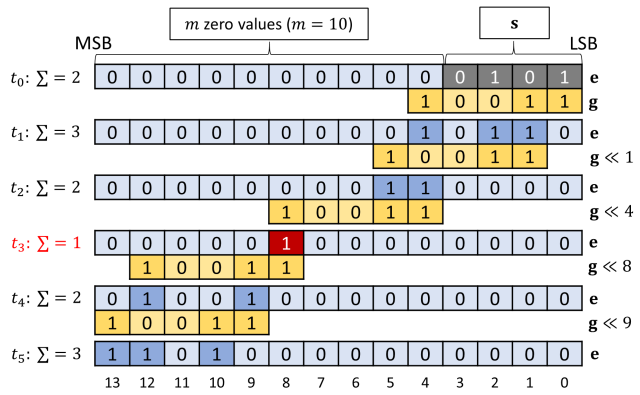


**FIGURE 4.** Illustration of the single-error search applied to CRC-4-ITU, where $g(x) = x^4 + x + 1$ (yellow cells) with a syndrome $s(x) = x^2 + 1$ (grey cells). In this notation, $\sum$ represents sum(e). A single-error pattern is found at bit position 8 at step $t_3$. In this example: $E_1 = \{8\}$ (i.e., the red cell).

(i.e., if the syndrome is corrupted). Since the sum of non-zero values in **s** is greater than 1, the syndrome does not contain a valid single-error pattern, and is thus not a candidate. At each step until we reach the end of the packet, we successively perform an XOR operation with **g** at each non-null position and check the resulting number of 1s in the updated error vector **e**. If this sum is equal to 1, the candidate is appended to the list. Such a candidate is found at time $t_3$, since there is only one bit set in the error vector. A first single-error candidate is thus identified, containing an error at position 8. Since there could be several candidates, we continue the scanning of the packet until the end. At step $t_5$, the algorithm reaches the end of the packet and the list of candidates contains a single entry. Flipping the bit at position 8 in the corrupted packet is the only valid correction if a single error has occurred.

### C. DOUBLE-ERROR CORRECTION

#### 1) PROBLEM WITH STRAIGHTFORWARD EXTENSION OF ALGORITHM 1

The method described in the previous section produces as output the exhaustive list of single-error patterns corresponding to a non-null syndrome at the receiver, given the generator polynomial used and the length of the protected data. To deal with double-error patterns, a straightforward method would be to run the exact same algorithm while appending all the error vectors **e** with two coefficients set to 1 to the candidate list. This approach would be able to output double-error patterns, but cannot ensure that an exhaustive list of such error patterns is provided. Actually, only one specific type of double-error patterns will be output, namely, those in which the double-error pattern covers $n$ bits or less (i.e., are close to each other). The single-error search aims at canceling non-zero values from LSB to MSB. This cancelation is performed thanks to an XOR operation between a shifted version of the generator polynomial and the constantly updated error vector. We can observe that there cannot be more than $n$ bits between the 1 located at the LSB position and the 1 at the

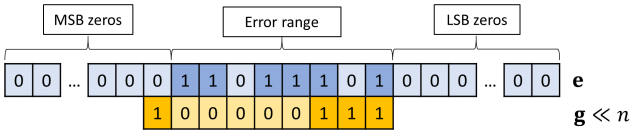


**FIGURE 5.** Illustration of the error range applied to CRC-CCITT-8 ($n = 8$). Canceling LSB non-zero values by performing an XOR operation with a generator polynomial of width $n + 1$ bits produces an error range of $n$ bits.

MSB position, as illustrated in Fig. 5, which represents the error vector during the single-error search. The MSB zeros correspond to the positions still in the original state of **e**, initialized as a null vector, and the LSB zeros correspond to the already canceled positions. Between these two null subvectors we have the possible non-zero positions, with a maximum width of $n$ bits. We will refer to the maximum distance between the first and last non-zero coefficients as the error range of the method. At step $t_9$ of Algorithm 1, the update of the error vector **e** can be expressed as:

$$\begin{aligned}\sum_{i=0}^{m+n-1} e_i x^i \leftarrow \sum_{i=0}^{m+n-1} e_i x^i + \sum_{i=j}^{j+n} g_{(i-j)} x^i \\ = \sum_{i=j+1}^{j+n} (e_i + g_{(i-j)}).x^i \\ = x^{(j+n)} + \sum_{i=j+1}^{j+n-1} (e_i + g_{(i-j)}).x^i \end{aligned} \quad (8)$$

From Eq.(8), it is clear that the whole set of non-null values in the error vector covers $n$ positions at most. In fact, all the values in **e** up to position $j$ are already canceled and set to 0, due to the design of the proposed algorithm, and all positions above $j + n$ are also set to 0 due to the initialization of the error vector (i.e., $\mathbf{e} = \mathbf{0} \oplus \mathbf{s}$). As the highest degree term of the generator polynomial is 1, we can see that position $x^{j+n}$ is set. The other non-null positions are subject to the values of the error vector at its current state and the other terms of the generator polynomial. The range of non-null values is denoted the error range, and is illustrated in Fig. 5. In conclusion, a straightforward extension of Algorithm 1 would only yield error patterns in which errors are within a range of $n$ bits. A different approach is thus required.

#### 2) PROPOSED DOUBLE-ERROR CORRECTION APPROACH

To obtain the exhaustive list of error patterns, we aim at expanding the error range to have it cover the entire length of the protected data. The method we propose is to force a bit to 1 during the process. Forcing a position consists in setting it (or leaving it) to 1 during the single-error search. In other words, it is equivalent to making the hypothesis that a specific position is actually erroneous in the packet. Hence, we force one bit to 1 at position $F_1$ during the process and run the single-error algorithm on the remaining length of the packet. If the bit is already 1, we leave it untouched. Otherwise, setting a bit to 1 is done by applying an XOR operation with $g(x)$

at position $F_1$ in order to maintain the equivalence relation. Throughout the cancelation process with the forced bit set, if a single-error position (denoted hereafter $P_1$) is obtained from the single-error correction algorithm, we determine a double-error pattern with errors at positions $F_1$ and $P_1$.

As we want to get the whole list and we do not know the actual position of the first error, we test each possible forced position in order to output all the double-error patterns associated with the computed syndrome. In the proposed algorithm, we suggest forcing positions starting from LSB to MSB. Moreover, starting from LSB at each tested forced position would lead to a cancelation of the same first positions several over and degrade the computational efficiency. To avoid verifying the same possibilities repeatedly, we store the value of $\mathbf{e}$ when a bit is forced and recall this state to start from it and save computations for the next forced position to test.

Fig. 6 illustrates the complete process for listing the double-error patterns corresponding to the syndrome $s(x) = x^3 + x^2 + 1$ applied to a CRC-4-ITU of generator polynomial $g(x) = x^4 + x + 1$ protecting 6 data bits. In this figure, forced positions are represented as black boxes through time. At step $t_0$, the error vector $\mathbf{e}$ is initialized as a null vector, to which syndrome vector $s(x) = x^3 + x^2 + 1$ is appended. As we start at position 0, and $\mathbf{e}_0$ is already set to 1, we simply jump to the next element and start the single-error search from the next non-zero position, checking at each step if the number of non-zero coefficients in $\mathbf{e}$ equals 2. A first candidate appears at $t_1$, corresponding to errors at positions $(F_1 = 0, P_1 = 6)$. Canceling the next non-zero value would move the operation out of the range of the packet, thus ending the search for a single error for this forced position. At step $t_2$ we recall the initial state of the error vector from $t_0$. The next forced bit to test is at position 1. Hence, we cancel position 0 and let position 1 be set to 1. From $t_3$ to $t_6$, we perform the single-error algorithm on the remaining length. No new error pattern is found. At time $t_7$, we recall the previous state from $t_3$ and cancel the former forced bit at position 1. At $t_8$, the next forced position to test, position 2, is not yet set to 1. We thus have to perform an XOR operation with $\mathbf{g}$ at this position to set it. We identify such cases as dark grey boxes in Fig. 6, at steps $t_8$ and $t_{15}$. We continue this process until reaching the last forced bit position, corresponding to the $m^{th}$ position starting from LSB. We can see at step $t_{18}$ that we cannot perform any XOR operation without going out of the range of the packet. Hence, the algorithm is stopped at $t_{18}$ and outputs the list of error patterns containing two errors corresponding to the received syndrome. The sums of errors in such cases are shown in red font in Fig. 6. The output list contains the following error patterns: $(F_1 = 0, P_1 = 6)$ , $(F_1 = 3, P_1 = 8)$ and $(F_1 = 5, P_1 = 7)$. The proposed approach for double-error correction is exemplified in Fig. 6 and presented in Algorithm 2 using $N = 2$.

### D. N-ERROR CORRECTION

We can further extend the proposed method to deal with any number $N$ of errors in a packet. The strategy applied is the extension of the double-error correction approach described in the previous section.

Much as we forced one position and scanned the remaining length of the packet using the single-error search, we can manage the $N$-error search. In such cases, we set $(N - 1)$ forced bits in the error vector, corresponding to the first $(N - 1)$ errors in the packet, and scan the remaining length using the single-error search to identify the position of the last error in the packet, if it exists. The $(N - 1)$ forced binary errors have to be tested in the packet. The proposed method to generate the list of potential error patterns containing up to $N$ errors is illustrated in Algorithm 2.

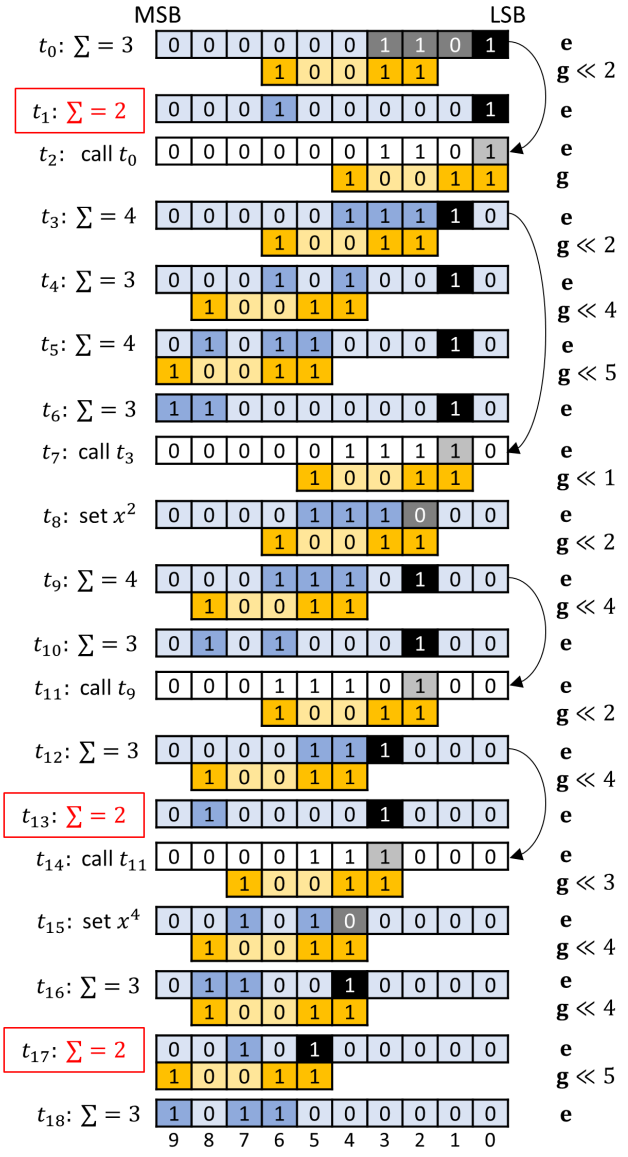


**FIGURE 6.** Visual example of the proposed algorithm applied to double-error correction, performed over CRC-4-ITU (yellow cells) protecting 6 data bits with a syndrome $s(x) = x^3 + x^2 + 1$ (grey cells). Each forced bit position, represented as a black cell, is tested throughout the process to get the exhaustive list of double-error patterns. In this example there are three such cases at steps $t_1$, $t_{13}$ and $t_{17}$, thus $E_2 = \{(0, 6); (3, 8); (5, 7)\}$.

**Algorithm 2** $N$-ErrorPatternsGeneration(**s**,**g**,$n$,$m$,$N$)

**Inputs:**

**s**: the syndrome
**g**: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector
$N$: the maximum number of bit errors considered

**Output:**

$E_N$ the list of valid error patterns up to $N$ bit errors

1: $E_N \leftarrow \{\}$
2: Let **e** be a vector of length $m+n$
3: $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
4: Let **v** be a vector of length $m$
5: **if** sum(**e**) $\leq N$ **then**
6: Add **e** to $E_N$
7: **end if**
8: $k \leftarrow N$
9: **while** $k \geq 1$ **do**
10: **if** $k = 1$ **then**
11: Add SingleErrorCorrection(**s**,**g**,$n$,$m$) to $E_N$
12: **else**
13: Let $\mathcal{F} \leftarrow (0, \ldots, k-2)$
14: $\mathbf{v} \leftarrow$ PositionsToVector($\mathcal{F}$)
15: **while** $\mathcal{F} \neq (m-(k-1), \ldots, m-1)$ **do**
16: start $\leftarrow \max(F_1 - 1, 0)$
17: **for** $j =$ start to $m-1$ **do**
18: **if** $e_j \neq v_j$ **then**
19: $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
20: **if** sum(**e**) $\leq N$ **then**
21: Add **e** to $E_N$
22: **end if**
23: **end if**
24: **if** $j = F_1$ **then**
25: $\mathbf{e}' \leftarrow \mathbf{e}$
26: **end if**
27: **end for**
28: $\mathcal{F} \leftarrow$ UpdateForcedPositions($\mathcal{F}$,$m$)
29: $\mathbf{v} \leftarrow$ PositionsToVector($\mathcal{F}$)
30: $\mathbf{e} \leftarrow \mathbf{e}'$
31: **end while**
32: **end if**
33: $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
34: $k \leftarrow k-1$
35: **end while**
36: Remove duplicate elements in $E_N$
37: Return $E_N$

**Algorithm 3** UpdateForcedPositions($\mathcal{F}$,$m$)

**Inputs:**

$\mathcal{F}$: sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$
$m$: the length of the payload vector

Note that $k = \text{len}(\mathcal{F}) + 1$, with len($\mathcal{F}$) being the number of elements in the list $\mathcal{F}$

**Output:**

$\mathcal{F}'$: the updated sorted list of forced positions

1: **if** $F_{k-1} < (m-1)$ **then**
2: $F_{k-1} \leftarrow F_{k-1} + 1$
3: Return $\mathcal{F}' \leftarrow (F_1, \ldots, F_{k-1})$
4: **else**
5: **for** $i = k-2$ to 1 **do**
6: **if** $F_i < F_{i+1} - 1$ **then**
7: $F_i \leftarrow F_i + 1$
8: $j \leftarrow i$
9: **while** $j < k-1$ **do**
10: $F_{j+1} \leftarrow F_j + 1$
11: $j \leftarrow j + 1$
12: **end while**
13: Return $\mathcal{F}' \leftarrow (F_1, \ldots, F_{k-1})$
14: **end if**
15: **end for**
16: **end if**

**Algorithm 4** PositionsToVector($\mathcal{F}$)

**Inputs:**

$\mathcal{F}$: sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$.

Note that $k = \text{len}(\mathcal{F}) + 1$, with len($\mathcal{F}$) being the number of elements in the list $\mathcal{F}$

**Output:**

**v**: the corresponding vector of forced positions

1: $\mathbf{v} \leftarrow \mathbf{0}$
2: **for** $i = 1$ to $k-1$ **do**
3: $v_{F_i} \leftarrow 1$
4: **end for**
5: Return **v**

We now present the key steps of the proposed algorithm , while the corresponding flowchart is given in Fig. 8:

**3:** The binary vector of length $M$ representing the error vector **e** is initialized to $m$ zeros, followed by $n$ values, corresponding to the computed syndrome **s**.

**5-6:** We first check if the number of non-zero values in this initial vector **e** is less than or equal to the targeted number of errors $N$. If so, a first candidate is added to the list $E_N$.

**8-9:** The local variable $k$ represents the current number of errors considered. $k$ is initialized to $N$, then decreased at each main loop of the algorithm to consider every number of errors from $N$ to 1.

**10-11:** In the last loop, the variable $k$ equals 1. In this case, no forced position must be set and the single-error correction

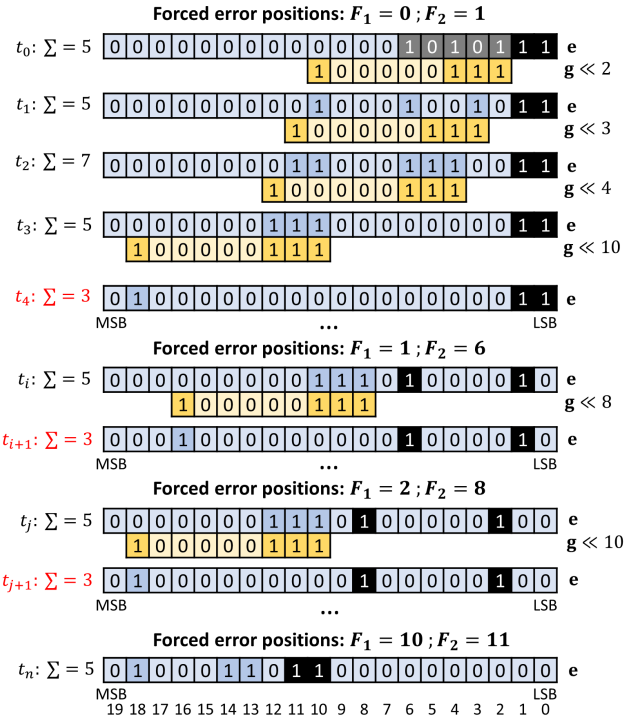


**FIGURE 7.** Illustrative example of the proposed algorithm performed over CRC-8-CCITT (yellow cells) protecting 10 data bits, where $N = 3$ and $s(x) = x^6 + x^4 + x^2 + x + 1$ (grey cells). Forced bit positions are represented as black cells in the vector e. Three solutions are valid candidates in this example, where $\sum$, representing sum(e), equals 3 (shown in red font). Here, $E_3 = \{(0, 1, 18); (1, 6, 16); (2, 8, 18)\}$.

algorithm is performed. The output candidate list is then appended to the global candidate list $E_N$.

**13:** The sorted list of forced positions $\mathcal{F}$ is initialized to the $(k-1)$ LSB values at the first iteration. At this step, the set of forced positions $\mathcal{F} = (F_1 = 0, F_2 = 1, \ldots, F_{k-1} = (k-2))$. The $(k-1)$ forced positions in the set $\mathcal{F}$ are ordered such that $F_1 < F_2 < \ldots < F_{k-1}$.

**14:** The binary vector is set according to the forced positions in $\mathcal{F}$. In Algorithm 4, the bits in $\mathbf{v}$ corresponding to forced positions in $\mathcal{F}$ are set to 1. The other bits in $\mathbf{v}$ are set to 0.

**15:** The forced positions will then be updated to cover the entire set of possible fixed error positions (until the forced positions are the $(k-1)$ MSB positions). For a packet of $M$ bits, there are $\binom{M-n}{k-1}$ such positions, thanks to the range of the XOR operation performed. After setting these forced positions, we are aiming at finding the last error by conducting the single-error algorithm on the remaining part of the packet.

**16:** In order to save computations, we use as a starting point the previously obtained vector $\mathbf{e}$ after cancelation of its LSB positions up to $F_1$, the LSB position we forced. With this approach we will not have to cancel the same first positions at each iteration as we increase $F_1$.

**17:** We perform a scan on the remaining length of the error vector $\mathbf{e}$, from LSB to MSB (i.e., single-error search).

**18-19:** At each position $j$, we compare the values of $e_j$ and $v_j$ to determine if the $j^{th}$ position corresponds to a forced position. If $v_j$ and $e_j$ are both set to 0 or 1, it means that position $j$ either must not be forced (to 1) and is already set to 0, or must be forced, but is already set to 1. In both cases, the algorithm simply jumps to the next element since what is required is already in place. However, when these two elements are set to different values, it corresponds to the cases where the position $j$ has to be canceled and set to 1, or where the position $j$ has to be forced to 1, but is set to 0. In both cases, an XOR operation with $\mathbf{g}$ must be performed to maintain the equivalence relation and obtain what is required.

**20-22:** At each stage, the number of non-zero coefficients in the newly accumulated $\mathbf{e}$ is observed, similarly to steps 5-6. Whenever $\mathbf{e}$ contains $N$ errors or less, a candidate is added to the list $E_N$.

**25:** We store the state of $\mathbf{e}$ in a vector $\mathbf{e}'$ to avoid re-canceling the same first LSBs at the next iteration.

**28:** At the end of each scan, the vector of forced positions is updated using the UpdateForcedPosition function illustrated in Algorithm 3. In this algorithm, the $(k-1)$ forced positions are successively updated to cover the entire message. At step 1, we check if the MSB forced position has reached its final position. If not, we increase its value by one. If it has reached its final position, we successively check forced positions from MSB to LSB at step 5. When a forced position can be increased, we reversely update the other positions, from LSB to MSB.

**29:** Each time the set of forced positions is updated, the binary vector $\mathbf{v}$ is modified.

**30:** We recall the state $\mathbf{e}'$ to start from it at the next iteration.

**33:** We recall the initial state (syndrome) when we update the number of errors to consider.

Fig. 7 shows a visual example of the algorithm applied to a CRC-8-CCITT, which has a generator polynomial $g(x) = x^8 + x^2 + x + 1$. This example illustrates a triple-error management, with $\mathbf{v}$ having two non-zero values, represented as black boxes in Fig. 7. Four stages are represented here: the initial stage, where the forced error positions are $(F_1 = 0; F_2 = 1)$, two different stages that produce a valid candidate, namely $(F_1 = 1; F_2 = 6)$ and $(F_1 = 2; F_2 = 8)$, and the final step, with the last two forced positions $(F_1 = 10; F_2 = 11)$. By definition, these positions must remain set to 1 at the end of the scan. The other non-null values in $\mathbf{e}$ must be canceled from LSB to MSB, using the single-error search method. At each step, we successively add left-shifted versions of $\mathbf{g}$ at these positions and if the sum of non-null values in $\mathbf{e}$ is equal to 3 $(N = 3)$, then $\mathbf{e}$ is considered as a valid candidate. We can observe that this example produces three valid error patterns with error positions $(F_1 = 0; F_2 = 1; P_1 = 18)$, $(F_1 = 1; F_2 = 6; P_1 = 16)$ and $(F_1 = 2; F_2 = 8; P_1 = 18)$, shown in red font in Fig. 7. The design of Algorithm 2 yields a total complexity, measured in number of XOR operations, of $O(m^N)$. Testing the entire error pattern to determine which would match the received syndrome (i.e., brute force scheme) would have a complexity of $O(m^{N+1})$. We can note that the

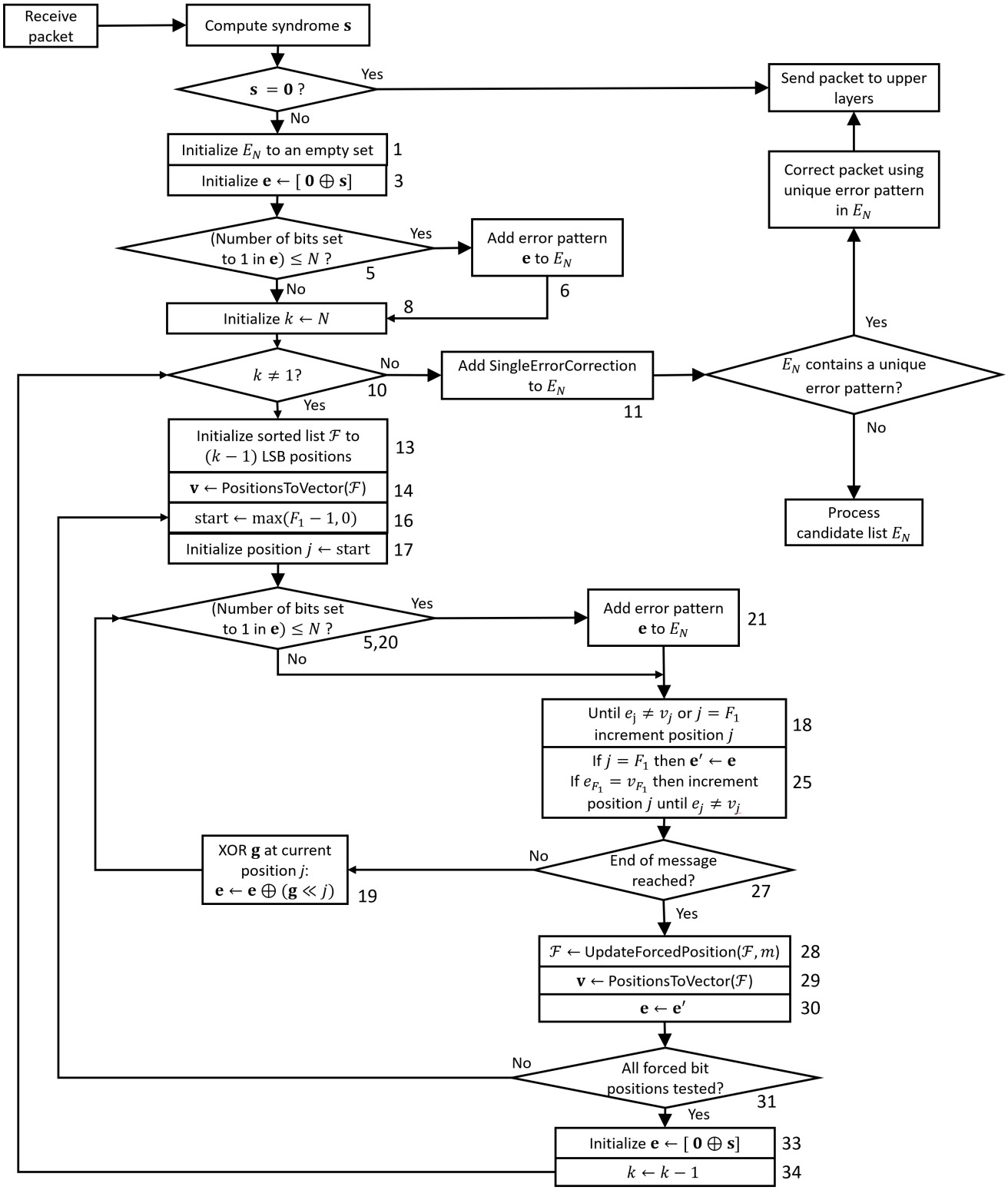


**FIGURE 8.** Flowchart of the proposed method algorithm for correcting multiple errors in the packet. Numbers show the corresponding steps in Algorithm 2.

complexity increases significantly with the number of errors considered $N$. We thus recommend using the algorithm when $N$ is low, depending on the processing time constraints of the targeted application. It is important to note that correcting a single error using algorithm 1 is not computationally more complex than performing a classic CRC check at the receiver.

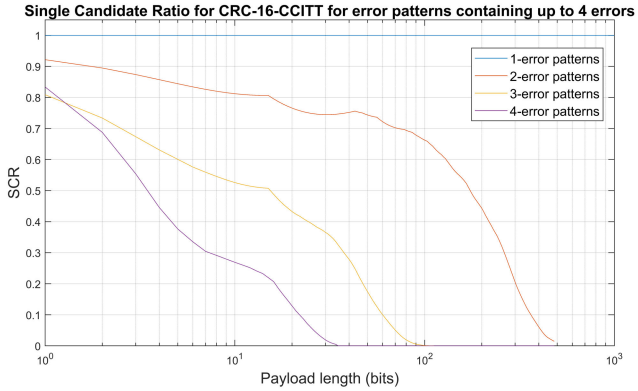


**FIGURE 9.** Single Candidate Ratio (SCR) for a payload length from 0 to 500 bits protected by a CRC-16-CCITT [25] of generator polynomial $g(x) = x^{16} + x^{12} + x^5 + 1$, considering up to 4 errors.

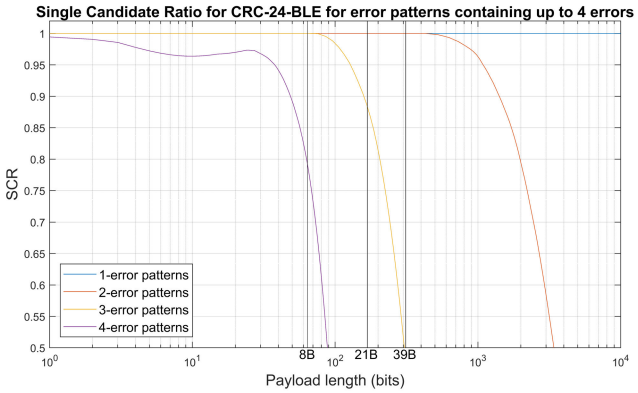


**FIGURE 10.** Single Candidate Ratio (SCR) for a payload length from 0 to 10 000 bits protected by a CRC-24-BLE [17] of generator polynomial $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$, considering up to 4 errors.

## III. SIMULATION AND RESULTS

In this section, we present the theoretical and simulation performance of the proposed method, as compared to the lookup table approaches from the literature. Finally, we apply our method to Bluetooth Low Energy used in the IoT and compare its performance to state-of-the-art methods.

### A. CORRECTION RATE

In this section, we evaluate the performance of the proposed error correction method. It is able to instantly correct the packet when there is only one candidate in the output candidate list. Hence, the performance can be expressed as the percentage of error cases that produce only one candidate in the list. We will refer to such a percentage as the Single Candidate Ratio (SCR). The SCR is a function of three parameters: the generator polynomial used, the length of the protected data and the number of errors considered. Given a generator polynomial $g(x)$, we denote the $N$-error patterns for a packet length $m$ leading to a single candidate as SinglePatterns$(m, N)$ and the total number of possible $N$-error patterns for the same length $m$ as TotalPatterns$(m, N)$, and we can express the SCR as:

$$\text{SCR}(m, N) = \frac{\text{SinglePatterns}(m, N)}{\text{TotalPatterns}(m, N)} \tag{9}$$

where SinglePatterns$(m, N)$ was determined by running the algorithm over all the error cases and TotalPatterns$(m, N) = \binom{m}{N}$. SCRs are thus not estimated but computed over the entire set of possible error patterns. We verify that when the length and the number of errors considered increase, the SCR decreases rapidly. Moreover, the length of the generator polynomial, as well as the number of non-zero coefficients, modify the way the SCR decreases as the length of the protected data increases.

In Figs. 9 and 10, we show the evolution of the SCR for different generator polynomials and different numbers of errors considered. We observe in Fig. 10 that when a long generator polynomial is used and few errors are considered, the SCR stays at 100% up to a significant length. On the other hand, a short generator polynomial leads to a faster decrease of the SCR as the packet's length increases, as illustrated in Fig. 9. When the SCR is 100% up to a certain threshold length for $N$ errors, it means that if $N$ errors or less occur during the transmission of the packet, these errors can be identified with a certainty of 100% and without any possible ambiguity when the packet's length is lower than this threshold. From this threshold, the SCR does not fall to zero immediately. Depending on the generator polynomial chosen, it can still be at a high percentage level up to a significant packet size. If we take the example of CRC-24 used in the Bluetooth Low Energy protocol [17] (of generator polynomial $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$), the SCR is still above 80% for a payload of up to 2000 bits when considering that two errors occurred in the packet. For three and four errors, this number decreases greatly, but is still over 80% for up to 220 bits for three errors and up to 85 bits when considering four errors. The applications targeted by the CRC-24 used in the Bluetooth Low Energy standard concern the Internet of Things (IoT) [23], [24]. In IoT environments, the average packet payload is often just a few bytes in size. Consequently, the proposed error correction method will be able to instantly correct most error patterns up to four errors, and even 100% of error patterns up to two errors, given a packet of 450 bits or less.

However, if the packet is highly corrupted, it may conceivably produce a syndrome the algorithm would recognize as the result of a low number of errors. In such a case, we would have a miscorrection. This corresponds, however, to very disadvantageous cases for all error correction methods. In fact, there is no error correction method that guarantees the validity of the reconstructed sequence. However, what we have is no more problematic than the case of highly corrupted packet yielding a CRC syndrome of zero, letting the receiver believe there is no error.

### B. MEMORY REQUIREMENTS

The proposed method does not require storing any table. In contrast, the main drawback of lookup approaches is their memory requirements. The table must be stored in the

**TABLE 1. Memory requirements for storing the lookup tables considering a payload of 1500 bytes for several CRC lengths and number of errors considered with implicit and explicit error positions.**

| Nb Errors $N$ | CRC-8 | | CRC-16 | | CRC-24 | | CRC-32 | |
|---|---|---|---|---|---|---|---|---|
| | Implicit | Explicit | Implicit | Explicit | Implicit | Explicit | Implicit | Explicit |
| 1 | 12 kB | 36 kB | 24 kB | 48 kB | 36 kB | 60 kB | 48 kB | 72 kB |
| 2 | 72 MB | 360 MB | 144 MB | 432 MB | 216 MB | 504 MB | 288 MB | 576 MB |
| 3 | 288 GB | 2.02 TB | 576 GB | 2.30 TB | 864 GB | 2.60 TB | 1.16 TB | 2.88 TB |
| 4 | 864 TB | 7.77 PB | 1.73 PB | 8.64 PB | 2.59 PB | 9.50 PB | 3.45 PB | 10.4 PB |

| Syndrome | Positions | | |
|---|---|---|---|
| 0000 0000 0000 0000 0000 0111 | 0 | 1 | 2 |
| 0000 0000 0000 0000 0000 1011 | 0 | 1 | 3 |
| 0000 0000 0000 0000 0001 0011 | 0 | 1 | 4 |
| 0000 0000 0000 0000 0010 0011 | 0 | 1 | 5 |
| ... | | | |
| 1111 1101 0010 1110 1011 0100 | 567 | 2504 | 8956 |
| 0110 0101 0001 0110 1111 0011 | 567 | 2504 | 8957 |
| 0101 0101 0110 0000 0010 0110 | 567 | 2504 | 8958 |
| ... | | | |
| 1111 0010 0010 1101 0000 0101 | 11996 | 11997 | 11998 |
| 0011 0111 1101 1000 1111 1111 | 11996 | 11997 | 11999 |
| 0101 0101 0010 0010 0000 0010 | 11996 | 11998 | 11999 |
| 1110 0100 0101 1100 0101 0001 | 11997 | 11998 | 11999 |

**FIGURE 11. Examples of the explicit design of a lookup table containing all triple-error patterns for packet length up to 12000 bits.**

receiver's memory, as shown in Fig. 11. On very small-sized packet lengths as considered in [6] and [7], the lookup table represents a viable solution. When dealing with large packets, the required memory increases very rapidly. This rapid increase is also seen as the number of errors considered increases.

We evaluate the memory required when considering a specific number of errors and for each common syndrome length. Two different approaches are used to construct the lookup table. In both cases, the number of entries in the table corresponds to the number of possible error patterns. From this definition, it becomes clear that a lookup table that considers a packet length $M$ and $N$ errors would have $\binom{M}{N}$ rows. At each of these entries, the table must store the non-null syndrome for every possible error pattern. The syndrome is stored as a 1 to 4-byte number if we consider codes from CRC-8 to CRC-32 (used in Ethernet [21], of generator polynomial $g(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$). To retrieve error positions, two strategies are used:

- The first one is to explicitly store the error positions associated with the syndrome in the table, as numbers coded on 16 bits (2 bytes) for each entry. There are $N$ such numbers per row. Using this lookup table design, the required memory, denoted $B_{exp}$, is expressed as:

$$B_{exp} = \binom{M}{N} \times [\text{length}(\mathbf{s}) + (2 \times N)] \tag{10}$$

  where $M$ is the total length of the packet, $N$ is the number of errors considered, and length($\mathbf{s}$) is the size in bytes of the syndrome associated with the CRC used. The expression $(2 \times N)$ is the representation of $N$ 2-byte numbers per row, representing the positions of the $N$ errors considered. This implementation allows finding directly the error patterns associated with the syndrome but at a significant memory cost.
- The second strategy uses an implicit error position. With this approach, the lookup table does not need to store $N$ 2-byte numbers per entry, which reduces the total memory requirements by up to 9 times when considering a CRC-8 and four errors, as compared to the aforementioned strategy. The memory requirements, denoted $B_{imp}$, can now be expressed as:

$$B_{imp} = \binom{M}{N} \times \text{length}(\mathbf{s}) \tag{11}$$

  However, such a strategy involves more calculations to update the error pattern corresponding to the syndrome as it navigates through the table.

We note that depending on the constraints present, one can choose among the two proposed designs to either save memory storage or save computations at the receiver side. Table 1 illustrates the memory requirements for both explicit and implicit implementations when considering large packets of 1500 bytes. We can see a significant increase in the memory requirements when considering each additional error. For such a packet length, considering three or four errors using a lookup table approach would be intractable.

### C. COMPUTATIONAL TIME COMPARISON

In terms of processing time, we ran the C implementation of the proposed algorithm for a single- and a double-error correction on a Raspberry Pi model 3B+ [22]. For comparison purpose, we also implemented a table approach capable of considering every single-error position for packets up to 1500 bytes. We executed both algorithms for packets of different lengths, from a few bits to the maximum size available here, set to 1500 bytes. The Raspberry model 3B+ used to conduct the experiment is equipped with a System on a Chip (SoC) Broadcom BCM2837BO with an ARM Cortex-A53 quad-core processor at 1.4 GHz and 1 GB SDRAM LPDDR2.

Figs. 12 and 13 show the relative time for the error correction method on single- and double-error patterns,

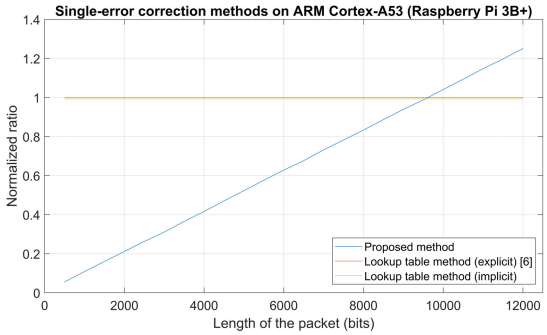


**FIGURE 12.** Evolution of the normalized time ratio to run the proposed method compared to lookup table-based approaches (explicit and implicit) for a single error in the packet depending on the length of the packet. The normalized time ratio corresponds to 155$\mu s$ in this case.

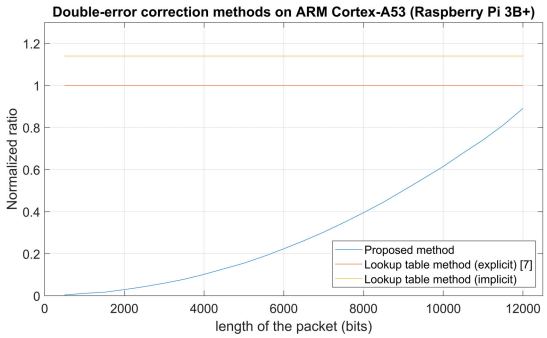


**FIGURE 13.** Evolution of the normalized time ratio to run the proposed method compared to lookup table-based approaches (explicit and implicit) for two errors in the packet depending on the length of the packet. The normalized time ratio corresponds to 1.4$s$ in this case.

respectively. The proposed method's complexity is compared to both lookup table approaches (i.e., explicit and implicit) for different packet sizes. Lookup table-based approaches have a constant complexity since they must always check every entry of the table prior to conducting error correction to ensure that all candidates are identified. When considering a single error, both table-based approaches are of equal complexity, and the conversion from explicit to implicit is straightforward. For two errors, however, the implicit method is 10 to 15% slower due to the computations required to convert the table index into error positions. We note that for a large payload, the methods are similar in terms of computational complexity. The lookup table approach is still faster when considering single errors in large packets. However, as the packet becomes smaller with respect to the maximum allowed packet size, the proposed method surpasses the lookup approach due to its adaptability to the received packet size. The method can be more than 10 times faster than lookup methods for very small packets, and the gain in speed is even greater when double-error correction is considered. When used as part of a standalone error correction process, the algorithm performs at its maximum in terms of both correction rate and complexity for small packets or CRC-protected headers. In such cases, the SCR is significantly high or at a maximum for multiple-error correction.

Comparing the proposed algorithm to the lookup table approaches in the literature, we can verify that it provides improved capabilities in two main respects:

1) **Flexibility**. Our method is more flexible than fixed-length lookup tables since it is not based on a specific packet size, but rather, is dynamically applied to protected data, and is thus adaptive to the data length. Consequently, the method will provide full coverage for any packet length. Furthermore, any generator polynomial, apart from the input parameter can be used with the proposed algorithm without modification. Lookup tables must be entirely recomputed when the generator polynomial considers changes. Alternatively, a table should be stored for each generator polynomial of interest, which significantly increases memory requirements.
2) **Memory-free multiple-error correction**. The proposed method does not assume that only single errors are likely to occur. Even if this scenario can still be supported by setting the number of errors to 1 as the input parameter, we can also assume that up to $N > 1$ errors are possible and consider the whole set of possible candidates up to this number. A lookup table approach is able to list such error patterns but needs an intractable amount of memory storage to consider the whole set of $N$-error cases in large packets. In order to optimize the management of the number $N$, further work can be carried out to dynamically choose it by extracting information about channel conditions, such as the channel Signal-to-Noise Ratio (CSNR) estimation at the physical layer or the Receiver Signal Strength Indicator (RSSI). The received RSSI level can be mapped to the crossover probability by measuring an average BER for each RSSI level. If this BER estimation is low enough in terms of the length of the packet, we can set the parameter $N$ to be 1.

### D. APPLICATION TO IoT

Considering the high performance of our method on small packets protected by strong generator polynomials, applying the proposed algorithm to the IoT domain can be highly desirable. A study of error distribution in a real environment of CRC-protected packets applied to the Internet of Things (IoT) [24] domain is proposed in [11]. The authors consider both Bluetooth Low Energy (BLE) packets protected by a CRC-24 and IEEE 802.15.4 [25] packets protected by CRC-16-CCITT. Two packet sizes, 21 bytes and 39 bytes, are considered. The results of the experiments are represented in Table 2 which shows that over 50% of the erroneous packets contain fewer than three errors in any selected scenario. Moreover, more than 40% contain two errors or less, making it an ideal context to evaluate our proposed method's performance. As noted the authors of [11], considering only slightly damaged packets can thus still enable a significant recovery rate. When soft information is unavailable, the authors of [11]

**TABLE 2. Error distribution in real environment for BLE and IEEE 801.15.4 and two packet sizes.**

| Number of bit errors | BLE | | IEEE 802.15.4 | |
|---|---|---|---|---|
| | 21B | 39B | 21B | 39B |
| 1 | 18% | 16% | 11% | 10% |
| 2 | 28% | 27% | 30% | 27% |
| 3 | 12% | 11% | 15% | 16% |
| >3 | 42% | 46% | 44% | 47% |

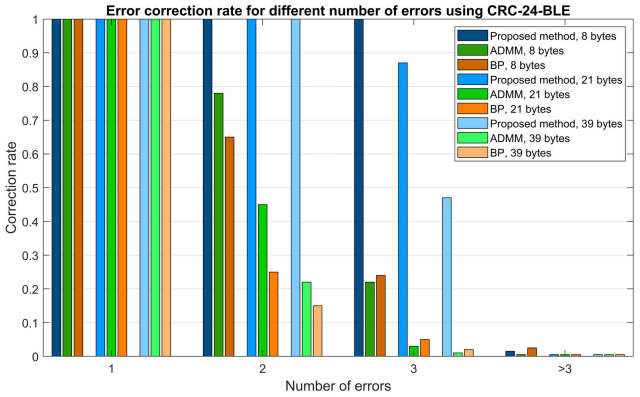


**FIGURE 14. Error correction rate of the proposed method compared to two methods recently proposed in [11] for different number of errors in the packet.**

propose to use a received packet's RSSI to determine the Bit Error Rate (BER).

In [11], the authors present the average correction rate of their methods when a specific number of errors occur in the packet. The simulation results can be seen in Fig. 14, considering three payload sizes: 8 bytes, 21 bytes and 39 bytes. To compare our algorithm with these approaches, we tested an exhaustive set of error patterns for each size and each number of errors to get the average correction rate over all possible error cases. We applied the algorithm for each error case and checked the resulting list at the end of the process. The correction is considered successful only if the actual error pattern is the only candidate in the output list. If there are no or several candidates in the list, the packet is considered lost. For the Alternating Direction Method of Multipliers (ADMM) and Belief Propagation (BP) [11], the simulation results in Fig. 14 show a maximum correction rate for single-error correction for all methods considered. For double-error correction, only the proposed method is able to achieve a 100% error correction. ADMM can correct an average of 80% for 8-byte payloads, which falls to less than 25% for 39-byte payloads. The results considering three errors are even more significant. The proposed method offers a 100% error correction rate for 8-byte payloads, whereas both ADMM and BP achieve 25%. When the payload length increases, the proposed method still can correct 86% and 47% for 21- and 39-byte payloads, respectively. Other methods can achieve a maximum of 5% error correction for such payloads.

These results can be retrieved in Fig. 10, where the three vertical bars correspond to the three payloads considered here. We can see that the correction rate for more than three errors is very low for all methods. In fact, it involves considering every error pattern containing more than three errors, which leads to a poor ratio since as the number of errors considered increases, the SCR decreases, becoming zero for large numbers of errors. However, we can note that we can still operate on four errors for small packets, as illustrated in Fig. 10 for 8-byte payloads, where the SCR is still 78%.

In [11], the authors propose a configurable iterative decoding process, which means that its performance will depend on the number of iterations performed on the corrupted packet. The results provided here consider 1000 iterations at the decoder. The timing for this decoding applied to the fastest method (ADMM) takes an average of 85 ms for 21-byte packets on a desktop computer with an Intel i7 3.1 GHz CPU, 8 GB RAM and Microsoft Visual C++ 2010 Compiler. We tested our method on a desktop computer with an Intel i7 3.4 GHz CPU, 8 GB RAM and GCC compiler, and we noted that depending on the number of errors to consider, it takes an average time ranging from 2 $\mu$s for single-error correction to 8 ms for three errors or less. Double-error correction takes an average of 150 $\mu$s. Therefore, the proposed method not only allows dramatically correcting more double- and triple-error cases, but it is also significantly faster than the state-of-the-art methods presented in [11].

### E. FUTURE WORK

In this paper, we have considered our algorithm as a standalone process that can only correct a packet when its output list contains a single element. In order to further increase the proposed method's error correction performance, it can be jointly used with other methods providing a list of potential error patterns as their output. For example, the work on UDP checksum proposed in [9], [26], [27] can be combined with our algorithm. Crosschecking both candidate lists would generate a matching list with a reduced number of entries. If our method is used in addition to the UDP checksum method, greater protected data lengths or a higher number of errors can be targeted for applications such as error correction on Ethernet frames, where a CRC covers the entire packet. Similarly, we could eliminate candidates leading to wrong values of known protocol fields, such as constant and predictable fields in the protocol's header (reserved and version fields are constant values during a communication, and some fields such as the sequence number in RTP are predictable since they are increased by 1 at each new packet throughout the communication). Some methods which consider a MAP approach have already proposed to use a CRC lookup table to validate their reconstruction, as described in [4] on Polar codes [28].It could be beneficial to compute only the probability of valid candidates rather than considering the whole set of possible sequences, determining their probability of being sent, and finally checking their CRC compliance.

## IV. CONCLUSION

In this work, we have proposed a novel algorithm to correct transmission errors within data covered by a CRC, using the computed non-null syndrome at the receiver. This method is able to instantly correct single errors if the protected data length does not exceed the period of the generator polynomial. This method is also able to correct multiple errors in small-sized packets, as used in the Bluetooth Low Energy standard. In such an environment, the proposed method achieves better error correction rates than the state-of-the-art methods considering up to three errors in the packet. The standalone error correction rate in BLE is at a maximum for single-, double- and some triple-error cases presented.

When instant correction is not possible, the algorithm still generates the list of all the possible error patterns that lead to the computed syndrome, according to a maximum number of errors considered. This list is usually small if we consider a reasonable number of errors. Further work to improve this method should use it in addition to existing methods that output a list of candidates. Crosschecking the lists of different methods would reduce the number of valid candidates, which would lead to fewer sequences to test or even to a reduction of the list size to a single candidate, allowing instant correction of damaged packets.

## REFERENCES

[1] J. Sobolewski, "Cyclic redundancy check," in *Encyclopedia of Computer Science*. Hoboken, NJ, USA: Wiley, 2003.

[2] J. Postel, *Transmission Control Protocol*, vol. 793. Fremont, CA, USA: IETF, RFC, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt

[3] R. T. Braden, D. A. Borman, and C. Partridge, *Computing the Internet Checksum*, vol. 1071. Fremont, CA, USA: IETF, RFC, Sep. 1988. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1071.txt

[4] K. Niu and K. Chen, "CRC-aided decoding of polar codes," *IEEE Commun. Lett.*, vol. 16, no. 10, pp. 1668–1671, Oct. 2012.

[5] X. Liu, S. Wu, X. Xu, J. Jiao, and Q. Zhang, "Improved polar SCL decoding by exploiting the error correction capability of CRC," *IEEE Access*, vol. 7, pp. 7032–7040, 2018.

[6] S. Shukla and N. W. Bergmann, "Single bit error correction implementation in CRC-16 on FPGA," in *Proc. IEEE Int. Conf. Field- Program. Technol.*, Dec. 2004, pp. 319–322.

[7] S. Babaie, A. K. Zadeh, S. H. Es-hagi, and N. J. Navimipour, "Double bits error correction using CRC method," in *Proc. 5th Int. Conf. Semantics, Knowl. Grid*, 2009, pp. 254–257.

[8] A. S. Aiswarya and A. George, "Fixed latency serial transceiver with single bit error correction on FPGA," in *Proc. Int. Conf. Trends Electron. Informat. (ICEI)*, May 2017, pp. 11–12.

[9] F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Checksum-filtered list decoding applied to H.264 and H.265 video error correction," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 8, pp. 1993–2006, Aug. 2018.

[10] E. Tsimbalo, X. Fafoutis, and R. Piechocki, "Fix it, don't bin it!–CRC error correction in Bluetooth low energy," in *Proc. IEEE 2nd World Forum Internet Things (WF-IoT)*, Dec. 2015, pp. 286–290.

[11] E. Tsimbalo, X. Fafoutis, and R. J. Piechocki, "CRC error correction in IoT applications," *IEEE Trans. Ind. Informat.*, vol. 13, no. 1, pp. 361–369, Feb. 2017.

[12] F. Caron and S. Coulombe, "Video error correction using soft-output and hard-output maximum likelihood decoding applied to an H.264 baseline profile," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 7, pp. 1161–1174, Jul. 2015.

[13] P. Duhamel and M. Kieffer, *Joint Source-Channel Decoding: A Cross-Layer Perspective With Applications in Video Broadcasting*. New York, NY, USA: Academic, 2009.

[14] G. Zhang, R. Heusdens, and W. B. Kleijn, "Large scale LP decoding with low complexity," *IEEE Commun. Lett.*, vol. 17, no. 11, pp. 2152–2155, Nov. 2013.

[15] S. Sankaranarayanan and B. Vasic, "Iterative decoding of linear block codes: A parity-check orthogonalization approach," *IEEE Trans. Inf. Theory*, vol. 51, no. 9, pp. 3347–3353, Sep. 2005.

[16] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, Standard P802.11bc, Dec. 2016.

[17] (2013). *Specification of the Bluetooth System. Core Version 4.1*. Bluetooth SIG. [Online]. Available: http://www.bluetooth.com

[18] S. Boyd and L. Vandenberghe, *Introduction to Applied Linear Algebra—Vectors, Matrices, and Least Squares*. Cambridge, U.K.: Cambridge Univ. Press, 2018.

[19] J. Arndt, "Binary polynomials," in *Matters Computational*. Berlin, Germany: Springer, 2011, pp. 822–863.

[20] N. Bhatnagar, *Mathematical Principles of the Internet, Volume 1: Engineering*. London, U.K.: Chapman & Hall, Dec. 2018.

[21] *IEEE Standard for Ethernet*, Standard 802.3-2018, IEEE Standard Association, 2018. [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html

[22] *Raspberry PI 3 Model B+*. Accessed: May 31, 2020. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

[23] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for smart cities," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 22–32, Feb. 2014.

[24] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, Jun. 2015.

[25] *IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, Standard 802.15.4-2006, 2007.

[26] F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, and P. Corlay, "The impact of H.264 non-desynchronizing bits on visual quality and its application to robust video decoding," in *Proc. 12th Int. Conf. Signal Process. Commun. Syst. (ICSPCS)*, Dec. 2018, pp. 17–19.

[27] F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Low complexity H.264 list decoder for enhanced quality real-time video over IP," in *Proc. IEEE 30th Can. Conf. Electr. Comput. Eng. (CCECE)*, Apr. 2017, pp. 1–6.

[28] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

**VIVIEN BOUSSARD** (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in broadcast engineering from Université Polytechnique Hauts-de-France, Valenciennes, France, in 2015 and 2017, respectively. He is currently pursuing the Ph.D. degree with the Department of Software Engineering and Information Technology, École de Technologie Supérieure, Université du Québec, Montreal, QC, Canada, and the Department of Opto-Acousto-Electronics, Institute of Electronics, Microelectronics, and Nanotechnologies, Valenciennes, (UMR 8520). His current research interests include error correction, vehicular communication, image, and video transmission.

**STÉPHANE COULOMBE** (Senior Member, IEEE) received the B.Eng. degree in electrical engineering from the École Polytechnique de Montréal, Canada, in 1991, and the Ph.D. degree in telecommunications (image processing) from INRS-Telecommunications, Montreal, in 1996. He is currently a Professor with the Department of Software and IT Engineering, École de technologie supérieure (ÉTS is a constituent of the Université du Québec network). From 1997 to 1999, he was with Nortel Wireless Network Group, Montreal, and from 1999 to 2004, he worked with the Nokia Research Center, Dallas, TX, USA, as Senior Research Engineer and as a Program Manager with the Audiovisual Systems Laboratory. He joined ETS, in 2004, where he currently carries out research and development on video processing and systems, compression, and transcoding. From 2009 to 2018, he has held the Vantrix Industrial Research Chair in Video Optimization.

**FRANÇOIS-XAVIER COUDOUX** (Senior Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from Université Polytechnique Hauts-de-France, Valenciennes, France, in 1991 and 1994, respectively. Since 2004, he has been a Professor with the Department of Opto-Acousto-Electronics, Institute of Electronics, Microelectronics, and Nanotechnologies, Valenciennes, (UMR 8520). His research interests include telecommunications, multimedia delivery over wired and wireless networks, image quality, and adaptive video processing.

**PATRICK CORLAY** received the Ph.D. degree from Université Polytechnique Hauts-de-France, Valenciennes, France, in 1994. Since 2016, he has been a Professor with the Department of Opto-Acousto-Electronics, Institute of Electronics, Microelectronics, and Nanotechnologies, France (UMR 8520). His current research interests are in telecommunications, multimedia delivery over wired and wireless networks, and optimal quality of service for video transmission.

# Erratum to "Table-Free Multiple Bit-Error Correction Using the CRC Syndrome"

Zouhair Ziani and Stéphane Coulombe
Validated by Vivien Boussard, François-Xavier Coudoux and Patrick Corlay

August 2023

In [1], the proposed Algorithm 2 for $N$-error correction doesn't select some particular candidates in which the error positions are in the MSBs since positions greater than $m-1$ can not be chosen as forced bits. For example, the 3-error search performed over CRC-8-SAE-J1850 ($g(x) = x^8 + x^4 + x^3 + x^2 + 1$) protecting 12 data bits, where the syndrome is $s(x) = x^4$, doesn't detect the pattern (12, 13, 19). In order to solve this issue, an adaptation SingleErrorCorrectionMult (Algorithm 5) of the single-error correction algorithm is proposed. This version allows the multiple-error candidates observed during the process to be added to $E_N$ alongside the single-error patterns, and thus should be called in the case $k = 1$ of Algorithm 2. The flowchart (Fig. 8.) has to include this change, and the explanation of the 10-11th lines must be updated:

> **10-11:** In the last loop, the variable $k$ equals 1. In this case, no forced position must be set and a modified version of the single-error correction algorithm, which accepts solutions where $\mathbf{e}$ contains $N$ errors or less, is performed (see Algorithm 5). The solutions are appended to the global candidate list $E_N$.

Two other errors occurred:

- In Fig. 7., the CRC code protects 12 bits instead of 10, and since the syndrome length is 8 bits, an additional bit should be colored in gray.
- In section III.A., the total number of possible $N$-error patterns for the payload length $m$ is TotalPatterns$(m, N) = \binom{m+n}{N}$.

## References

[1] Vivien Boussard, Stéphane Coulombe, François-Xavier Coudoux, and Patrick Corlay. Table-Free Multiple Bit-Error Correction Using the CRC Syndrome. *IEEE Access*, 8:102357–102372, 2020.

**Algorithm 2** $N$-ErrorPatternsGeneration$(\mathbf{s}, \mathbf{g}, n, m, N)$

**Inputs:**

$\mathbf{s}$: the syndrome
$\mathbf{g}$: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector
$N$: the maximum number of bit errors considered

**Output:**

$E_N$: the list of valid error patterns up to $N$ bits

$E_N \leftarrow \{\}$
Let $\mathbf{e}$ be a vector of length $m + n$
$\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
Let $\mathbf{v}$ be a vector of length $m$
**if** sum$(\mathbf{e}) \leq N$ **then**
    Add $\mathbf{e}$ to $E_N$
**end if**
$k \leftarrow N$
**while** $k \geq 1$ **do**
    **if** $k = 1$ **then**
        Add SingleErrorCorrectionMult$(\mathbf{s}, \mathbf{g}, n, m, N)$ to $E_N$
    **else**
        Let $\mathcal{F} \leftarrow (0, ..., k-2)$
        $v \leftarrow$ PositionsToVector$(\mathcal{F})$
        **while** $\mathcal{F} \neq (m-(k-1), ..., m-1)$ **do**
            start $\leftarrow \max(F_1 - 1, 0)$
            **for** $j =$ start to $m - 1$ **do**
                **if** $e_j \neq v_j$ **then**
                    $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
                    **if** sum$(\mathbf{e}) \leq N$ **then**
                        Add $\mathbf{e}$ to $E_N$
                    **end if**
                **end if**
                **if** $j = F_1$ **then**
                    $\mathbf{e}' \leftarrow \mathbf{e}$
                **end if**
            **end for**
            $\mathcal{F} \leftarrow$ UpdateForcedPositions$(\mathcal{F}, m)$
            $\mathbf{v} \leftarrow$ PositionsToVector$(\mathcal{F})$
            $\mathbf{e} \leftarrow \mathbf{e}'$
        **end while**
    **end if**
    $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
    $k \leftarrow k - 1$
**end while**
Remove duplicate elements in $E_N$
Return $E_N$

**Algorithm 5** SingleErrorCorrectionMult($\mathbf{s}, \mathbf{g}, n, m, N$)

**Inputs:**

$\mathbf{s}$: the syndrome
$\mathbf{g}$: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector
$N$: the maximum number of bit errors considered

**Output:**

$\tilde{E}_N$: the list of valid error patterns obtained during the single-bit error correction process and containing up to $N$ errors

$\tilde{E}_N \leftarrow \{\}$
Let $\mathbf{e}$ be a vector of length $m + n$
$\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
**if** $\text{sum}(\mathbf{e}) \leq N$ **then**
    Add $\mathbf{e}$ to $\tilde{E}_N$
**end if**
**for** $j = 0$ to $m - 1$ **do**
    **if** $e_j = 1$ **then**
        $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
        **if** $\text{sum}(\mathbf{e}) \leq N$ **then**
            Add $\mathbf{e}$ to $\tilde{E}_N$
        **end if**
    **end if**
**end for**
Return $\tilde{E}_N$

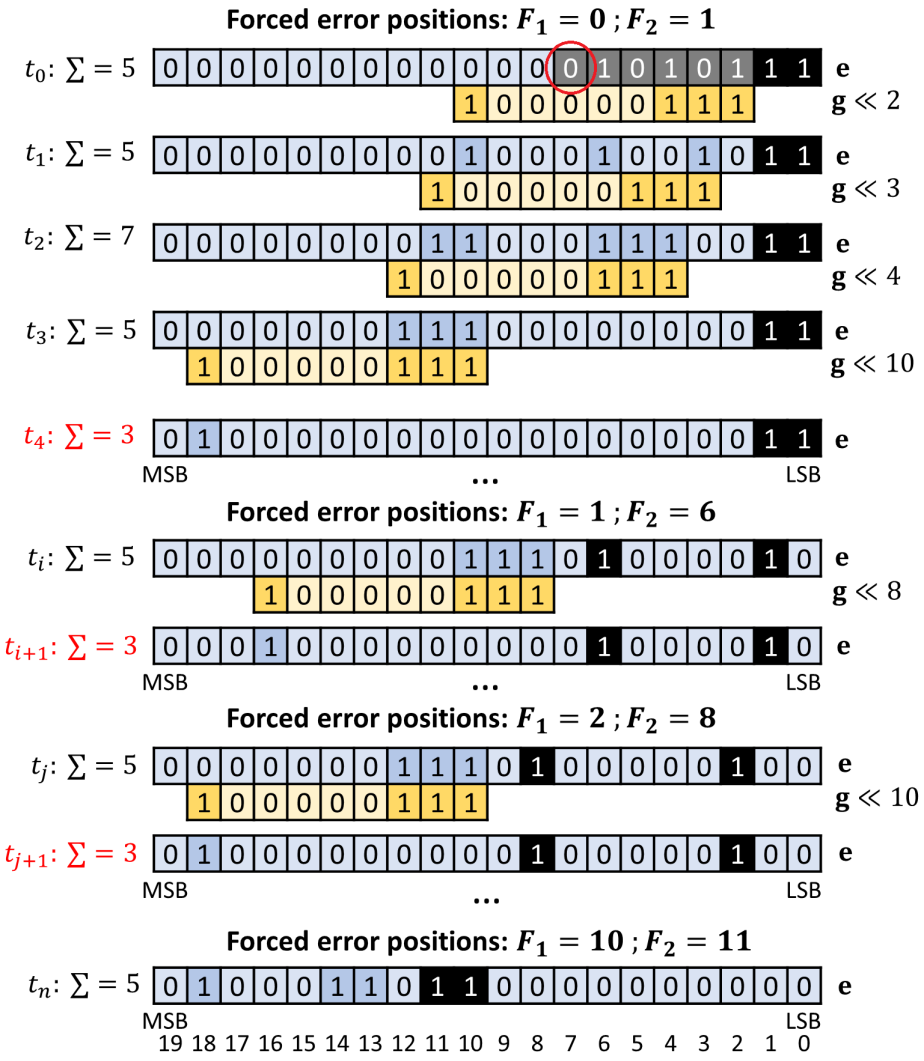


Figure 7: Illustrative example of the proposed algorithm performed over CRC-8-CCITT (yellow cells) protecting 12 data bits, where $N = 3$ and $s(x) = x^6 + x^4 + x^2 + x + 1$ (grey cells). Forced bit positions are represented as black cells in the vector $\mathbf{e}$. Three solutions are valid candidates in this example, where $\sum$, representing sum($\mathbf{e}$), equals 3 (shown in red font). Here, $E_3 = \{(0, 1, 18); (1, 6, 16); (2, 8, 18)\}$.

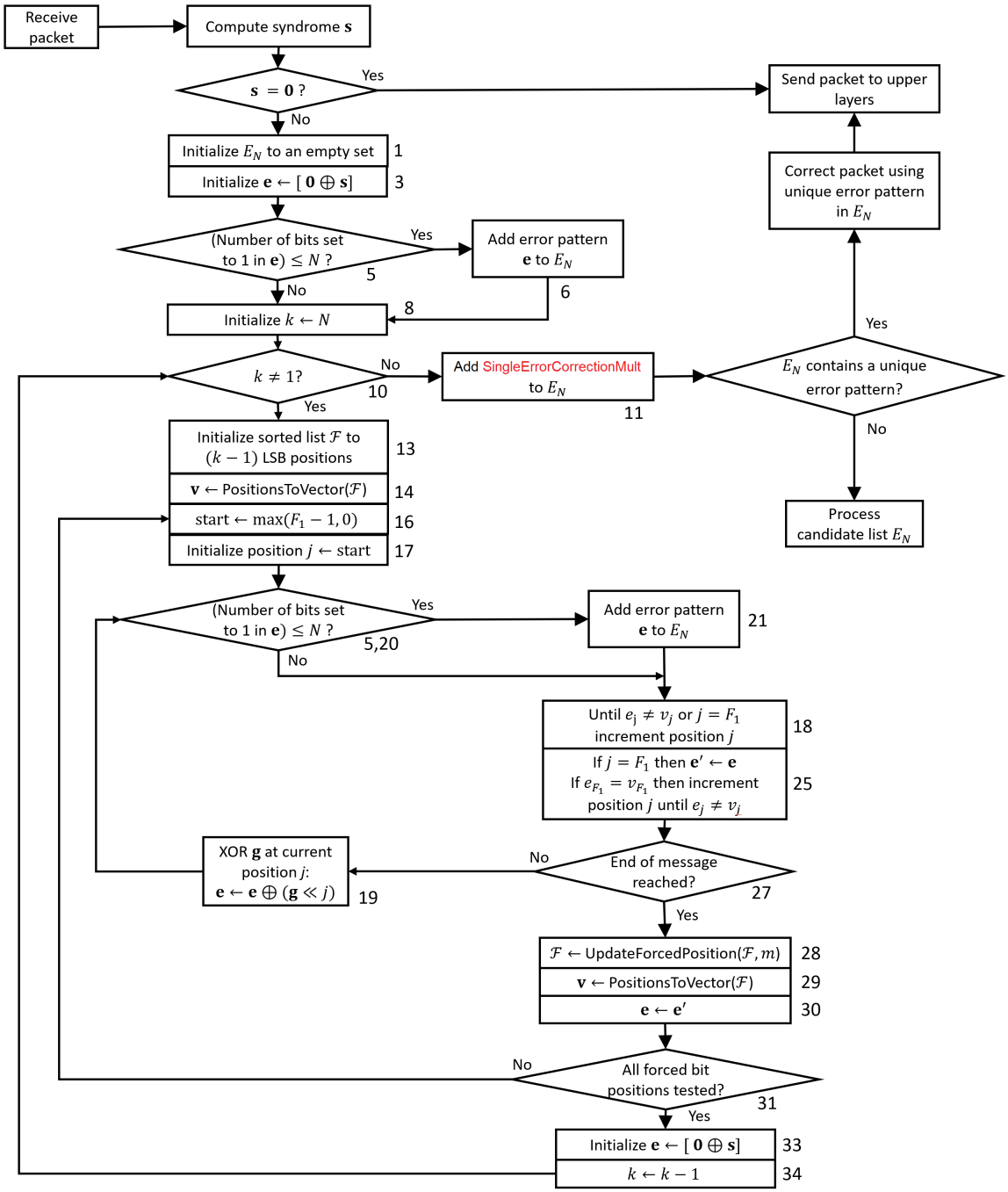


Figure 8: Flowchart of the proposed method algorithm for correcting multiple errors in the packet. Numbers show the corresponding steps in Algorithm 2.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

# Table-free multiple bit-error correction using the CRC syndrome

**VIVIEN BOUSSARD[1,2], STÉPHANE COULOMBE[1], FRANÇOIS-XAVIER COUDOUX[2] AND PATRICK CORLAY[2],**

[1]Department of Software and IT Engineering, École de technologie supérieure, Université du Québec, Montreal, QC, H3C 1K3, Canada

[2]Univ. Polytechnique Hauts-de-France, CNRS, Univ. Lille, ISEN, Centrale Lille, UMR 8520 - IEMN - Institut d'Électronique de Microélectronique et de Nanotechnologie, DOAE - Département d'Opto-Acousto-Électronique, F-59313 Valenciennes, France

Corresponding author: V. Boussard (e-mail: vivien.boussard.1@etsmtl.net).

**ABSTRACT** In this paper, we propose a novel method for correcting multiple errors in data packets, using the Cyclic Redundancy Check (CRC) syndrome present in low layers of protocol stacks. The proposed method generates the whole list of error patterns, leading to a received syndrome containing up to a given maximum number of errors. Our approach is table-free, is computationally efficient, and can instantly correct erroneous packets when the output list contains a single element. A performance study is conducted, and shows that the proposed approach outperforms existing ones in Bluetooth Low Energy (BLE) as it can correct all single- and double-error patterns as well as most triple-error cases when considering small payloads used in Internet of Things (IoT) applications.

**INDEX TERMS** Data communication, Error correction, Cyclic Redundancy Check (CRC), Internet of Things (IoT), Bluetooth Low Energy (BLE)

## I. INTRODUCTION

CYCLIC Redundancy Check (CRC) codes constitute a well-known special case of checksum functions, which are typically used for packet error detection in a wide variety of low-layer protocols [1]. Their main purpose is to validate the integrity of received packets. If an error is detected by such codes, the corrupted packet is normally discarded and a data recovery mechanism can be set, as implemented in protocols such as the Transmission Control Protocol (TCP) [2], where reliability is ensured through retransmission of the corrupted data. In order to avoid systematic retransmission, which would lead to an increased amount of data and extra delays within the network, error correction methods have been proposed at the receiver side. In addition, error detection codes such as CRCs and Checksums [3] have also been demonstrated to allow error correction [4]–[11]. The principle of CRC error detection is based on the computation of a so-called CRC field at the transmitter side. The value of this field is the remainder of the long division of the protected bit sequence, the data, which we will refer to as the *payload*, denoted $d(x)$, by a generator polynomial (a binary polynomial of degree $n$ defined by the protocol used, denoted $g(x)$). The payload is left-shifted by $n$ positions before the division. In Eq.(1), the remainder is denoted $r(x)$ and $q(x)$ represents the quotient of the long division [1]:

$$d(x).x^n = q(x).g(x) + r(x) \Rightarrow \text{CRC} = r(x) \qquad (1)$$

The computed CRC field $r(x)$ is then appended to the payload and sent to the receiver. The transmitted packet, comprising the payload and its associated remainder, is denoted $p_T(x) = d(x).x^n + r(x)$.

At the receiver, a long division by $g(x)$ is performed on the received packet, denoted $p_R(x)$, in order to check it's integrity. An error-free packet (i.e., $p_R(x) = p_T(x)$) is thus a multiple of $g(x)$ and the remainder is zero. In the contrary case, an error will modify $p_T(x)$ and produce a non-zero value as the remainder. The result is called the syndrome of the CRC, denoted $s(x)$. The standard management here consists in automatically discarding a received packet with a non-null syndrome. However, such management leads to a waste of information. In real-time applications such as video conferencing, packet retransmission is unavailable. One would then benefit from extracting as much information as possible from a received corrupted packet. Our approach is to propose algorithms to attempt to repair such corrupted data using the actual syndrome value.

CRC-based error correction techniques have been explored in previous works, and can be divided into two main categories:

1) **Estimator approaches [11]–[13]**: These approaches use statistical estimators, such as the Maximum A Posteriori (MAP) estimator, and aim at finding the most probable binary sequence that has been sent, considering the received erroneous sequence. The CRC is used to check the validity of the MAP sequence or can be part of the estimation process. Such methods can use optimization techniques such as the Alternating Direction Method of Multipliers (ADMM) [14] or Belief Propagation (BP) [15]. These approaches to MAP are costly, and generally use Log Likelihood Ratios (LLR) [13] and provide information on the confidence of the received bit, expressed as a real value between $-\infty$ and $+\infty$, based on the received soft values. Unfortunately, today's TCP/IP and User Datagram Protocol UDP/IP protocol stacks are essentially designed to deal with hard values (decoded bits), and consequently, such approaches cannot be implemented in current architectures without great effort.
2) **Lookup table approaches [4]–[8]**: These approaches implement lookup tables prior to the communication, in which each entry contains the syndrome resulting from one [6] or two [7] errors at specific positions. Upon reception, when a CRC check results in a non-null syndrome, the table is scanned. If a match is found, the corresponding bit positions are flipped to correct the packet. By definition, the CRC codes are designed such that each single error leads to a unique syndrome within the period of the generator polynomial used. The period of a generator polynomial is thus the number of different syndromes it can output for single errors. If the packet length surpasses the period of the generator polynomial, several single-error positions could lead to the same syndrome, thereby introducing ambiguity. Recently, some CRC-aided error correction methods have implemented a lookup table approach to increase their correction capacities [5]. Besides high memory requirements for storing the table, such approaches raise two main issues:
    - Lack of flexibility: lookup tables must be generated prior to the transmission, and cannot be dynamically modified to support multiple generator polynomials and larger packet sizes than those for which they were designed.
    - Memory constraints: memory requirements for lookup table-based approaches rapidly increase with the number of errors to consider. In fact, such methods must store the entire set of possible error patterns and their associated syndrome.

In this paper, we propose a novel approach to error correction that outputs the exhaustive list of CRC-compliant binary sequences containing up to $N$ errors. This method does not need any lookup table, which thus reduces the memory resources needed and allows the algorithm to be flexible as it can be used for any number of errors and any payload length without the need to rebuild a lookup table. Whereas CRC-aided Maximum Likelihood (ML) methods [4] typically use CRC to check the validity of the candidates at the end of the MAP process, our method uses the CRC syndrome itself to produce the list of candidates, thus ensuring the CRC integrity of every candidate. The output list can be used to instantly correct the packet if it contains a single element or it can be used along with error correction or validation methods from upper layers of the protocol stack in order to reduce the list of candidates.

The paper is organized as follows. In section II, we give a detailed description of the proposed method in three distinct parts. The first one describes the concept of the approach and its application to single-error correction. Challenges encountered with the double-error correction are then introduced and generalization to any number of errors is explained. In section III, we present the proposed algorithm's performance as compared to state-of-the-art approaches. Tests are conducted according to different standards used in targeted applications, such as Wi-Fi [16] and Bluetooth Low Energy [17]. Simulation results demonstrate the superiority of the proposed solution in terms of error correction rate, computational complexity and memory usage. They show that for small-sized packets such as those found in IoT, we can achieve a 100% correction rate for corrupted packets containing two errors or less, as well as high correction rates for three errors. In section IV, we conclude and give an overview of future research works.

## II. PROPOSED METHOD

The proposed method uses the CRC syndrome value $s(x)$ computed at the receiver to list all the possible error patterns that lead to such a specific syndrome, considering a maximum number of errors. The resulting list can contain one or several entries at the end of the process. Each entry represents the positions of the bits to be flipped to recover a CRC-valid packet, i.e., it reveals the error positions. When the list contains only one element, we can instantly correct the packet, but when it contains several entries, additional information is required in order to identify the actual error pattern among the candidates. The proposed approach is flexible, and lists the whole set of possible error patterns with up to $N$ errors, where the parameter $N$ can be set according to the observed channel conditions, for instance. In this section, we first introduce the basic theoretical concepts of the proposed method for the single-error case. Then, we extend the method to double-error patterns, followed by multiple-error patterns.

### A. FUNDAMENTALS

For convenience, it is common to use a binary vector representation of binary polynomials as described in [19] and illustrated in Fig. 1. Using the vector representation, the degree of a coefficient corresponds to the bit position of the associated element in the vector. The length, in bits, of a vector is equal to the degree of the polynomial increased by 1, due to the existence of degree 0 in the polynomial (i.e., a polynomial of highest degree $x^{15}$ will be represented as a 16-bit vector). Vectors allow a better understanding of operators

such as *exclusive or* (XOR) and *binary left shifts*. Throughout this paper, specific notations will be used. The following is a list of such notations based on [18] and their definitions:

- $\mathbf{a}$ : binary vector $[a_k, \ldots, a_0]$ of length $k+1$ associated with the binary polynomial $a(x)$ of degree $k$
- $a_i$ : $i^{th}$ bit (entry) of binary vector $\mathbf{a}$, starting from least significant bit (LSB)
- $m$ : payload length in bits
- $n$ : syndrome length in bits
- $M$ : total packet length in bits ($M = m + n$)
- $N$ : number of errors searched
- $P_1$ : error position obtained from the single-error correction algorithm (Algorithm 1)
- $\mathcal{F}$ : sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$
- $\text{len}(\mathcal{F})$ : number of elements in the list $\mathcal{F}$
- $E_i$ : set of valid error patterns containing $i$ errors or less
- $\text{sum}(\mathbf{a})$: number of non-zero elements in a binary vector $\mathbf{a}$; also denoted $\sum$ when the context is clear
- $\oplus$ : XOR operator between binary vectors
- $+$ : XOR operator between polynomials
- $\ll$ : left shift operator
- $\leftarrow$ : affectation operator
- $t_i$ : $i^{th}$ step

We will frequently use the following binary vectors:

- $\mathbf{0}$ : null vector (the length depends on the context)
- $\mathbf{g}$ : generator polynomial vector of length $n+1$ with $g_0 = 1$ and $g_n = 1$, given its definition [20]
- $\mathbf{s}$ : syndrome vector of length $n$
- $\mathbf{e}$ : error vector of length $M = n + m$

According to the definition of the CRC [1], we know that the syndrome $s(x)$ is computed at the receiver as the remainder of the division of the received packet by the generator polynomial, which can be expressed as:

$$s(x) = p_R(x) \bmod g(x) \tag{2}$$

When no error occurs, the syndrome $s(x)$ is equal to a null polynomial. If we consider an error pattern $e(x)$, the syndrome of the received packet can be expressed as:

$$s(x) = (p_T(x) + e(x)) \bmod g(x) \tag{3}$$

where $s(x)$ is a non-null syndrome. A given syndrome value can be the result of several different error patterns $e(x)$, containing different numbers of errors. We denote $E_M(s(x))$ the set of all valid error patterns leading to the syndrome $s(x)$. In order to lighten the notation, we will use $E_M$ since we are interested in a single syndrome value throughout the process. The error patterns in $E_M$ contain between 1 and $M$ errors (all bits of the packet are erroneous in the latter case). We denote $E_i$ the subset of $E_M$ comprising error patterns with $i$ errors or less ($1 \leq i \leq M$). We thus have:

$$E_i \in E_{i+1} \quad \forall \quad 1 \leq i \leq M-1 \tag{4}$$

where the $E_i$ are not disjoint sets. The number of elements in $E_M$ and in each subset $E_i$ depends on the syndrome, the generator polynomial used and the packet length. We aim at finding the actual error pattern $e_A(x)$ among the set of all error patterns leading to the computed syndrome value $s(x)$. Given the definition of the modulo operator and Eq.(3), all error patterns of $E_M$ are defined as:

$$E_M = \{e(x) \in \text{GF}(2^M) \mid e(x) = s(x) + q(x).g(x) \quad \text{with} \quad q(x) \in \text{GF}(2^m)\} \tag{5}$$

where $m$ is the payload length and $\text{GF}(2^m)$ is the Galois Field of order $2^m$ (i.e., the set of binary polynomials of length $m$ [19]). In other words, the error pattern corresponding to the syndrome can be any binary polynomial of highest degree $m-1$ (that we denoted as $q(x)$) multiplied by the generator polynomial, with $s(x)$ added. The set $E_M$ is called the equivalence class containing $s(x)$. Each element is equivalent under mod $g(x)$ operation since adding any multiple of $g(x)$ to $s(x)$ does not affect the result. Every possible value of $q(x)$ in this equation will produce a CRC-compliant error pattern $e(x)$ (i.e., an element of $E_M$). The degrees of the non-zero coefficients in the resulting $e(x)$ correspond to the erroneous positions in the corrupted packet. Assuming that packets are not too damaged, the straightforward approach to identifying candidates having a maximum number of errors would be to test every possible value of $q(x)$ and to count the number of non-zero coefficients in the resulting error polynomial $e(x)$. If this number, denoted $\text{sum}(\mathbf{e})$, is greater than a fixed threshold, the candidate is discarded. Otherwise, it is appended to the list of valid candidates. This method is computationally complex, and would require $2^m$ tests to consider all the possible values of $q(x)$. Such a complex process is therefore prohibitive to conduct in real-time scenarios, such as videoconferencing, for instance.

It can be verified that most of the possible values of $q(x)$ produce error polynomials $e(x)$ containing many errors (i.e., corresponding to highly corrupted packets cases, where $e(x)$ and its associated vector $\mathbf{e}$ contain a significant amount of non-null values). Considering the whole set of possibilities would only increase the complexity of the method. We make the hypothesis that highly corrupted packets are too damaged to be recovered. Thus, in the rest of this paper, we focus on recovering slightly corrupted packets that are worth extracting information from. Some indicators such as the Receiver Signal Strength Indicator (RSSI), included in the 802.11 standard [16], can be used to indicate the degree of corruption of a received packet. In the remainder of this

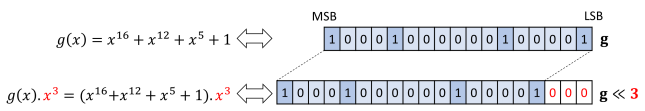


**FIGURE 1.** Illustration of the binary vector representation: each polynomial $g(x)$ with binary coefficients can be seen as a binary vector $\mathbf{g}$. Multiplying $g(x)$ by $x^n$ corresponds to a left shift of $\mathbf{g}$ by $n$ positions.

section, we first describe the single-error correction method (the search for all elements in $E_1$), and then we introduce the double-error correction and extend it to any number $N$ of errors (i.e., we determine the elements of $E_N$).

### B. SINGLE-ERROR CORRECTION

We exploit the knowledge on both the generator polynomial and the way the syndrome is computed to reversely find the position of the single error at the receiver side. With such an approach, we are not testing possible values of $q(x)$, but rather, are gradually building a specific polynomial $q(x)$, one coefficient at a time. If a single candidate is identified at the end of the process, the packet can be corrected. If not, some additional processes must be used to determine the only candidate to consider.

We now demonstrate that the proposed approach is guaranteed to identify single errors. Suppose that the error is at position $P_1$ (i.e., $e(x) = x^{P_1}$). We know from the definition of Eq.(5) that:

$$x^{P_1} = s(x) + q(x).g(x) \text{ for a } q(x) \in \text{GF}(2^m) \qquad (6)$$

It is clear that $q(x)$ must be constructed such that $s(x) + q(x).g(x)$ has zero coefficients for all positions $i \neq P_1$. Having coefficients at positions $i < P_1$ is ensured by successively determining, from LSB to MSB, the coefficient values of $q(x)$ meeting this condition. For simplicity, in the following derivations, we can consider $s(x)$ of degree $m-1$ with $s_i = 0, i > n-1$. We have:

$$\begin{aligned}
x^{P_1} &= \sum_{i=0}^{m-1} s_i x^i + \left(\sum_{i=0}^{m-1} q_i x^i\right)\left(\sum_{j=0}^{n} g_j x^j\right) \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} q_i . \sum_{j=0}^{n} g_j x^{i+j} \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} q_i . \sum_{r=i}^{i+n} g_{r-i} x^r \\
&= \sum_{i=0}^{m-1} s_i x^i + \sum_{i=0}^{m-1} \left(q_i . g_0 x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right) \\
&= \sum_{i=0}^{m-1} \left((s_i + q_i . g_0) x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right) \\
&= \sum_{i=0}^{m-1} \left((s_i + q_i) x^i + q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r\right), \text{since } g_0 = 1
\end{aligned} \qquad (7)$$

From Eq.(7), it is clear that for every value of $i$ in the main summation, $(s_i + q_i.g_0)x^i$ is of a lower degree than $q_i . \sum_{r=i+1}^{i+n} g_{r-i} x^r$. Thus, for $i = 0$ and each successive value of $i$, we can easily determine the $q_i$ value resulting in the desired result, namely, zero coefficients for positions $i < P_1$. Of course, setting $q_i$ to 1 creates terms that must be considered in subsequent positions. If $s(x)$ was generated by a single error, performing the process on increasing values of $i$ would eventually lead to a monomial (i.e., $x^{P_1}$ for a certain value of $P_1$). This must happen, otherwise, after position $i = P_1$, adding $\sum_{r=i+1}^{i+n} g_{r-i} x^r$ (i.e., $q_i \neq 0$) would add a coefficient at position $i+n$ (the MSB) that cannot be canceled without adding a coefficient of even higher degree.

**Algorithm 1** SingleErrorCorrection($\mathbf{s}$,$\mathbf{g}$,$n$,$m$)

**Inputs:**

$\mathbf{s}$: the syndrome vector
$\mathbf{g}$: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector

**Output:**

$E_1$ the list of valid error patterns for a single bit error

1: $E_1 \leftarrow \{\}$
2: Let $\mathbf{e}$ be a vector of length $m+n$
3: $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
4: **if** $\text{sum}(\mathbf{e}) = 1$ **then**
5: Add $\mathbf{e}$ to $E_1$
6: **end if**
7: **for** $j = 0$ to $m-1$ **do**
8: **if** $e_j = 1$ **then**
9: $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
10: **if** $\text{sum}(\mathbf{e}) = 1$ **then**
11: Add $\mathbf{e}$ to $E_1$
12: **end if**
13: **end if**
14: **end for**
15: Return $E_1$

The search for single-error patterns is illustrated in Algorithm 1 and the corresponding flowchart is given in Fig. 2. Each step of the algorithm is identified in Fig. 2 using binary notations. We provide further details in the following steps:

**3**: We first initialize the error $\mathbf{e}$ to a zero vector of length $M = m + n$ and replace the $n$ LSB values with the computed syndrome $\mathbf{s}$, as shown in Fig. 3. We can note that it corresponds in Eq.(5) to $e(x) = q(x).g(x) + s(x)$, where $q(x)$ is equal to zero. Such initialization allows to comply with Eq.(5) and maintains its equivalence relation as we are adding shifted versions of $g(x)$ to build $q(x)$ in step 9.

**4-5**: At this point, we compute the sum of non-zero elements in $\mathbf{e} = \mathbf{s}$, denoted $\text{sum}(\mathbf{e})$, equivalent to the number of errors in the computed syndrome. If it contains only one element to 1, then it is itself a suitable candidate as a single-error pattern.

**7**: We scan the $m$ first payload positions from 0 to $m-1$. We do not consider the last $n$ positions since they correspond to the range of the XOR operation to perform. Hence, it reaches the end of the payload at position $m-1$ and beyond this position would be out of the payload range.

**8-9**: For each scanned position, we check the $j^{th}$ bit value of the current error vector. If this value is 0, we simply jump to

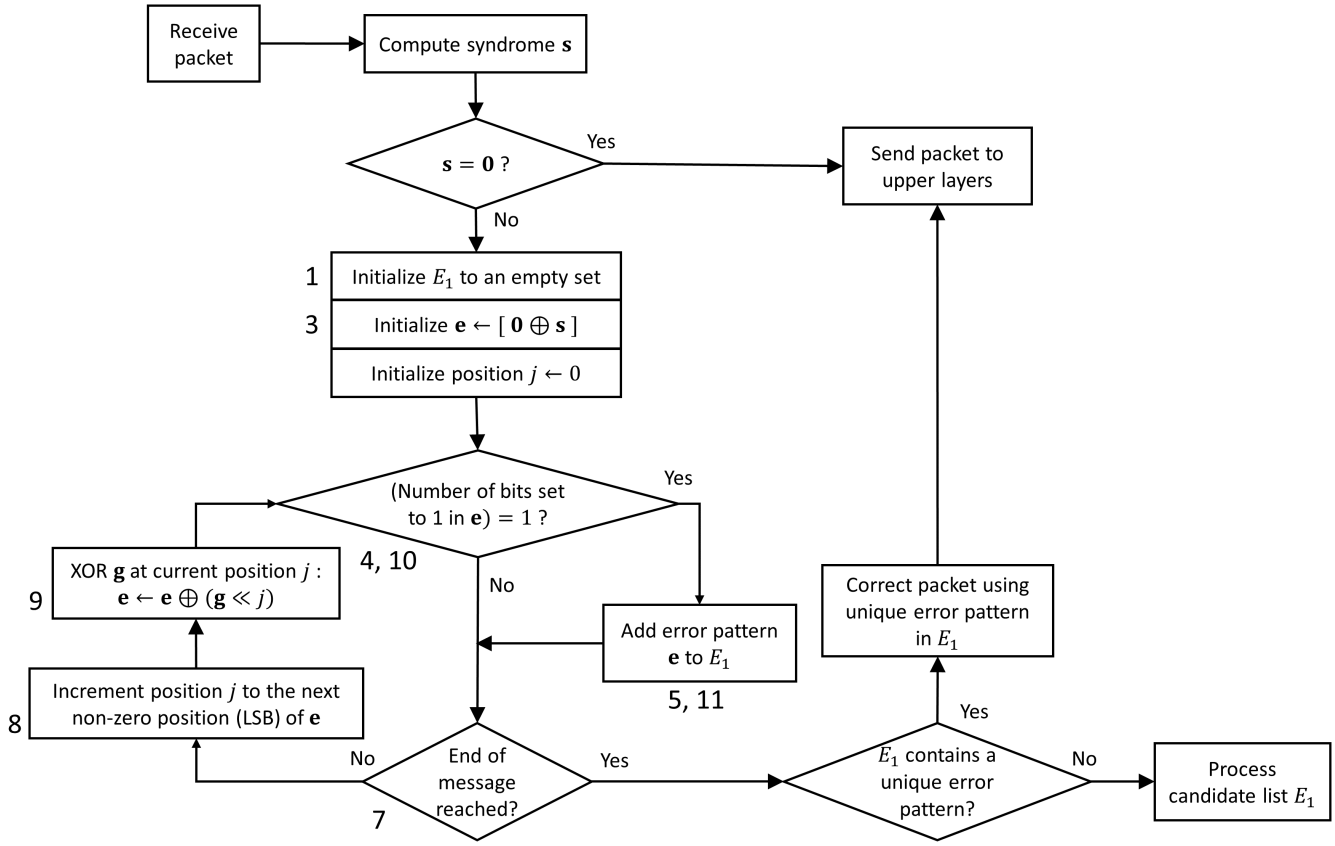


**FIGURE 2.** Flowchart of the proposed method's algorithm to correct a single error in the packet. Numbers show the corresponding steps in Algorithm 1.

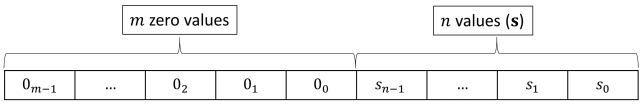


**FIGURE 3.** Structure of the initial error vector $\mathbf{e} = \mathbf{0} \oplus \mathbf{s}$

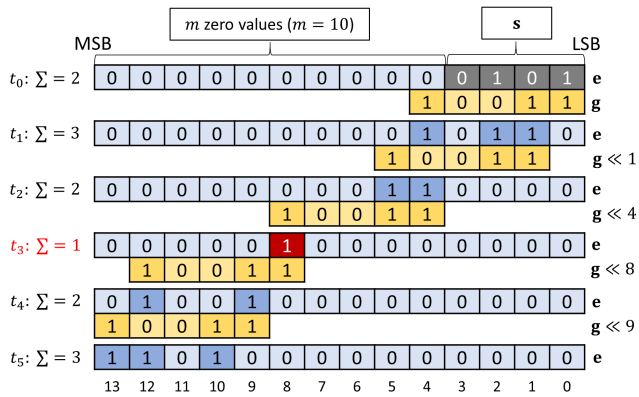


**FIGURE 4.** Illustration of the single-error search applied to CRC-4-ITU, where $g(x) = x^4 + x + 1$ (yellow cells) with a syndrome $s(x) = x^2 + 1$ (grey cells). In this notation, $\sum$ represents $\mathrm{sum}(\mathbf{e})$. A single-error pattern is found at bit position 8 at step $t_3$. In this example: $E_1 = \{8\}$ (i.e., the red cell).

the next element. If it is 1, we cancel the non-zero value by performing an XOR operation with $\mathbf{g}$ at this position, as its LSB is 1 (i.e., $g_0 = 1$). Note that for clarity, we simplified this step in the figures and flowcharts by directly incrementing the current position to the next element set to 1. Each time we perform an XOR operation at position $j$, a 1 is added at MSB position $j + n$ since $g_n = 1$. If the error pattern is a single error at position $k$, the proposed method will reveal this since the XOR operations will be able to cancel all bits at positions $j < k$, and all bits at positions $j > k$ are already set to zero. The strategy is to cancel every LSB non-zero element until the end of the packet is reached, and thus not miss any single-error candidate.

**10-11**: After each cancelation, we check the number of non-zero coefficients in the error vector $\mathbf{e}$. If this number is equal to 1, a valid single-error candidate is identified and its position is appended to the list.

At the end of the whole process, if the algorithm does not provide any candidate, it means the syndrome was caused by multiple errors in the packet.

So, depending on the packet size and syndrome, there can be zero, one or multiple candidates. This latter case occurs in long enough packets due to the periodic aspect of generator polynomials, as discussed in the introduction. The whole single-error search process is illustrated in Fig. 4. In the

present case, the payload consists of 10 data bits and the CRC-4-ITU where a generator polynomial $g(x) = x^4+x+1$ is applied. At the receiver, the computed syndrome is $s(x) = x^2 + 1$, represented in dark grey boxes at step $t_0$.

At step $t_0$, the error vector $\mathbf{e}$ is initialized to $m$ zeros, $m$ being the length of the protected data, and the syndrome $\mathbf{s}$ is appended. We first check the number of non-zero values in the error vector to verify if the syndrome itself is a valid candidate (i.e., if the syndrome is corrupted). Since the sum of non-zero values in $\mathbf{s}$ is greater than 1, the syndrome does not contain a valid single-error pattern, and is thus not a candidate. At each step until we reach the end of the packet, we successively perform an XOR operation with $\mathbf{g}$ at each non-null position and check the resulting number of 1s in the updated error vector $\mathbf{e}$. If this sum is equal to 1, the candidate is appended to the list. Such a candidate is found at time $t_3$, since there is only one bit set in the error vector. A first single-error candidate is thus identified, containing an error at position 8. Since there could be several candidates, we continue the scanning of the packet until the end. At step $t_5$, the algorithm reaches the end of the packet and the list of candidates contains a single entry. Flipping the bit at position 8 in the corrupted packet is the only valid correction if a single error has occurred.

### C. DOUBLE-ERROR CORRECTION

#### 1) Problem with straightforward extension of Algorithm 1

The method described in the previous section produces as output the exhaustive list of single-error patterns corresponding to a non-null syndrome at the receiver, given the generator polynomial used and the length of the protected data. To deal with double-error patterns, a straightforward method would be to run the exact same algorithm while appending all the error vectors $\mathbf{e}$ with two coefficients set to 1 to the candidate list. This approach would be able to output double-error patterns, but cannot ensure that an exhaustive list of such error patterns is provided. Actually, only one specific type of double-error patterns will be output, namely, those in which the double-error pattern covers $n$ bits or less (i.e., are close to each other). The single-error search aims at canceling non-zero values from LSB to MSB. This cancelation is performed thanks to an XOR operation between a shifted version of the generator polynomial and the constantly updated error vector. We can observe that there cannot be more than $n$ bits between the 1 located at the LSB position and the 1 at the MSB position, as illustrated in Fig. 5, which represents the error vector during the single-error search. The MSB zeros correspond to the positions still in the original state of $\mathbf{e}$, initialized as a null vector, and the LSB zeros correspond to the already canceled positions. Between these two null subvectors we have the possible non-zero positions, with a maximum width of $n$ bits. We will refer to the maximum distance between the first and last non-zero coefficients as the error range of the method. At step $t_9$ of Algorithm 1, the update of the error vector $\mathbf{e}$ can be expressed as:

$$\begin{aligned}\sum_{i=0}^{m+n-1} e_i x^i \leftarrow & \sum_{i=0}^{m+n-1} e_i x^i + \sum_{i=j}^{j+n} g_{(i-j)} x^i \\ &= \sum_{i=j+1}^{j+n} (e_i + g_{(i-j)}).x^i \\ &= x^{(j+n)} + \sum_{i=j+1}^{j+n-1} (e_i + g_{(i-j)}).x^i\end{aligned} \tag{8}$$

From Eq.(8), it is clear that the whole set of non-null values in the error vector covers $n$ positions at most. In fact, all the values in $\mathbf{e}$ up to position $j$ are already canceled and set to 0, due to the design of the proposed algorithm, and all positions above $j + n$ are also set to 0 due to the initialization of the error vector (i.e., $\mathbf{e} = \mathbf{0} \oplus \mathbf{s}$). As the highest degree term of the generator polynomial is 1, we can see that position $x^{j+n}$ is set. The other non-null positions are subject to the values of the error vector at its current state and the other terms of the generator polynomial. The range of non-null values is denoted the error range, and is illustrated in Fig. 5. In conclusion, a straightforward extension of Algorithm 1 would only yield error patterns in which errors are within a range of $n$ bits. A different approach is thus required.

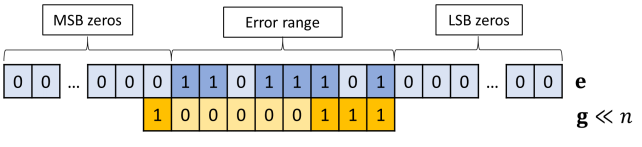


**FIGURE 5.** Illustration of the error range applied to CRC-CCITT-8 ($n = 8$). Canceling LSB non-zero values by performing an XOR operation with a generator polynomial of width $n + 1$ bits produces an error range of $n$ bits.

#### 2) Proposed double-error correction approach

To obtain the exhaustive list of error patterns, we aim at expanding the error range to have it cover the entire length of the protected data. The method we propose is to force a bit to 1 during the process. Forcing a position consists in setting it (or leaving it) to 1 during the single-error search. In other words, it is equivalent to making the hypothesis that a specific position is actually erroneous in the packet. Hence, we force one bit to 1 at position $F_1$ during the process and run the single-error algorithm on the remaining length of the packet. If the bit is already 1, we leave it untouched. Otherwise, setting a bit to 1 is done by applying an XOR operation with $g(x)$ at position $F_1$ in order to maintain the equivalence relation. Throughout the cancelation process with the forced bit set, if a single-error position (denoted hereafter $P_1$) is obtained from the single-error correction algorithm, we determine a double-error pattern with errors at positions $F_1$ and $P_1$.

As we want to get the whole list and we do not know the actual position of the first error, we test each possible forced position in order to output all the double-error patterns associated with the computed syndrome. In the proposed

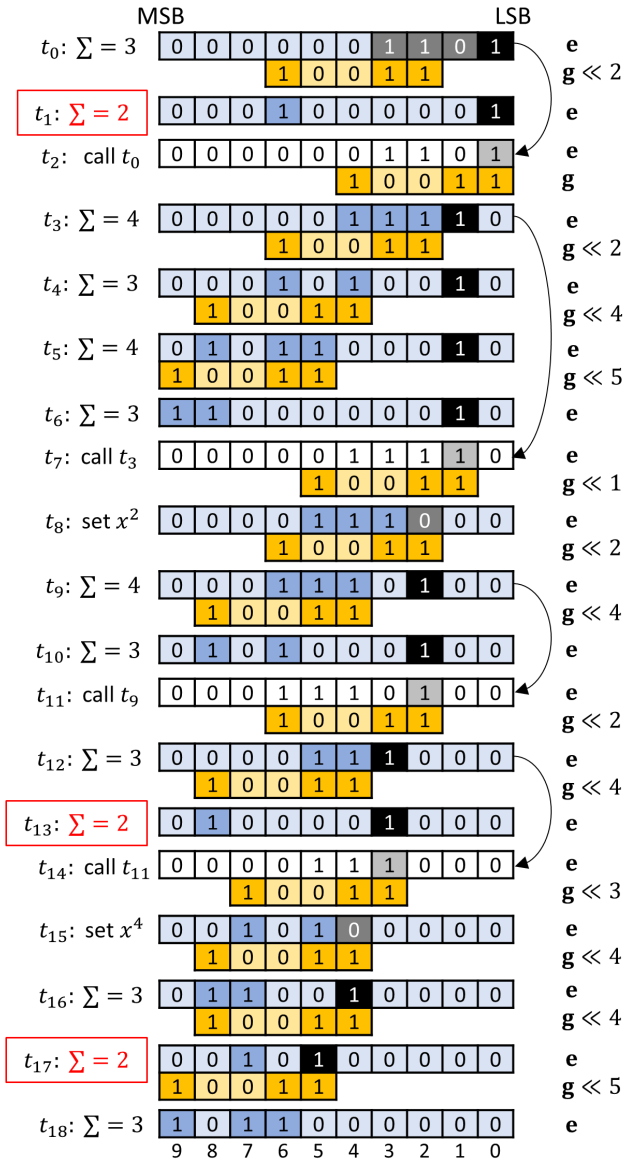


**FIGURE 6.** Visual example of the proposed algorithm applied to double-error correction, performed over CRC-4-ITU (yellow cells) protecting 6 data bits with a syndrome $s(x) = x^3 + x^2 + 1$ (grey cells). Each forced bit position, represented as a black cell, is tested throughout the process to get the exhaustive list of double-error patterns. In this example there are three such cases at steps $t_1$, $t_{13}$ and $t_{17}$, thus $E_2 = \{(0, 6); (3, 8); (5, 7)\}$.

algorithm, we suggest forcing positions starting from LSB to MSB. Moreover, starting from LSB at each tested forced position would lead to a cancelation of the same first positions several over and degrade the computational efficiency. To avoid verifying the same possibilities repeatedly, we store the value of $\mathbf{e}$ when a bit is forced and recall this state to start from it and save computations for the next forced position to test.

Fig. 6 illustrates the complete process for listing the double-error patterns corresponding to the syndrome $s(x) = x^3 + x^2 + 1$ applied to a CRC-4-ITU of generator polynomial $g(x) = x^4 + x + 1$ protecting 6 data bits. In this figure, forced positions are represented as black boxes through time. At step $t_0$, the error vector $\mathbf{e}$ is initialized as a null vector, to which syndrome vector $s(x) = x^3 + x^2 + 1$ is appended. As we start at position 0, and $\mathbf{e}_0$ is already set to 1, we simply jump to the next element and start the single-error search from the next non-zero position, checking at each step if the number of non-zero coefficients in $\mathbf{e}$ equals 2. A first candidate appears at $t_1$, corresponding to errors at positions $(F_1 = 0, P_1 = 6)$. Canceling the next non-zero value would move the operation out of the range of the packet, thus ending the search for a single error for this forced position. At step $t_2$ we recall the initial state of the error vector from $t_0$. The next forced bit to test is at position 1. Hence, we cancel position 0 and let position 1 be set to 1. From $t_3$ to $t_6$, we perform the single-error algorithm on the remaining length. No new error pattern is found. At time $t_7$, we recall the previous state from $t_3$ and cancel the former forced bit at position 1. At $t_8$, the next forced position to test, position 2, is not yet set to 1. We thus have to perform an XOR operation with $\mathbf{g}$ at this position to set it. We identify such cases as dark grey boxes in Fig. 6, at steps $t_8$ and $t_{15}$. We continue this process until reaching the last forced bit position, corresponding to the $m^{th}$ position starting from LSB. We can see at step $t_{18}$ that we cannot perform any XOR operation without going out of the range of the packet. Hence, the algorithm is stopped at $t_{18}$ and outputs the list of error patterns containing two errors corresponding to the received syndrome. The sums of errors in such cases are shown in red font in Fig. 6. The output list contains the following error patterns: $(F_1 = 0, P_1 = 6)$ , $(F_1 = 3, P_1 = 8)$ and $(F_1 = 5, P_1 = 7)$. The proposed approach for double-error correction is exemplified in Fig. 6 and presented in Algorithm 2 using $N = 2$.

### D. $N$-ERROR CORRECTION

We can further extend the proposed method to deal with any number $N$ of errors in a packet. The strategy applied is the extension of the double-error correction approach described in the previous section.

Much as we forced one position and scanned the remaining length of the packet using the single-error search, we can manage the $N$-error search. In such cases, we set $(N - 1)$ forced bits in the error vector, corresponding to the first $(N - 1)$ errors in the packet, and scan the remaining length using the single-error search to identify the position of the last error in the packet, if it exists. The $(N - 1)$ forced binary errors have to be tested in the packet. The proposed method to generate the list of potential error patterns containing up to $N$ errors is illustrated in Algorithm 2.

We now present the key steps of the proposed algorithm , while the corresponding flowchart is given in Fig. 8:

**3:** The binary vector of length $M$ representing the error vector $\mathbf{e}$ is initialized to $m$ zeros, followed by $n$ values, corresponding to the computed syndrome $\mathbf{s}$.

**5-6:** We first check if the number of non-zero values in this initial vector $\mathbf{e}$ is less than or equal to the targeted number of

**Algorithm 2** $N$-ErrorPatternsGeneration($\mathbf{s}$,$\mathbf{g}$,$n$,$m$,$N$)

**Inputs:**

$\mathbf{s}$: the syndrome
$\mathbf{g}$: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector
$N$: the maximum number of bit errors considered

**Output:**

$E_N$ the list of valid error patterns up to $N$ bit errors

$E_N \leftarrow \{\}$
Let $\mathbf{e}$ be a vector of length $m+n$
$\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
Let $\mathbf{v}$ be a vector of length $m$
**if** $\mathrm{sum}(\mathbf{e}) \leq N$ **then**
  Add $\mathbf{e}$ to $E_N$
**end if**
$k \leftarrow N$
**while** $k \geq 1$ **do**
  **if** $k = 1$ **then**
    Add SingleErrorCorrectionMult($\mathbf{s}$,$\mathbf{g}$,$n$,$m$,$N$) to $E_N$
  **else**
    Let $\mathcal{F} \leftarrow (0, \ldots, k-2)$
    $\mathbf{v} \leftarrow$ PositionsToVector($\mathcal{F}$)
    **while** $\mathcal{F} \neq (m-(k-1), \ldots, m-1)$ **do**
      start $\leftarrow \max(F_1 - 1, 0)$
      **for** $j =$ start to $m-1$ **do**
        **if** $e_j \neq v_j$ **then**
          $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
          **if** $\mathrm{sum}(\mathbf{e}) \leq N$ **then**
            Add $\mathbf{e}$ to $E_N$
          **end if**
        **end if**
        **if** $j = F_1$ **then**
          $\mathbf{e}' \leftarrow \mathbf{e}$
        **end if**
      **end for**
      $\mathcal{F} \leftarrow$ UpdateForcedPositions($\mathcal{F}$,$m$)
      $\mathbf{v} \leftarrow$ PositionsToVector($\mathcal{F}$)
      $\mathbf{e} \leftarrow \mathbf{e}'$
    **end while**
  **end if**
  $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
  $k \leftarrow k-1$
**end while**
Remove duplicate elements in $E_N$
Return $E_N$

**Algorithm 3** UpdateForcedPositions($\mathcal{F}$,$m$)

**Inputs:**

$\mathcal{F}$: sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$
$m$: the length of the payload vector

Note that $k = \mathrm{len}(\mathcal{F}) + 1$, with $\mathrm{len}(\mathcal{F})$ being the number of elements in the list $\mathcal{F}$

**Output:**

$\mathcal{F}'$: the updated sorted list of forced positions

**if** $F_{k-1} < (m-1)$ **then**
  $F_{k-1} \leftarrow F_{k-1} + 1$
  Return $\mathcal{F}' \leftarrow (F_1, \ldots, F_{k-1})$
**else**
  **for** $i = k-2$ to $1$ **do**
    **if** $F_i < F_{i+1} - 1$ **then**
      $F_i \leftarrow F_i + 1$
      $j \leftarrow i$
      **while** $j < k-1$ **do**
        $F_{j+1} \leftarrow F_j + 1$
        $j \leftarrow j + 1$
      **end while**
      Return $\mathcal{F}' \leftarrow (F_1, \ldots, F_{k-1})$
    **end if**
  **end for**
**end if**

**Algorithm 4** PositionsToVector($\mathcal{F}$)

**Inputs:**

$\mathcal{F}$: sorted list $(F_1, \ldots, F_{k-1})$ of $(k-1)$ bit positions forced to 1, such that $F_i < F_{i+1}, \forall i$.

Note that $k = \mathrm{len}(\mathcal{F}) + 1$, with $\mathrm{len}(\mathcal{F})$ being the number of elements in the list $\mathcal{F}$

**Output:**

$\mathbf{v}$: the corresponding vector of forced positions

$\mathbf{v} \leftarrow \mathbf{0}$
**for** $i = 1$ to $k-1$ **do**
  $v_{F_i} \leftarrow 1$
**end for**
Return $\mathbf{v}$

**Algorithm 5** SingleErrorCorrectionMult($\mathbf{s}$,$\mathbf{g}$,$n$,$m$,$N$)

**Inputs:**

$\mathbf{s}$: the syndrome vector
$\mathbf{g}$: the vector associated with the generator polynomial used to compute the CRC
$n$: the length of the syndrome vector
$m$: the length of the payload vector
$N$: the maximum number of bit errors considered

**Output:**

$\tilde{E}_N$ the list of valid error patterns during the single bit error correction process accepting up to $N$ errors

1: $\tilde{E}_N \leftarrow \{\}$
2: Let $\mathbf{e}$ be a vector of length $m+n$
3: $\mathbf{e} \leftarrow \mathbf{0} \oplus \mathbf{s}$
4: **if** $\text{sum}(\mathbf{e}) \leq N$ **then**
5: Add $\mathbf{e}$ to $\tilde{E}_N$
6: **end if**
7: **for** $j = 0$ to $m-1$ **do**
8: **if** $e_j = 1$ **then**
9: $\mathbf{e} \leftarrow \mathbf{e} \oplus (\mathbf{g} \ll j)$
10: **if** $\text{sum}(\mathbf{e}) \leq N$ **then**
11: Add $\mathbf{e}$ to $\tilde{E}_N$
12: **end if**
13: **end if**
14: **end for**
15: Return $\tilde{E}_N$

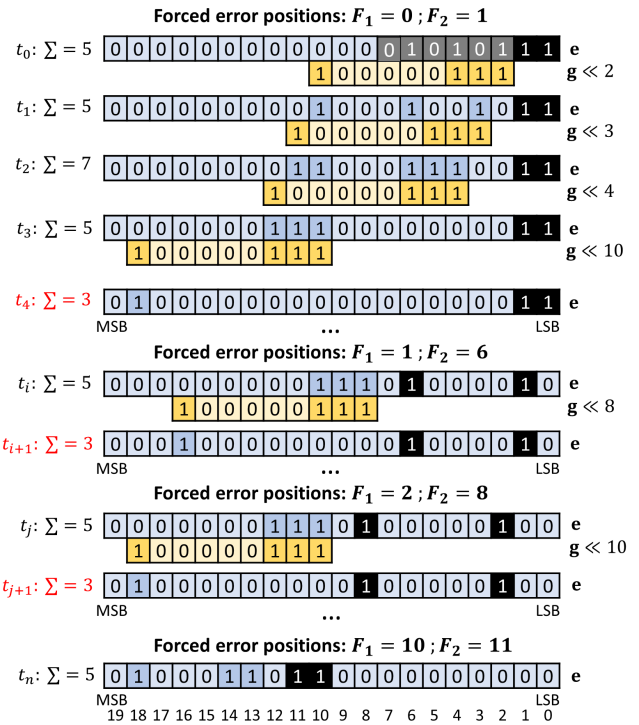


**FIGURE 7.** Illustrative example of the proposed algorithm performed over CRC-8-CCITT (yellow cells) protecting 12 data bits, where $N = 3$ and $s(x) = x^6 + x^4 + x^2 + x + 1$ (grey cells). Forced bit positions are represented as black cells in the vector $\mathbf{e}$. Three solutions are valid candidates in this example, where $\sum$, representing $\text{sum}(\mathbf{e})$, equals 3 (shown in red font). Here, $E_3 = \{(0, 1, 18); (1, 6, 16); (2, 8, 18)\}$.

errors $N$. If so, a first candidate is added to the list $E_N$.

**8-9:** The local variable $k$ represents the current number of errors considered. $k$ is initialized to $N$, then decreased at each main loop of the algorithm to consider every number of errors from $N$ to 1.

**10-11:** In the last loop, the variable $k$ equals 1. In this case, no forced position must be set and a modified version of the single-error correction algorithm, which accepts solutions where $\mathbf{e}$ contains $N$ errors or less, is performed (see Algorithm 5). The solutions are appended to the global candidate list $E_N$.

**13:** The sorted list of forced positions $\mathcal{F}$ is initialized to the $(k-1)$ LSB values at the first iteration. At this step, the set of forced positions $\mathcal{F} = (F_1 = 0, F_2 = 1, \ldots, F_{k-1} = (k-2))$. The $(k-1)$ forced positions in the set $\mathcal{F}$ are ordered such that $F_1 < F_2 < \ldots < F_{k-1}$.

**14:** The binary vector is set according to the forced positions in $\mathcal{F}$. In Algorithm 4, the bits in $\mathbf{v}$ corresponding to forced positions in $\mathcal{F}$ are set to 1. The other bits in $\mathbf{v}$ are set to 0.

**15:** The forced positions will then be updated to cover the entire set of possible fixed error positions (until the forced positions are the $(k-1)$ MSB positions). For a packet of $M$ bits, there are $\binom{M-n}{k-1}$ such positions, thanks to the range of the XOR operation performed. After setting these forced positions, we are aiming at finding the last error by conducting the single-error algorithm on the remaining part of the packet.

**16:** In order to save computations, we use as a starting point the previously obtained vector $\mathbf{e}$ after cancelation of its LSB positions up to $F_1$, the LSB position we forced. With this approach we will not have to cancel the same first positions at each iteration as we increase $F_1$.

**17:** We perform a scan on the remaining length of the error vector $\mathbf{e}$, from LSB to MSB (i.e., single-error search).

**18-19:** At each position $j$, we compare the values of $e_j$ and $v_j$ to determine if the $j^{th}$ position corresponds to a forced position. If $v_j$ and $e_j$ are both set to 0 or 1, it means that position $j$ either must not be forced (to 1) and is already set to 0, or must be forced, but is already set to 1. In both cases, the algorithm simply jumps to the next element since what is required is already in place. However, when these two elements are set to different values, it corresponds to the cases where the position $j$ has to be canceled and set to 1, or where the position $j$ has to be forced to 1, but is set to 0. In both cases, an XOR operation with $\mathbf{g}$ must be performed to maintain the equivalence relation and obtain what is required.

**20-22:** At each stage, the number of non-zero coefficients in the newly accumulated $\mathbf{e}$ is observed, similarly to steps 5-6. Whenever $\mathbf{e}$ contains $N$ errors or less, a candidate is added to the list $E_N$.

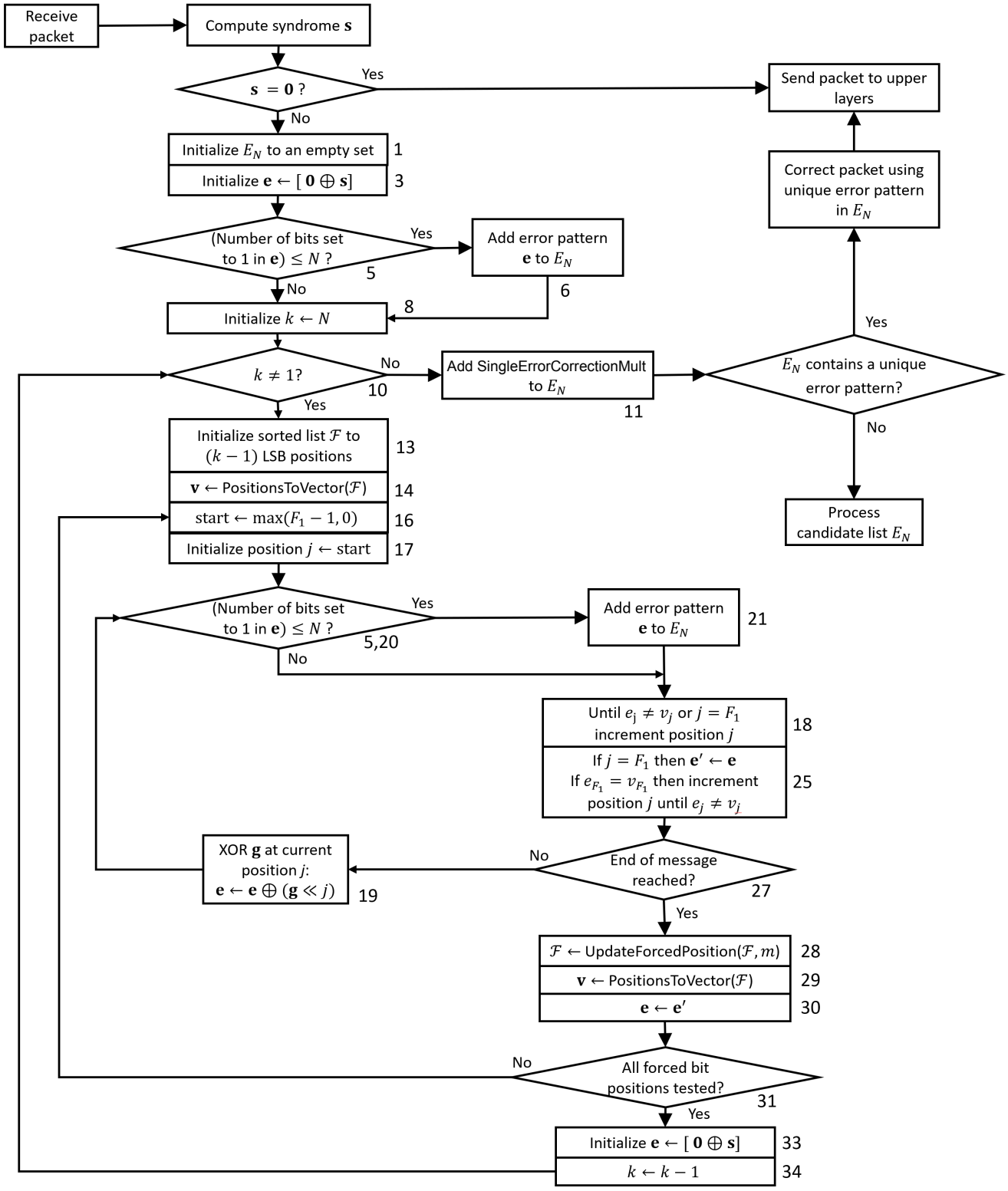


**FIGURE 8.** Flowchart of the proposed method algorithm for correcting multiple errors in the packet. Numbers show the corresponding steps in Algorithm 2.

**25:** We store the state of $\mathbf{e}$ in a vector $\mathbf{e}'$ to avoid re-canceling the same first LSBs at the next iteration.

**28:** At the end of each scan, the vector of forced positions is updated using the UpdateForcedPosition function illustrated

in Algorithm 3. In this algorithm, the $(k-1)$ forced positions are successively updated to cover the entire message. At step 1, we check if the MSB forced position has reached its final position. If not, we increase its value by one. If it has reached its final position, we successively check forced positions from MSB to LSB at step 5. When a forced position can be increased, we reversely update the other positions, from LSB to MSB.

**29:** Each time the set of forced positions is updated, the binary vector $\mathbf{v}$ is modified.

**30:** We recall the state $\mathbf{e}'$ to start from it at the next iteration.

**33:** We recall the initial state (syndrome) when we update the number of errors to consider.

Fig. 7 shows a visual example of the algorithm applied to a CRC-8-CCITT, which has a generator polynomial $g(x) = x^8 + x^2 + x + 1$. This example illustrates a triple-error management, with $\mathbf{v}$ having two non-zero values, represented as black boxes in Fig. 7. Four stages are represented here: the initial stage, where the forced error positions are $(F_1 = 0; F_2 = 1)$, two different stages that produce a valid candidate, namely $(F_1 = 1; F_2 = 6)$ and $(F_1 = 2; F_2 = 8)$, and the final step, with the last two forced positions $(F_1 = 10; F_2 = 11)$. By definition, these positions must remain set to 1 at the end of the scan. The other non-null values in $\mathbf{e}$ must be canceled from LSB to MSB, using the single-error search method. At each step, we successively add left-shifted versions of $\mathbf{g}$ at these positions and if the sum of non-null values in $\mathbf{e}$ is equal to 3 ($N = 3$), then $\mathbf{e}$ is considered as a valid candidate. We can observe that this example produces three valid error patterns with error positions $(F_1 = 0; F_2 = 1; P_1 = 18)$, $(F_1 = 1; F_2 = 6; P_1 = 16)$ and $(F_1 = 2; F_2 = 8; P_1 = 18)$, shown in red font in Fig. 7. The design of Algorithm 2 yields a total complexity, measured in number of XOR operations, of $O(m^N)$. Testing the entire error pattern to determine which would match the received syndrome (i.e., brute force scheme) would have a complexity of $O(m^{N+1})$. We can note that the complexity increases significantly with the number of errors considered $N$. We thus recommend using the algorithm when $N$ is low, depending on the processing time constraints of the targeted application. It is important to note that correcting a single error using algorithm 1 is not computationally more complex than performing a classic CRC check at the receiver.

## III. SIMULATION AND RESULTS

In this section, we present the theoretical and simulation performance of the proposed method, as compared to the lookup table approaches from the literature. Finally, we apply our method to Bluetooth Low Energy used in the IoT and compare its performance to state-of-the-art methods.

### A. CORRECTION RATE

In this section, we evaluate the performance of the proposed error correction method. It is able to instantly correct the packet when there is only one candidate in the output candidate list. Hence, the performance can be expressed as the percentage of error cases that produce only one candidate in the list. We will refer to such a percentage as the Single Candidate Ratio (SCR). The SCR is a function of three parameters: the generator polynomial used, the length of the protected data and the number of errors considered. Given a generator polynomial $g(x)$, we denote the $N$-error patterns for a payload length $m$ leading to a single candidate as $\mathrm{SinglePatterns}(m, N)$ and the total number of possible $N$-error patterns for the same length $m$ as $\mathrm{TotalPatterns}(m, N)$, and we can express the SCR as:

$$\mathrm{SCR}(m, N) = \frac{\mathrm{SinglePatterns}(m, N)}{\mathrm{TotalPatterns}(m, N)} \tag{9}$$

where $\mathrm{SinglePatterns}(m, N)$ was determined by running the algorithm over all the error cases and $\mathrm{TotalPatterns}(m, N) = \binom{m+n}{N}$. SCRs are thus not estimated but computed over the entire set of possible error patterns. We verify that when the length and the number of errors considered increase, the SCR decreases rapidly. Moreover, the length of the generator polynomial, as well as the number of non-zero coefficients, modify the way the SCR decreases as the length of the protected data increases.

In Figs. 9 and 10, we show the evolution of the SCR for different generator polynomials and different numbers of errors considered. We observe in Fig. 10 that when a long generator polynomial is used and few errors are considered, the SCR stays at 100% up to a significant length. On the other hand, a short generator polynomial leads to a faster decrease of the SCR as the packet's length increases, as illustrated in Fig. 9. When the SCR is 100% up to a certain threshold length for $N$ errors, it means that if $N$ errors or less occur during the transmission of the packet, these errors can be identified with a certainty of 100% and without any possible ambiguity when the packet's length is lower than this threshold. From this threshold, the SCR does not fall to zero immediately. Depending on the generator polynomial chosen, it can still be at a high percentage level up to a significant packet size. If we take the example of CRC-24 used in the Bluetooth Low Energy protocol [17] (of generator

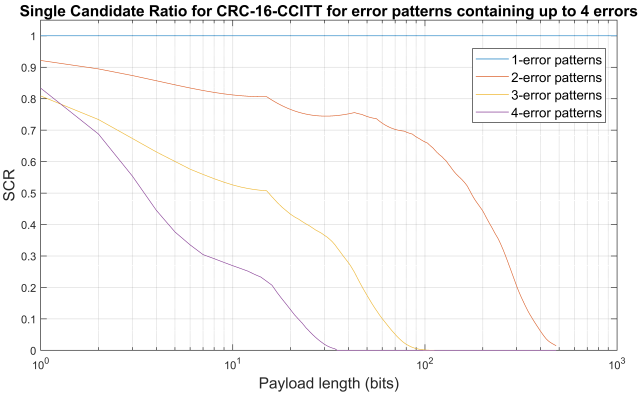


**FIGURE 9.** Single Candidate Ratio (SCR) for a payload length from 0 to 500 bits protected by a CRC-16-CCITT [25] of generator polynomial $g(x) = x^{16} + x^{12} + x^5 + 1$, considering up to 4 errors.

polynomial $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$), the SCR is still above 80% for a payload of up to 2000 bits when considering that two errors occurred in the packet. For three and four errors, this number decreases greatly, but is still over 80% for up to 220 bits for three errors and up to 85 bits when considering four errors. The applications targeted by the CRC-24 used in the Bluetooth Low Energy standard concern the Internet of Things (IoT) [23], [24]. In IoT environments, the average packet payload is often just a few bytes in size. Consequently, the proposed error correction method will be able to instantly correct most error patterns up to four errors, and even 100% of error patterns up to two errors, given a packet of 450 bits or less.

However, if the packet is highly corrupted, it may conceivably produce a syndrome the algorithm would recognize as the result of a low number of errors. In such a case, we would have a miscorrection. This corresponds, however, to very disadvantageous cases for all error correction methods. In fact, there is no error correction method that guarantees the validity of the reconstructed sequence. However, what we have is no more problematic than the case of highly corrupted packet yielding a CRC syndrome of zero, letting the receiver believe there is no error.

### B. MEMORY REQUIREMENTS

The proposed method does not require storing any table. In contrast, the main drawback of lookup approaches is their memory requirements. The table must be stored in the receiver's memory, as shown in Fig. 11. On very small-sized packet lengths as considered in [6] and [7], the lookup table represents a viable solution. When dealing with large packets, the required memory increases very rapidly. This rapid increase is also seen as the number of errors considered increases.

We evaluate the memory required when considering a specific number of errors and for each common syndrome length. Two different approaches are used to construct the lookup table. In both cases, the number of entries in the table corresponds to the number of possible error patterns. From this definition, it becomes clear that a lookup table that considers a packet length $M$ and $N$ errors would have $\binom{M}{N}$ rows. At each of these entries, the table must store the non-null syndrome for every possible error pattern. The syndrome is stored as a 1 to 4-byte number if we consider codes from CRC-8 to CRC-32 (used in Ethernet [21], of generator polynomial $g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$). To retrieve error positions, two strategies are used:

- The first one is to explicitly store the error positions associated with the syndrome in the table, as numbers coded on 16 bits (2 bytes) for each entry. There are $N$ such numbers per row. Using this lookup table design, the required memory, denoted $B_{exp}$, is expressed as:

$$B_{exp} = \binom{M}{N} \times [\text{length}(\mathbf{s}) + (2 \times N)] \tag{10}$$

  where $M$ is the total length of the packet, $N$ is the number of errors considered, and $\text{length}(\mathbf{s})$ is the size in bytes of the syndrome associated with the CRC used. The expression $(2 \times N)$ is the representation of $N$ 2-byte numbers per row, representing the positions of the $N$ errors considered. This implementation allows finding directly the error patterns associated with the syndrome but at a significant memory cost.
- The second strategy uses an implicit error position. With this approach, the lookup table does not need to store $N$ 2-byte numbers per entry, which reduces the total memory requirements by up to 9 times when considering a CRC-8 and four errors, as compared to the aforementioned strategy. The memory requirements, denoted $B_{imp}$, can now be expressed as:

$$B_{imp} = \binom{M}{N} \times \text{length}(\mathbf{s}) \tag{11}$$

  However, such a strategy involves more calculations to update the error pattern corresponding to the syndrome as it navigates through the table.

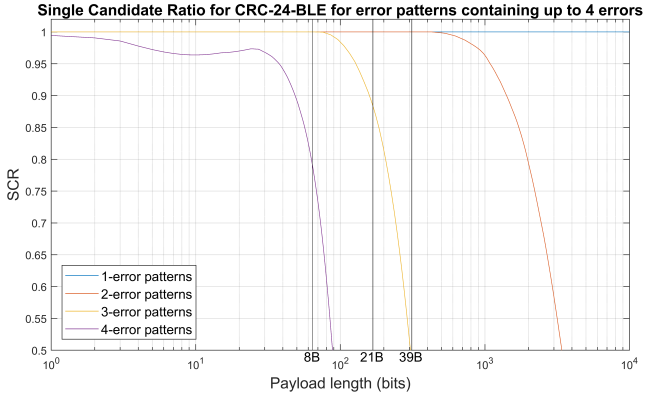


**FIGURE 10.** Single Candidate Ratio (SCR) for a payload length from 0 to 10 000 bits protected by a CRC-24-BLE [17] of generator polynomial $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$, considering up to 4 errors.

| Syndrome | Positions | | |
|---|---|---|---|
| 0000 0000 0000 0000 0000 0111 | 0 | 1 | 2 |
| 0000 0000 0000 0000 0000 1011 | 0 | 1 | 3 |
| 0000 0000 0000 0000 0001 0011 | 0 | 1 | 4 |
| 0000 0000 0000 0000 0010 0011 | 0 | 1 | 5 |
| ... | | | |
| 1111 1101 0010 1110 1011 0100 | 567 | 2504 | 8956 |
| 0110 0101 0001 0110 1111 0011 | 567 | 2504 | 8957 |
| 0101 0101 0110 0000 0010 0110 | 567 | 2504 | 8958 |
| ... | | | |
| 1111 0010 0010 1101 0000 0101 | 11996 | 11997 | 11998 |
| 0011 0111 1101 1000 1111 1111 | 11996 | 11997 | 11999 |
| 0101 0101 0010 0010 0000 0010 | 11996 | 11998 | 11999 |
| 1110 0100 0101 1100 0101 0001 | 11997 | 11998 | 11999 |

**FIGURE 11.** Examples of the explicit design of a lookup table containing all triple-error patterns for packet length up to 12000 bits.

**TABLE 1.** Memory requirements for storing the lookup tables considering a payload of 1500 bytes for several CRC lengths and number of errors considered with implicit and explicit error positions

| Nb Errors $N$ | CRC-8 | | CRC-16 | | CRC-24 | | CRC-32 | |
|---|---|---|---|---|---|---|---|---|
| | Implicit | Explicit | Implicit | Explicit | Implicit | Explicit | Implicit | Explicit |
| 1 | 12 kB | 36 kB | 24 kB | 48 kB | 36 kB | 60 kB | 48 kB | 72 kB |
| 2 | 72 MB | 360 MB | 144 MB | 432 MB | 216 MB | 504 MB | 288 MB | 576 MB |
| 3 | 288 GB | 2.02 TB | 576 GB | 2.30 TB | 864 GB | 2.60 TB | 1.16 TB | 2.88 TB |
| 4 | 864 TB | 7.77 PB | 1.73 PB | 8.64 PB | 2.59 PB | 9.50 PB | 3.45 PB | 10.4 PB |

We note that depending on the constraints present, one can choose among the two proposed designs to either save memory storage or save computations at the receiver side. Table 1 illustrates the memory requirements for both explicit and implicit implementations when considering large packets of 1500 bytes. We can see a significant increase in the memory requirements when considering each additional error. For such a packet length, considering three or four errors using a lookup table approach would be intractable.

### C. COMPUTATIONAL TIME COMPARISON

In terms of processing time, we ran the C implementation of the proposed algorithm for a single- and a double-error correction on a Raspberry Pi model 3B+ [22]. For comparison purpose, we also implemented a table approach capable of considering every single-error position for packets up to 1500 bytes. We executed both algorithms for packets of different lengths, from a few bits to the maximum size available here, set to 1500 bytes. The Raspberry model 3B+ used to conduct the experiment is equipped with a System on a Chip (SoC) Broadcom BCM2837BO with an ARM Cortex-A53 quad-core processor at 1.4 GHz and 1 GB SDRAM LPDDR2.

Figs. 12 and 13 show the relative time for the error correction method on single- and double-error patterns, respectively. The proposed method's complexity is compared to both lookup table approaches (i.e., explicit and implicit) for different packet sizes. Lookup table-based approaches have a constant complexity since they must always check every entry of the table prior to conducting error correction to ensure that all candidates are identified. When considering a single error, both table-based approaches are of equal complexity, and the conversion from explicit to implicit is straightforward. For two errors, however, the implicit method is 10 to 15% slower due to the computations required to convert the table index into error positions. We note that for a large payload, the methods are similar in terms of computational complexity. The lookup table approach is still faster when considering single errors in large packets. However, as the packet becomes smaller with respect to the maximum allowed packet size, the proposed method surpasses the lookup approach due to its adaptability to the received packet size. The method can be more than 10 times faster than lookup methods for very small packets, and the gain in speed is even greater when double-error correction is considered.


**FIGURE 12.** Evolution of the normalized time ratio to run the proposed method compared to lookup table-based approaches (explicit and implicit) for a single error in the packet depending on the length of the packet. The normalized time ratio corresponds to $155\mu s$ in this case.

When used as part of a standalone error correction process, the algorithm performs at its maximum in terms of both correction rate and complexity for small packets or CRC-protected headers. In such cases, the SCR is significantly high or at a maximum for multiple-error correction.

Comparing the proposed algorithm to the lookup table approaches in the literature, we can verify that it provides improved capabilities in two main respects:

1) **Flexibility**. Our method is more flexible than fixed-length lookup tables since it is not based on a specific packet size, but rather, is dynamically applied to protected data, and is thus adaptive to the data length. Consequently, the method will provide full coverage for any packet length. Furthermore, any generator polynomial, apart from the input parameter can be used with the proposed algorithm without modification. Lookup tables must be entirely recomputed when the generator polynomial considers changes. Alternatively, a table should be stored for each generator polynomial of interest, which significantly increases memory requirements.
2) **Memory-free multiple-error correction**. The proposed method does not assume that only single errors are likely to occur. Even if this scenario can still be supported by setting the number of errors to 1 as the input parameter, we can also assume that up to $N > 1$ errors are possible and consider the whole set of possible candidates up to this number. A lookup table approach is able to list such error patterns but needs an intractable amount of memory

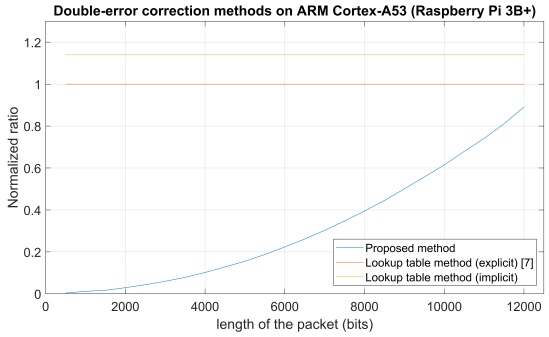


**FIGURE 13.** Evolution of the normalized time ratio to run the proposed method compared to lookup table-based approaches (explicit and implicit) for two errors in the packet depending on the length of the packet. The normalized time ratio corresponds to $1.4s$ in this case.

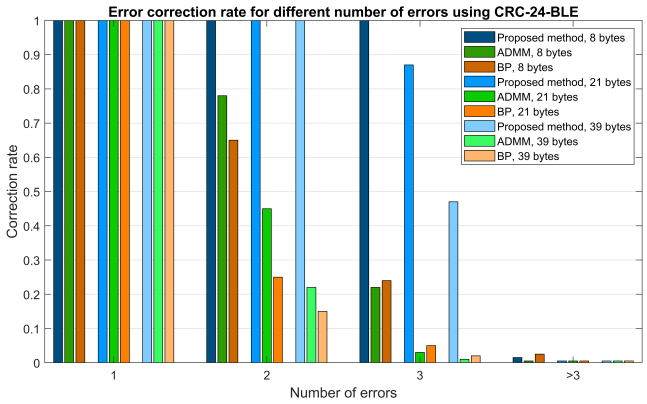


**FIGURE 14.** Error correction rate of the proposed method compared to two methods recently proposed in [11] for different number of errors in the packet

storage to consider the whole set of $N$-error cases in large packets. In order to optimize the management of the number $N$, further work can be carried out to dynamically choose it by extracting information about channel conditions, such as the channel Signal-to-Noise Ratio (CSNR) estimation at the physical layer or the Receiver Signal Strength Indicator (RSSI). The received RSSI level can be mapped to the crossover probability by measuring an average BER for each RSSI level. If this BER estimation is low enough in terms of the length of the packet, we can set the parameter $N$ to be 1.

### D. APPLICATION TO IOT

Considering the high performance of our method on small packets protected by strong generator polynomials, applying the proposed algorithm to the IoT domain can be highly desirable. A study of error distribution in a real environment of CRC-protected packets applied to the Internet of Things (IoT) [24] domain is proposed in [11]. The authors consider both Bluetooth Low Energy (BLE) packets protected by a CRC-24 and IEEE 802.15.4 [25] packets protected by CRC-16-CCITT. Two packet sizes, 21 bytes and 39 bytes, are considered. The results of the experiments are represented in Table 2 which shows that over 50% of the erroneous packets contain fewer than three errors in any selected scenario. Moreover, more than 40% contain two errors or less, making it an ideal context to evaluate our proposed method's performance. As noted the authors of [11], considering only slightly damaged packets can thus still enable a significant recovery rate. When soft information is unavailable, the authors of [11] propose to use a received packet's RSSI to determine the Bit Error Rate (BER).

In [11], the authors present the average correction rate of their methods when a specific number of errors occur in the packet. The simulation results can be seen in Fig. 14, considering three payload sizes: 8 bytes, 21 bytes and 39 bytes. To compare our algorithm with these approaches, we tested an exhaustive set of error patterns for each size and each number of errors to get the average correction rate over all possible error cases. We applied the algorithm for each error case and checked the resulting list at the end of the process. The correction is considered successful only if the actual error pattern is the only candidate in the output list. If there are no or several candidates in the list, the packet is considered lost. For the Alternating Direction Method of Multipliers (ADMM) and Belief Propagation (BP) [11], the simulation results in Fig. 14 show a maximum correction rate for single-error correction for all methods considered. For double-error correction, only the proposed method is able to achieve a 100% error correction. ADMM can correct an average of 80% for 8-byte payloads, which falls to less than 25% for 39-byte payloads. The results considering three errors are even more significant. The proposed method offers a 100% error correction rate for 8-byte payloads, whereas both ADMM and BP achieve 25%. When the payload length increases, the proposed method still can correct 86% and 47% for 21- and 39-byte payloads, respectively. Other methods can achieve a maximum of 5% error correction for such payloads.

These results can be retrieved in Fig. 10, where the three vertical bars correspond to the three payloads considered here. We can see that the correction rate for more than three errors is very low for all methods. In fact, it involves considering every error pattern containing more than three errors, which leads to a poor ratio since as the number of errors considered increases, the SCR decreases, becoming zero for large numbers of errors. However, we can note that

**TABLE 2.** Error distribution in real environment for BLE and IEEE 801.15.4 and two packet sizes

| Number of bit errors | BLE | | IEEE 802.15.4 | |
|---|---|---|---|---|
| | 21B | 39B | 21B | 39B |
| 1 | 18% | 16% | 11% | 10% |
| 2 | 28% | 27% | 30% | 27% |
| 3 | 12% | 11% | 15% | 16% |
| $>3$ | 42% | 46% | 44% | 47% |

we can still operate on four errors for small packets, as illustrated in Fig. 10 for 8-byte payloads, where the SCR is still 78%.

In [11], the authors propose a configurable iterative decoding process, which means that its performance will depend on the number of iterations performed on the corrupted packet. The results provided here consider 1000 iterations at the decoder. The timing for this decoding applied to the fastest method (ADMM) takes an average of 85 ms for 21-byte packets on a desktop computer with an Intel i7 3.1 GHz CPU, 8 GB RAM and Microsoft Visual C++ 2010 Compiler. We tested our method on a desktop computer with an Intel i7 3.4 GHz CPU, 8 GB RAM and GCC compiler, and we noted that depending on the number of errors to consider, it takes an average time ranging from 2 $\mu$s for single-error correction to 8 ms for three errors or less. Double-error correction takes an average of 150 $\mu$s. Therefore, the proposed method not only allows dramatically correcting more double- and triple-error cases, but it is also significantly faster than the state-of-the-art methods presented in [11].

### E. FUTURE WORK

In this paper, we have considered our algorithm as a standalone process that can only correct a packet when its output list contains a single element. In order to further increase the proposed method's error correction performance, it can be jointly used with other methods providing a list of potential error patterns as their output. For example, the work on UDP checksum proposed in [9], [26], [27] can be combined with our algorithm. Crosschecking both candidate lists would generate a matching list with a reduced number of entries. If our method is used in addition to the UDP checksum method, greater protected data lengths or a higher number of errors can be targeted for applications such as error correction on Ethernet frames, where a CRC covers the entire packet. Similarly, we could eliminate candidates leading to wrong values of known protocol fields, such as constant and predictable fields in the protocol's header (reserved and version fields are constant values during a communication, and some fields such as the sequence number in RTP are predictable since they are increased by 1 at each new packet throughout the communication). Some methods which consider a MAP approach have already proposed to use a CRC lookup table to validate their reconstruction, as described in [28] on Polar codes [29].It could be beneficial to compute only the probability of valid candidates rather than considering the whole set of possible sequences, determining their probability of being sent, and finally checking their CRC compliance.

## IV. CONCLUSION

In this work, we have proposed a novel algorithm to correct transmission errors within data covered by a CRC, using the computed non-null syndrome at the receiver. This method is able to instantly correct single errors if the protected data length does not exceed the period of the generator polynomial. This method is also able to correct multiple errors in small-sized packets, as used in the Bluetooth Low Energy standard. In such an environment, the proposed method achieves better error correction rates than the state-of-the-art methods considering up to three errors in the packet. The standalone error correction rate in BLE is at a maximum for single-, double- and some triple-error cases presented.

When instant correction is not possible, the algorithm still generates the list of all the possible error patterns that lead to the computed syndrome, according to a maximum number of errors considered. This list is usually small if we consider a reasonable number of errors. Further work to improve this method should use it in addition to existing methods that output a list of candidates. Crosschecking the lists of different methods would reduce the number of valid candidates, which would lead to fewer sequences to test or even to a reduction of the list size to a single candidate, allowing instant correction of damaged packets.

## REFERENCES


[1] J. Sobolewski, "Cyclic Redundancy Check," in *Encyclopedia of Computer Science*, John Wiley and Sons Ltd, 2003.

[2] J. Postel, "Transmission Control Protocol," IETF, RFC 793, Sept. 1981. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt.

[3] R. T. Braden, D. A. Borman, and C. Partridge, "Computing the internet checksum," IETF, RFC 1071, Sep. 1988. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1071.txt.

[4] K. Niu, K. Chen, "CRC-Aided Decoding of Polar Codes," in *IEEE Communications Letters*, vol. 16, no. 10, pp. 1668-1671, Oct. 2012.

[5] X. Liu, S. Wu, X. Xu, J. Jiao and Q. Zhang, "Improved Polar SCL Decoding by Exploiting the Error Correction Capability of CRC," in *IEEE Access*, vol. 7, pp. 7032–7040, Dec. 2018.

[6] S. Shukla, N. W. Bergmann, "Single Bit Error Correction Implementation in CRC-16 on FPGA," in *IEEE International Conference on Field-Programmable Technology*, Brisbane, Australia, pp. 319-322, 6-8 Dec. 2004.

[7] S. Babaie, A. K. Zadeh, S. H. Es-Hagi and N. j. Navimpour, "Double Bits Error Correction using CRC Method," in Fifth International Conference on Semantics, Knowledge and Grid, pp. 254-257, 12-14 Oct. 2009.

[8] A. S. Aiswarya and G. Anu, "Fixed Latency Serial Transceiver with Single Bit Error Correction on FPGA," *2017 International Conference on trends in Electronics and Informatics (ICEI)*, 11-12 May 2017.

[9] F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, P. Corlay, "Checksum Filtered List Decoding Applied to H.264 and H.265 Video Error Correction," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 8, pp. 1993-2006, Aug. 2018.

[10] E. Tsimbalo, X. Fafoutis, and R. Piechocki, "Fix It, Don't Bin It! CRC Error correction in Bluetooth low energy," in *Proceedings IEEE 2nd World Forum of Internet Things*, pp. 286–290, 14-16 Dec. 2015.

[11] E. Tsimbalo, X. Fafoutis, R. J. Piechocki, "CRC Error Correction in IoT Applications," in *IEEE Transactions on Industrial Informatics*, vol. 13, no. 1, pp. 361-369, Feb. 2017.

[12] F. Caron, S. Coulombe, "Video Error Correction using Soft-Output and Hard-Output Maximum Likelihood Decoding applied to H.264 Baseline Profile," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 7, pp. 1161-1174, Jul. 2015.

[13] P. Duhamel and M. Kieffer, "Joint source-channel decoding: A cross-layer perspective with applications in video broadcasting," Academic Press, 2009.

[14] G. Zhang, R. Heusdens, and W. B. Kleijn, "Large scale LP decoding with low complexity," *IEEE Communication Letters*, vol. 17, no. 11, pp. 2152–2155, Nov. 2013.

[15] S. Sankaranarayanan and B. Vasic, "Iterative decoding of linear block codes: A parity-check orthogonalization approach," *IEEE Transactions on Information Theory*, vol. 51, no. 9, pp. 3347–3353, Sep. 2005.

[16] IEEE 802.11: Part 11: "Wireless LAN medium access control (MAC) and physical layer (PHY) specifications", Dec. 2016.

[17] Specification of the Bluetooth system. Core Version 4.1, Bluetooth SIG, 2013. [Online]. Available: http://www.bluetooth.com.

[18] S. Boyd and L. Vandenberghe, "Introduction to Applied Linear Algebra – Vectors, Matrices, and Least Squares", Cambridge University Press, 2018.
[19] J. Arndt, "Binary polynomials," In: Matters Computational. Springer, Berlin, Heidelberg, pp. 822-863, 2011.
[20] N. Bhatnagar, "Mathematical Principles of the Internet, Volume 1: Engineering", Chapman and Hall / CRC, Dec. 2018
[21] IEEE Standard Association, "IEEE 802.3-2018 - IEEE Standard for Ethernet", 2018, [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html
[22] Raspberry PI 3 Model B+. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.
[23] A. Zanella, N. Bui, A. Castellani, L. Vangelista and M. Zorzi, "Internet of Things for Smart Cities," in *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22-32, Feb. 2014.
[24] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," in *IEEE Communication Surveys and Tutorials*, vol. 17, no. 4, Jun. 2015.
[25] "IEEE Standard for Information Technology — Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)," IEEE Std. 802.15.4-2006.
[26] F. Golaghazadeh, S. Coulombe, F-X. Coudoux and P. Corlay, "The Impact of H.264 Non-Desynchronizing Bits on Visual Quality and its Application to Robust Video Decoding," *2018 IEEE International Conference on Signal Processing and Communication Systems (ISPCS 2018)*, 17-19 Dec. 2018.
[27] F. Golaghazadeh, S. Coulombe, F-X. Coudoux and P. Corlay, "Low Complexity H.264 List Decoder for Enhanced Quality Real-Time Video over IP," *30th annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2017)*, 30 Apr. - 3 May 2017.
[28] K. Niu, K. Chen, "CRC-Aided Decoding of Polar Codes," on *IEEE Communications Letters*, vol. 16, no. 10, pp. 1668-1671, Oct. 2012.
[29] E. Arikan, "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels," in *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051-3073, Jul. 2009.