IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 23, 2022, accepted February 19, 2022, date of publication February 28, 2022, date of current version March 8, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3155457

CRC-Based Correction of Multiple Errors Using an Optimized Lookup Table

VIVIEN BOUSSARD[1,2], (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]CNRS, UMR 8520, ISEN, Centrale Lille, DOAE—Département d'Opto-Acousto-Électronique, IEMN—Institut d'Électronique de Microélectronique et de Nanotechnologie, University of Lille, Université Polytechnique Hauts-de-France, 59313 Valenciennes, France

Corresponding author: Stéphane Coulombe (stephane.coulombe@etsmtl.ca)

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant, and in part by the Université Polytechnique Hauts-de-France (UPHF).

ABSTRACT In this paper, we propose a new approach to perform multiple error correction in wireless communications over error-prone networks. It is based on the cyclic redundancy check syndrome, using an optimized lookup table that avoids performing arithmetic operations. This method is able to achieve the same correction performance as the state-of-the-art approaches while significantly reducing the computational complexity. The table is designed to allow multiple bit error correction simply by navigating within it. Its size is constant when considering more than two errors, which represents a tremendous advantage over earlier lookup table-based approaches. Simulation results of a C implementation performed on a Raspberry Pi 4 show that the proposed method is able to process single and double error corrections of large payloads in 100 ns and 642 μs, respectively, while it would take 300 μs and 1.5 s, respectively, with the state-of-the-art CRC multiple error correction technique. This represents a speedup of nearly 3000× for single error and 2300× for double error correction, respectively. Compared to table-based approaches, the proposed method offers a speedup of nearly 1200× for single error and 2300× for double error correction under the same conditions. We also show that when multiple candidate error patterns are present, numerous errors can be corrected by adding a checksum cross-validation step.

INDEX TERMS Cyclic redundancy check, error correction, lookup table, checksum control, wireless communication.

I. INTRODUCTION

In wireless communications, cyclic redundancy checks (CRCs) [1], [2] are widely adopted in order to enhance the communication reliability between a transmitter and a receiver. Indeed, CRCs are broadly used at different layers of the protocol stack of a data transmission as they detect transmission errors at the receiver. For instance, CRCs are present at the physical layer of widely deployed wireless protocols such as 802.11 [3] to protect the header of the packet, and at the Medium Access Control (MAC) layer, to protect the entire packet. An example of the latter is the 802.3 Ethernet protocol [4]. In general, CRCs are only used to detect whether a transmitted packet has bit errors, in which case the erroneous packet is discarded.

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

CRCs are computed from two main components: the protected bitstream, i.e., the message or data to transmit, which we will call the payload, denoted $d_T(x)$ for transmitted data, and a generator polynomial, denoted $g(x)$. Both $d_T(x)$ and $g(x)$ are binary polynomials. The generator polynomial is a constant binary polynomial, which is defined according to the transmission standard used. There are various generator polynomials in use, differentiated by their length and their number of non-null coefficients. Larger generator polynomials will lead to stronger error detection capabilities, but at the cost of higher packet overheads.

The typical transmission and reception of a CRC-protected data packet follow three main steps [1]. Firstly, at the transmitter, the payload $d_T(x)$ is left-shifted by n positions, where n is the degree of the generator polynomial, to produce the transmitted packet $p_T(x) = d_T(x).x^n$. This result is then divided by the generator polynomial $g(x)$ to obtain the remainder of this division, denoted $r_T(x)$. Secondly, $r_T(x)$

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

is appended to the payload to be sent, and occupies the n rightmost positions of the transmitted message. The transmitted message is thus $d_T(x).x^n + r_T(x)$, where $+$ is the addition which corresponds to an *exclusive or* (XOR) between two binary polynomials,[1] and has a total length of $m + n$ bits, where m is the payload length of the message. Thirdly, at the receiver, the received packet, denoted p_R, is divided by the generator polynomial $g(x)$ in order to check if an error occurred during the transmission. If the message is intact, the remainder of the division by $g(x)$, called the computed CRC syndrome $s(x)$, will produce a result equal to zero since $r_T(x)$ has been added to generate an entire multiple of $g(x)$. On the other hand, a non-zero CRC syndrome indicates an error in the transmitted message.

In currently deployed systems and methods, conventional CRC error management involves discarding a packet when an error is detected at the receiver; that is, when the remainder is not equal to zero at the receiver. These approaches do not allow for packets to be corrected. In such systems, a received corrupted packet is processed as a lost packet. For example, a packet having a single bit in error, and which would otherwise be a good packet, will be detected as an erroneous packet and will be entirely discarded. Discarding entire packets having only one or a few errors leads to a significant loss of valuable information which could have been exploited if the errors had been corrected.

Therefore, various methods, described in Section II, were developed to exploit the CRC to perform error correction in a packet [6]–[16]. These methods include Meggitt decoding [8] and its practical variation called *error-trapping* which is capable of correcting, with high efficiency, single errors, short double errors and burst errors [9]–[12]. This is achieved using specialized circuitry. However, the approach cannot handle errors which are not close to each other in a packet. To avoid complex computational circuitry to identify the error positions, some approaches consist in storing the different syndromes produced by the error patterns in a lookup table [1]. This is typically performed for small packets and for a single error to maintain the size of these lookup tables reasonable. For instance, in [15], the authors propose a fixed latency serial transceiver with single bit error correction using a lookup table. However, when considering challenging applications such as video streaming, where packet length may be large and where multiple errors may occur in a same packet, this kind of solution becomes impracticable. In this paper, we propose an alternative and efficient method for CRC-based multiple error correction using an optimized lookup table. The proposed method is not limited to a specific packet size and is able to handle multiple errors. It is based on unpublished research work from [17]. In our method, most of the required operations are done offline and then stored in a table. The table is designed to allow the results for single and double error patterns to be accessed by simply navigating through the table, which results in significant processing time reductions versus table-free methods. The proposed approach can also be applied to identify any number of errors and at any position, with significant speedups. This paper does not intend to compete with existing error-correcting codes (e.g., turbo, LDPC, polar), but rather provides a new error-correction method that takes advantages of the widely used CRC present in the protocol stack. In fact, the proposed method can be used in addition to the codes mentioned above to reinforce the robustness of transmitted data against channel errors in a cross-layer context. The following are the benefits and contributions of the proposed method:

- **Significant speed gains**: The proposed approach is designed to allow most to all of the arithmetic operations required in the state-of-the-art table-free error correction approach [18] to be performed offline and stored in a table. This design contributes to greatly reducing the processing time of the proposed method.
- **Fixed-length tables**: State-of-the-art table-based approaches define distinct tables for each number of errors and maximum packet size considered. Furthermore, these tables grow in size exponentially with the number of errors considered, making them impractical when considering 3 or more errors. In contrast, the tables derived and used for the correction of multiple bits in the proposed approach are fixed for a given generator polynomial, and therefore possess a fixed length, regardless of the number of errors or packet size under consideration. We also propose another version of the table for the special case of a single error correction, which requires less memory storage.
- **Analysis of the syndromes and of the table structure**: We provide equations to identify, for any generator polynomial, syndromes with special properties (e.g., syndromes for which there is no valid error pattern comprising a single erroneous bit). We also highlight the cyclic properties of the table-based process which produces syndrome elements looping on themselves.
- **Applicability to multiple error correction**: Because of its large speed gains and reasonable table sizes, the proposed method is well-positioned to correct multiple errors. Also, unlike state-of-the-art methods [8]–[12] only handling errors occurring in bursts, the proposed method can handle multiple errors regardless of their individual positions in the packet. However, when multiple errors are considered, and especially when the burst and maximum packet size conditions are relaxed, the candidate error patterns leading to the computed syndrome are numerous, making the identification of the true error pattern challenging. We show that we can significantly reduce the list of candidates, even to the point of having a single one and being able to correct the packet, by performing an additional validation step in the form of testing of the candidates with the checksum found in UDP and TCP protocols.

[1] Binary packets of length m belong to the Galois Field GF(2^m), where the addition is performed as the bitwise XOR [5].

This paper is mainly based on the theory described and discussed in [18], with which the reader is expected to be familiar. The rest of the paper is organized as follows. In Section II, we introduce related works on CRC error correction covering lookup table methods and state-of-the-art CRC multiple error correction. In Section III, we present the concepts and implementation of the proposed optimized table, as well as its use in multiple error correction systems. We also analyze the structure of the table and syndromes with special properties. In Section IV, we evaluate the performance of the proposed approach in terms of processing speed and memory usage, as compared to existing methods. In Section V, we conclude and give an overview of future research works. We assume that the reader is familiar with the notations and concepts described in our previous works [18], especially those related to the Galois Field GF(2) [19] and its generalization to GF(2^m).

II. RELATED WORKS

Several works have explored the error correction possibilities of error detection codes, such as CRC [8]–[16] or checksums [20]. They can be categorized as table-based and table-free approaches. As described in [1] and more recently in [13], [14], [16], the table-based approach consists in computing a lookup table (LUT) prior to communication. In this scheme, each LUT entry contains the pre-computed syndrome corresponding to a specific single error position in the received packet. An example of such a table for CRC-8-CCITT is given in Table 1. A search for the computed syndrome is performed in the table. If a match is found, the bit at the corresponding position is flipped. The number of entries in the lookup table is based on the size of the expected payload, and is constant. It has been applied to other generator polynomials by [15]. Such tables have been recently exploited in improved successive cancellation list (SCL) decoding schemes of Polar codes [16]. This method is fast when conducted on small packets, but suffers from some serious disadvantages. Firstly, the approach assumes that a single error occurred in the packet, yielding a miscorrection probability in severe channel conditions, since a highly corrupted packet can produce the same syndrome as a single error. Furthermore, to support the correction of multiple errors, methods based on this approach must store the syndromes of all combinations of error positions. Consequently, memory requirements grow exponentially with the number of errors considered, limiting their use to small packet sizes and few errors in practice. For instance, as shown in [16], a table conceived to correct all single and double errors in packets having bit lengths of 128 bits requires 128 entries for single error positions and 8128 entries to cover all double error positions.

Table-free approaches rely on on-the-fly arithmetic operations instead of pre-computed lookup tables to perform error correction. They include Meggitt decoding [8] and error-trapping [9]–[12] for which multiple error correction is possible only when errors are concentrated in a region of the packet not exceeding the CRC size, meaning that they cannot correct errors which are located far apart in the packet. In [18], we introduced a generic table-free multiple error correction. This approach generates an exhaustive list of all error patterns, regardless of where they are located in the packet, corresponding to the computed syndrome up to a predetermined (desired) number of errors. The data packet includes a payload $d_T(x)$ and cyclic redundancy check (CRC) information. The latter is calculated using a generator function or polynomial $g(x)$. The method generates an exhaustive list of valid error patterns containing N errors or less, by exploiting the definition of CRC computation. The list can comprise one or several candidates depending on $g(x)$, the CRC syndrome and the packet length. When the list contains several candidates, various methods can be applied to identify the error pattern within the list that actually occurred. For instance, in [21], we proposed using a UDP checksum and a non-desynchronizing bits validation steps to eliminate invalid candidates in the context of error-prone transmission of H.264 baseline profile compressed video streams. In [22], we combined a UDP checksum and video decoding validation steps for H.264 and H.265 video communications over 802.11p and Bluetooth Low Energy wireless networks. We demonstrated the possibility of correcting up to 3 errors in a video packet. In [23], we extended our work to estimate the number of candidates in the list and applied our method to correct up to 5 errors. All the approaches led to substantial video quality improvements. Another example of using additional information to perform CRC-based multiple bit error correction is found in [24] where the authors identify the bits having a high error probability to determine the most likely error patterns. This shows that although the proposed method generates several candidate error patterns, depending on the targeted

TABLE 1. Example of lookup table for single error correction as proposed in the literature [13], applied to CRC8-CCITT of generator polynomial $g(x) = x^8 + x^2 + x + 1$, considering a 10-bit payload.

Error position	Associated syndrome
0	0000 0001
1	0000 0010
2	0000 0100
3	0000 1000
4	0001 0000
5	0010 0000
6	0100 0000
7	1000 0000
8	0001 1101
9	0011 1010
10	0111 0100
11	1110 1000
12	1100 1101
13	1000 0111
14	0001 0011
15	0010 0110
16	0100 1100
17	1001 1000

application, it is possible to eliminate all but one candidates using additional information present at other layers of the protocol stack or from the application itself.

We now summarize the main approach described in [18]. We can express the syndrome computed at the receiver as:

$$s(x) = (d_T(x).x^n + r_T(x) + e(x)) \bmod g(x) \quad (1)$$

where $e(x)$ represents the potential error pattern that corrupted the packet during the transmission, and where erroneous positions in $e(x)$ are identified by values of 1. From this definition, it is clear that the result of $(d_T(x).x^n + r_T(x)) \bmod g(x)$ is zero, as $r_T(x)$ is the remainder of the division of $d_T(x)$ by $g(x)$, and:

$$s(x) = e(x) \bmod g(x) \quad (2)$$

If we isolate the error vector, $e(x)$, we obtain:

$$e(x) = s(x) + q(x).g(x) \quad (3)$$

where $q(x)$ can be any binary polynomial of the highest degree $(m-1)$, with m being the payload length. As the total number of possible values of $q(x)$ is too high (there are 2^m such polynomials), the method proposes focusing on lightly corrupted packets and then building $q(x)$ term by term. This is achieved by canceling the Least Significant Bit (LSB) term of the current $e(x)$, at each step of the process, through the addition of properly left-shifted $g(x)$ (i.e., so that its LSB term is aligned with the term of $e(x)$ to cancel[2]). Adding $g(x)$ aligned at any position maintains the class equivalence of (3) and produces error pattern candidates having the same computed syndrome at each step [18]. The number of non-null coefficients in the polynomial $e(x)$ corresponds to the number of errors in the current candidate. The method thus only keeps candidates when the number of non-null coefficients in $e(x)$ is equal to or less than a predefined number N. To handle the correction of multiple bit errors, the method forces term values in $e(x)$ throughout the process (i.e., it sets such terms to 1 or turns them into a 1 by adding $g(x)$ aligned with their positions). Once $(N-1)$ positions have been forced, single error management is performed on the remaining length of the packet to locate the last error. If this last error does not exist, it means that the forced term values did not correspond to a valid error pattern for the given syndrome (i.e., an error pattern leading to the computed syndrome).

Although this method requires very little memory space, one drawback with it, however, is that it is very complex when considering several errors in a packet. This complexity, measured in the number of additions involving $g(x)$, is $O(m^{N-1})$, where m represents the payload length and N the number of errors considered. It thus grows exponentially with N.

III. PROPOSED METHOD

While the state-of-the-art method discussed in the last section aims at generating the list of valid error patterns with arithmetic operations, the proposed method does the same by exploiting tables, and accordingly avoids most arithmetic operations. Unlike previous methods [13], where tables are sparse and contain error positions only for CRC syndromes compatible with a certain packet size (e.g., that associated with a standard), the proposed tables provide error patterns for every possible syndrome value. In addition to being usable for any packet size, these tables can be indexed directly by syndrome value instead of being searched, as was the case in the previous case. Indeed, the process of searching for a pattern in a table, as in the previous approaches, is computationally intensive. This method thus provides enhanced speed performance, but at the cost of higher memory requirements. However, the proposed tables are solely dependent on the generator polynomial of interest, and are fixed for any number of errors N, while the previous methods lead to tables whose sizes grow exponentially with N.

In what follows, we will use the notation a to represent the binary vector associated with a binary polynomial $a(x)$. The i-th element (LSB-wise) of this vector will be denoted a_i. The addition of polynomials $a(x)$ and $g(x)$, denoted $a(x) + g(x)$, will become the XOR of vectors a and g, denoted $a \oplus g$. The left shift of $g(x)$ by n positions, denoted $g(x).\ x^n$ will be denoted $g \ll n$. The addition of two vectors should be interpreted as an XOR operation.

A. SINGLE ERROR CORRECTION

The state-of-the-art method achieves CRC syndrome simplification through successive additions (XORs) of the current error pattern vector (initialized to the computed syndrome) with left-shifted versions of the generator polynomial, and an updating of the error vector e with the result of the addition until a single-bit error pattern appears in e (i.e., a single-bit is set to 1). In the proposed method, we aim to avoid these computations by storing the location (relative distance) of the single error corresponding to each syndrome value in a table. Because the table is directly indexed by these syndrome values, the error location can be accessed directly by using the syndrome as an index in the table. The first step of the proposed method is thus to generate the table for single error correction, following the steps illustrated in the flowchart given in Figure 1.

Step 1: The table is initialized to -1. We choose this initialization value as it cannot be a valid entry in the list, contrary to 0, which would correspond to a single error at position 0. The size of the table corresponds to the size of the possible set of syndromes, (i.e., $2^n - 1$, where n is the bit length of the syndrome s).

Step 2: The syndrome is then initialized to 0, which is equivalent to null vector $\mathbf{0}$, although this value is not of interest as it would indicate that there is no error in the packet.

Step 3: Next, the single error position P_1 associated with the syndrome s for a given polynomial g is determined. The sub-steps to perform are illustrated in a separate flowchart in Figure 2, where it can be seen that the state-of-the-art single error algorithm is actually performed in steps 1, 2, 4, 5 and 6. Figure 2 presents an error handling process similar

[2]In this paper, we assume that $g_n = g_0 = 1$ as observed in practice.

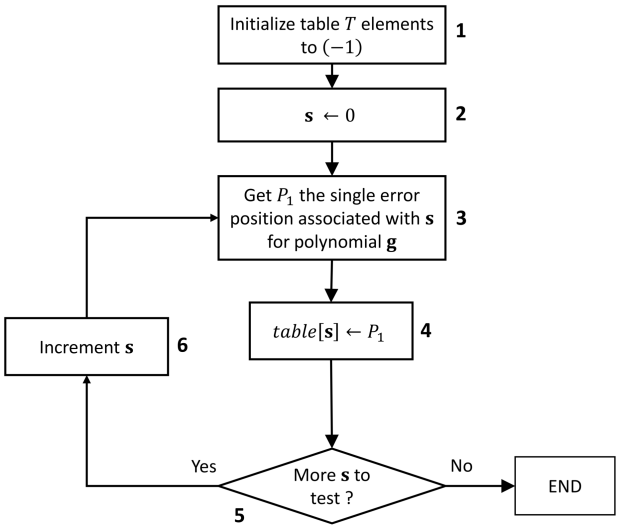


FIGURE 1. Flowchart representing the steps to build the table containing the whole list of syndromes along with their associated single error positions.

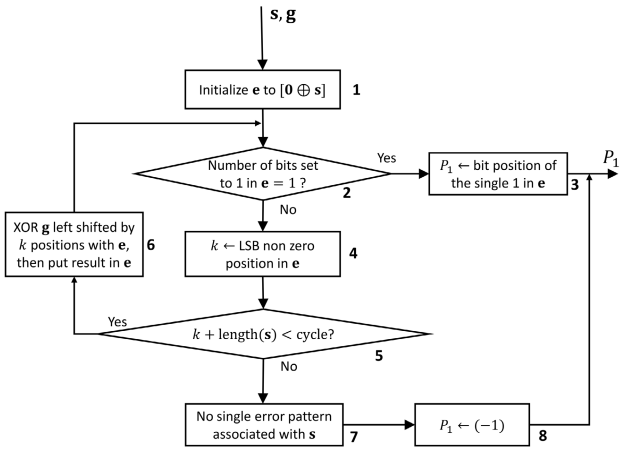


FIGURE 2. Flowchart representing the steps to obtain the single error position from a computed syndrome (step 3 in Fig 1). In step 1, **0** represents the null vector.

to the CRC-based single error correction available in the literature, where the packet length m is set to the largest packet size before single error periodicity (i.e., we denote this length as the cycle length of a generator polynomial). The first step is to initialize e to m zeros and set its LSB value to s. Then, for each syndrome, we thus count the number of non-null values in the error vector e and check if this number equals 1. If that is the case, the corresponding error position is set in the corresponding entry of the table. If not, the LSB non-null value is canceled and the error vector is checked again, until we reach the cycle length. The cycle length for any generator polynomial is equal to or less than $2^n - 1$, and can be retrieved experimentally by determining the distance between two single error patterns having the same syndrome.

Step 4: The single error position provided by step 3 is stored in the table.

Step 5: If the whole set of possible syndromes has been tested (final value of s reached), the process stops. If not, the syndrome is updated by incrementing its value by 1.

At the end of the process, a table similar to Table 2 is obtained, and shows an example for the polynomial generator $g(x) = x^5 + x^4 + x^2 + 1$. The index column is actually implicit and does not need to be stored; in fact, it was added here simply for better readability. In this table, P_1 denotes the degree of the single error position (i.e., the single non-null position in e). For instance, looking at the table, the error position is x^{10} when the computed syndrome is 7. It can be seen that half the indexes indicate a position P_1 of (-1). This notation means that there is no single error position or solution associated with the corresponding syndrome value. Since the generator polynomial's parity is even, it mainly corresponds to syndromes with even parity. Of note, when the generator polynomial has even parity, syndrome values with even parity cannot lead to odd parity error patterns, and thus, they cannot lead to single error solutions. We will see later in this paper that there is also a specific syndrome for which no solution exists when the generator polynomial has even parity.

TABLE 2. Single error position table generated for a CRC-5 of generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$.

Index	P_1	Index	P_1
0	-1	16	4
1	0	17	-1
2	1	18	-1
3	-1	19	-1
4	2	20	-1
5	-1	21	5
6	-1	22	8
7	10	23	-1
8	3	24	-1
9	-1	25	9
10	-1	26	14
11	7	27	-1
12	-1	28	12
13	13	29	-1
14	11	30	-1
15	-1	31	6

The steps for identifying candidate single error positions when receiving a corrupted packet are illustrated in Algorithm 1. The method proceeds by simply checking the value of P_1 in the table at the index corresponding to the computed syndrome s, as shown in step 2. If P_1 is different from (-1) and indicates a value less than the length of the packet, a candidate is appended to the list. A search for more valid candidates is conducted, considering the cycle length of the generator polynomial used. For example, in Table 2, the cycle length is $(2^4 - 1) = 15$. Thus, the position P_1 ranges from 0 to 14, producing corresponding syndromes obtained

from the Index column in the table (e.g. syndromes of 1 and 26 correspond to $P_1 = 0$ and $P_1 = 14$, respectively). If an error occurs at position 15, as it corresponds to position $P_1 = (0 + cycle)$, it will produce a syndrome equal to 1. For a packet of length 50, when computing syndrome 1 (i.e., the same as position $P_1 = 0$), the error can thus be at positions 0, 15, 30 and 45. Since valid single error positions are cyclic, we test all possible values up to the length of the received packet. An error correction method may correct the packet if a single candidate is returned by Algorithm 1 or if one of them passes additional validations such as a UDP or TCP checksum [25], [26].

The proposed single error correction approach differs from the state-of-the-art lookup table approaches as it considers the whole set of possible syndromes that can occur, regardless of the packet length. Moreover, current lookup table approaches are sorted in a bit error position-wise order, making it mandatory to scan the table to retrieve the error position. By sorting the table in a syndrome-wise order, we do not have to scan the table as it can be indexed with the computed syndrome value.

B. DOUBLE ERROR CORRECTION

In order to handle double error correction, the strategy proposed in the state-of-the-art CRC-based multiple error correction method is to force a first error and then search for the second using the single error method [18]. As the first error is forced at a specific position F_1 by setting all lower positions to 0 and this position to 1 through successive conditional additions of g at required positions, the resulting syndrome is expected to correspond to a single error syndrome. Checking the position of this error by applying the single error method will indicate whether the remaining error at position P_1 (as well as the other single errors at positions separated by the cycle length for sufficiently large packets) occurs within the packet, which would lead to a new candidate error pattern with errors at positions (P_1+1+F_1) and F_1. Thus, to consider the entire set of possibilities, single error management must be called for each possible location of the first error. At the end of each verification, the LSB error is moved by one bit toward the most significant bit (MSB), from the LSB position where $F_1 = 0$ to the end of the payload at position $F_1 = m - 1$. This repositioning is conducted in two steps:

- The previous bit position tested (former F_1, forced to 1) is set to 0 by XORing g at this position.
- The following bit toward the MSB is considered as the new first error (i.e., $F_1 \leftarrow F_1 + 1$), and is either kept at 1 if it was 1 or forced to 1 by XORing g at this position if it was 0.

The process is repeated until the forced bit position reaches the position m, corresponding to the length of the protected data (position $m - 1$ is the last position processed).

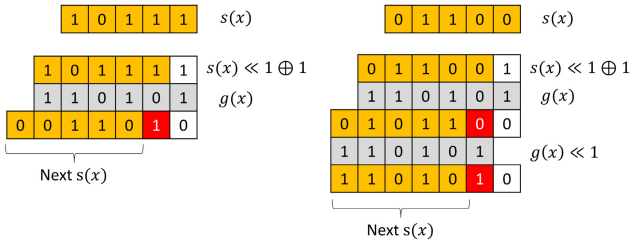
It can be seen that based on these steps, the succession of syndromes is always the same for a given generator polynomial, and therefore we propose, in this method, to store the value of the next syndrome based on the current one. More specifically, the *next* element s' of a given syndrome s is:

$$next = \begin{cases} (s' \oplus g) \gg 1 & \text{if } s_0 = 0 \\ s' \gg 1 & \text{if } s_0 = 1 \end{cases} \tag{4}$$

where s' is defined as:

$$s' = [((s \ll 1) \oplus 1) \oplus g] \gg 1 \tag{5}$$

The form of (5) actually corresponds to an updating of the forced error position by canceling the previously forced one and setting the new forced position to 1. These steps are illustrated in Figure 3, where both cases discussed are presented. In Figure 3a, the syndrome is first left-shifted by one position and 1 is appended as the LSB, which represents the formerly forced position. We cancel this position by XORing the generator polynomial. As the new forced position (represented by a red box) is already set to 1, no further step is required, and we simply consider the next syndrome.



(a) Next element case 1 (b) Next element case 2

FIGURE 3. Illustration of the next elements generation for the two possible cases. In case 1, the newly forced value is already set to 1 after the first XOR of $g(x)$. In case 2, the newly forced position is 0 after the first XOR, which requires a second XOR of $g(x)$ to force this position to 1. The red boxes represent the new position to force.

Algorithm 1 SingleErrorCorrection($T[2^n]$,s,n,m,$cycle$)

Inputs:

$T[2^n]$: indexed table containing P_1 for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial

Output:

E_1: the list of valid error positions for single-bit error

```
E1 ← {}
P1 ← T[s]
if P1 ≠ −1 then
    while (P1 < m + n) do
        Add P1 to E1
        P1 ← P1 + cycle
    end while
end if
Return E1
```

On the other hand, in Figure 3b, the new forced position is not set to 1 after the first XOR. In this case, therefore, another XOR with the generator polynomial is needed to produce the next syndrome. In the example of Figure 3a, it is clear that starting from the syndrome $s = [10111] = 23$, we cancel the previously forced value by XORing the generator polynomial and leave the new position to force as 1, identified as a red box. We thus obtain the next element for the syndrome equal to 26, which is $s = [00110] = 6$. Linking a syndrome to the next will significantly reduce the complexity of the approach since the arithmetic operations will be pre-computed and stored in the reference table.

In [18], it was shown that throughout the process of identifying error patterns with N or fewer errors, which operates from the LSB to the MSB, only a range of n bits can contain non-zero values since the lower positions are already eliminated and the higher positions, initialized to 0, have not yet been altered. The proposed method exploits this property by considering only values within this range (sliding window) and processing them as the syndromes of interest. In the case of a double error, the forced error position will alter the syndrome value on which our subsequent single error position determination is based. Essentially, we consider the effect of forcing the error position on the syndrome used for identifying the remaining error position. This latter position becomes relative to the forced position. It should be noted that the forced error position, represented by a red box in Figure 3a, is not part of the syndrome considered for single-bit error determination, but is implicitly present throughout the process described in (5).

The following are the steps for generating the table containing both the single error position and the next element, as illustrated in Figure 4:

Step 1: The whole table is initialized to (-1). The number of entries in the table is the same as for the single error table previously described, with the difference lying in the number of columns. For each row, another column containing the next element of the current syndrome is appended.

Step 2: We initialize a, a variable representing the first syndrome of the loop, to 1.

Step 3: We now begin to complete the table. In order to avoid unnecessary computations, we first check that the current table position has yet not been processed and that the current values of a and g are able to produce a candidate.

Step 4: If that is the case, we initialize the next element to 0 and set a local syndrome b to the value of a.

Step 5: We first fill the single error position through to steps described in the previous section (see Figure 2).

Step 6: The next element is identified for the current syndrome b and the generator polynomial g. The steps to determine the next element are described in a separate flowchart in Figure 5. To generate the next element, we cancel the previously forced bit position by XORing g and set the new forced position through a new XOR if necessary.

Step 7: We store the single error position and the next element in the table.

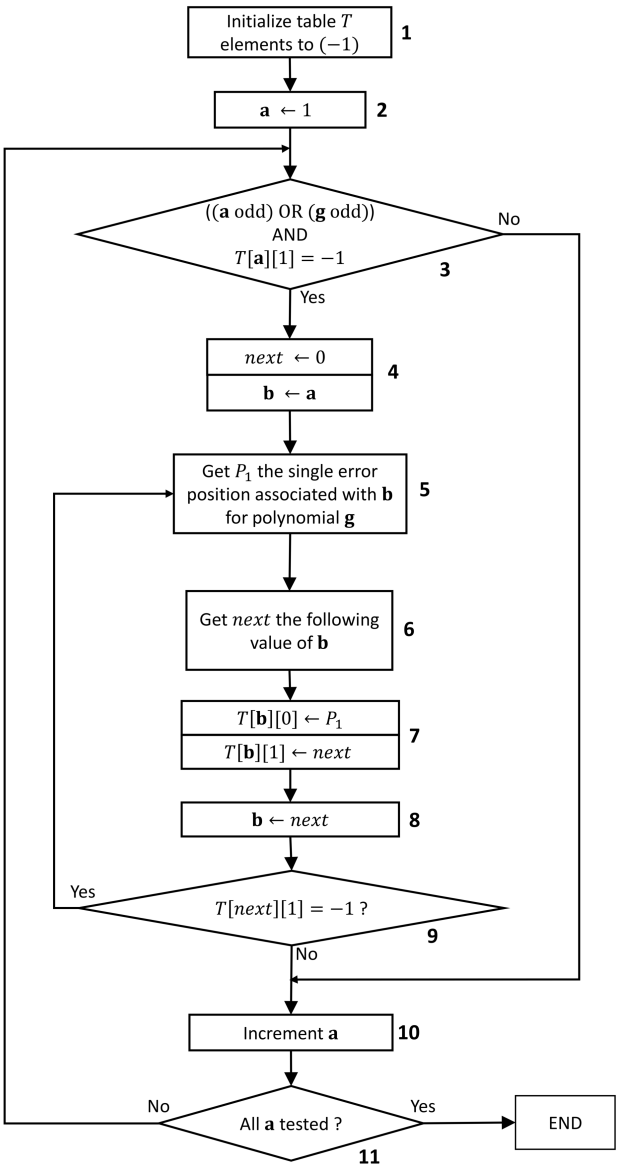


FIGURE 4. Flowchart representing the steps to generate the table T containing single error position and next element to handle double error correction.

Step 8: The local syndrome b is then updated to the next position computed in step 6.

Step 9: If the next element has not yet been processed, we keep on computing its associated single error position and next element. If the next element has already been processed, the current cycle of *next* has looped, which means that we have to increment a and start a new loop.

Step 10: Once every value of a has been tested, the process is ended as the table is complete.

Table 3 shows the complete table at the end of the process for the generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$. The P_1 column corresponds to the relative distance of the remaining single error, and the *next* column stores the next syndrome to be tested (i.e., the syndrome after the forced

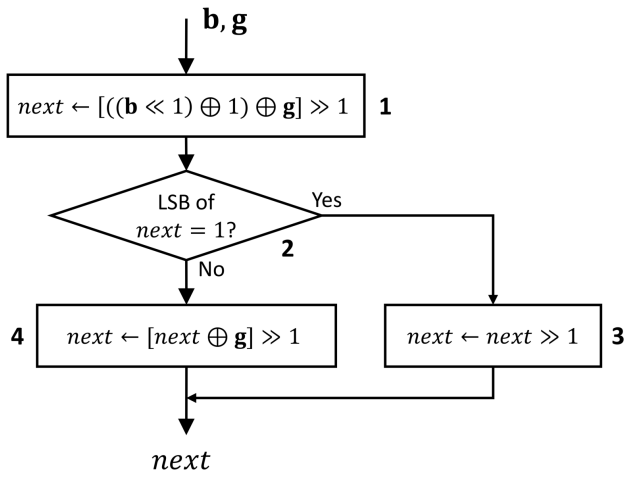


FIGURE 5. Flowchart representing the steps to follow to generate the next element from the computed syndrome and the generator polynomial (step 6 of Fig. 4, illustrated in Fig. 3).

position is updated). Algorithm 2 illustrates the different steps for performing the correction of double-bit errors when the table is generated:

Step 1: The candidate list containing the candidates with 2 errors is initialized as an empty set.

Step 2: A vector s' of length n bits is created, and it will correspond to the updated version of the original syndrome s.

Step 3-7: As each *next* element from the table has been generated given that an error had already been forced, the first error must be added prior to navigation within the table. Thus, if the LSB of the syndrome s is 0, an XOR with g must be performed to force the first position. Since this forced bit is implicit and not part of the syndrome, a right shift is performed in both cases. Note that in practice, such operations can also be stored in the table. Adding two columns to store these results would double the memory storage needed but avoid any arithmetic operation.

Step 8: At this point, we consider the first error at position 0, and perform the loop over all possible forced positions, from 0 to $m+n-1$.

Step 9-10: The relative single error position is accessed from table T. Two conditions must be met to consider the candidate as valid. First, the position P_1 must indicate a relative distance (i.e., the single error position for a syndrome value equal to the index), as tested in step 10, and then the position of the error indicated by P_1 must be in the range of the packet.

Step 11: As the distance P_1 is relative to the current forced position F_1, $P_1 < m+n$ is not enough to guarantee that the error pattern is possible. Thus, considering a relative packet size of $m+n-F_1$, we ensure that we identify only valid error patterns.

Step 12: When both conditions described in Step 10 and Step 11 are met, a candidate comprising the forced position F_1 and the remaining error position P_1+1+F_1 is appended to the list.

Algorithm 2 TwoErrorCorrection($T[2^n][2]$,s,n,m,$cycle$)

Inputs:

$T[2^n][2]$: indexed table containing P_1 and *next* elements for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial

Output:

E_2: the list of valid error positions for double-bit error

```
1:  E2 ← {}
2:  Let s' be a vector of length n
3:  if s ∧ 1 = 0 then
4:      s' ← [s ⊕ g] ≫ 1 // Can be stored in a table
5:  else
6:      s' ← s ≫ 1 // Can be stored in a table
7:  end if
8:  for F1 = 0 to m + n − 1 do
9:      P1 ← T[s'][0]
10:     if P1 ≠ −1 then
11:         while (P1 < m + n − F1) do
12:             Add (P1 + 1 + F1, F1) to E2
13:             P1 ← P1 + cycle
14:         end while
15:     end if
16:     s' ← T[s'][1]
17: end for
18: Return E2
```

Step 13: As the single error position is cyclic, we should test every possible error pattern. At each loop, we add the *cycle* length to the position P_1 and check if the resulting value is within the range of the packet. If not, the next forced position must be tested.

Step 16: Once a forced position has been processed, the updated syndrome, corresponding to the syndrome resulting from the next forced position is accessed from the table. It corresponds to the second column of the current syndrome index, as illustrated in Table 3.

C. N ERROR CORRECTION

For N errors, the generalization of the strategy used for double error correction is performed, as illustrated in Algorithm 3. The concept is as follows. $(N-2)$ bit values must be forced to 1 at each step (initialized to the $(N-2)$ LSBs of the syndrome). Then, the double error method is performed on the remaining bits of the packet. The number of errors, N, is then decreased and when it reaches 1, single error correction is performed on the computed syndrome.

For each forced bit error position (in the double error approach after forcing the $(N-2)$ bits), the *next* element (forcing the next bit toward the MSB) can be accessed from the reference table to reduce the computational complexity

TABLE 3. Table generated for double-bit error correction for a CRC-5 of generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$.

Index	P_1	$next$	Index	P_1	$next$
0	-1	23	16	4	31
1	0	13	17	-1	5
2	1	22	18	-1	30
3	-1	12	19	-1	4
4	2	21	20	-1	29
5	-1	15	21	5	7
6	-1	20	22	8	28
7	10	14	23	-1	6
8	3	19	24	-1	27
9	-1	9	25	9	1
10	-1	18	26	14	26
11	7	8	27	-1	0
12	-1	17	28	12	25
13	13	11	29	-1	3
14	11	16	30	-1	24
15	-1	10	31	6	2

and avoid arithmetic operations. Therefore, we look at every possible combination of forcing $(N-2)$ bit values to 1 within the $m-1$ first bits (LSB) of the packet. Let the forced bits be at positions $F_1, F_2, \ldots, F_{N-2}$ with $F_1 < F_2 < \ldots < F_{N-2}$ (sorted by increasing the bit position). Forcing some bits to 1 means that the bits at positions $F_1, F_2, \ldots, F_{N-2}$ are set to 1 and the other bit positions below position F_{N-2} are set to 0. With this definition, it can be noted that all the bits with a position below F_{N-2} are actually forced (to a value of 0 or 1). Steps 4 to 12 of Algorithm 3 illustrate this forcing process.

Steps 1-2: First, we initialize both the candidate list to an empty set, and k, the local number of errors to consider, to its maximum value N.

Step 3: This step and the following ones are repeated for every value of k for which $N \geq k > 2$.

Step 4-5: At step 4, the set of forced bits $\mathcal{F}$ is initialized to the $(k-1)$ LSB positions. Step 5 shows that the process continues until $\mathcal{F}$ is set to the $(m+n)$ MSB positions. In step 8, & and || represent the *logical and* and *logical or* operations, respectively. In this approach, "forced bits" means the positions forced to 1, representing the $(N-2)$ errors placed. However, it should be understood that if a bit within the range of forced bit positions is not forced to 1, then it is forced to 0. More generally, we focus on the positions forced to 1 because the algorithm strives to set the other positions to 0 during its elimination process.

Step 6: We start by initializing the binary vector s', representing the updated syndrome, to s.

Steps 7-11: Forcing of bit positions must be achieved through the successive addition of the generator polynomial vector and an accumulation in the updated syndrome s' in order to obtain the desired result.

The syndrome cannot simply be ignored and the bit values changed at desired positions. Rather, the equivalence relationship with the original syndrome must be maintained, i.e., only shifted versions of g may be added to it. This is done by starting at bit 0 of the syndrome, and if its value is the desired value, then nothing is done for that position. Otherwise, g must be added at that position (i.e., XOR performed), which contains 1 at its LSB, to modify it. The decision as to whether or not to perform the XOR is illustrated in step 8 of Algorithm 3.

As the method successively processes the next positions from LSB (bit 0) to MSB (bit F_{N-2}) in a similar fashion, it is important that g be added at each step at the current position (i.e., g should be XORed at step 9 when processing position i) and the syndrome value when the conditions of step 8 are met until position F_{N-2} is processed. Otherwise, the syndrome is right-shifted by one position at each step, as shown in step 11.

In practice, these operations can also be stored in a table when searching for N error patterns. Thus, two additional columns would be added, doubling the required storage, but every computation would be stored in the table. Only the

Algorithm 3 NErrorCorrection($T[2^n][2]$,s,n,m,$cycle$,N)

Inputs:

$T[2^n][2]$: indexed table containing P_1 and $next$ elements for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial
N: the maximum number of errors to consider

Output:

E_N: the list of valid error patterns up to N errors

1: $E_N \leftarrow \{\}$
2: $k \leftarrow N$
3: **while** $k > 2$ **do**
4: $\quad \mathcal{F} \leftarrow (0, \ldots, k-2)$
5: $\quad$ **while** $\mathcal{F} \neq (m+n-k+1, \ldots, m+n-1)$ **do**
6: $\quad\quad s' \leftarrow s$
7: $\quad\quad$ **for** $i = 0$ to F_{k-2} **do**
8: $\quad\quad\quad$ **if** $(s'_1 = 0$ & $i \in \mathcal{F})$ || $(s'_1 = 1$ & $i \notin \mathcal{F})$ **then**
9: $\quad\quad\quad\quad s' \leftarrow (s' \oplus g) \gg 1$ // Can be stored in a table
10: $\quad\quad\quad$ **else**
11: $\quad\quad\quad\quad s' \leftarrow (s' \gg 1)$ // Can be stored in a table
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad\quad m' \leftarrow m - (F_{k-2} + 1)$
15: $\quad\quad$ Add TwoErrorCorrection(T,s',n,m',$cycle$), $\mathcal{F}$ to E_N
16: $\quad\quad \mathcal{F} \leftarrow$UpdateForcedPosition($\mathcal{F}$,$m$)
17: $\quad$ **end while**
18: $\quad k \leftarrow k-1$
19: **end while**
20: Add TwoErrorCorrection(T,s,n,m,$cycle$), $\mathcal{F}$ to E_N
21: Add SingleErrorCorrection($T[][0]$,s,n,m,$cycle$) to E_N
22: Return E_N

forced bit positions set $\mathcal{F}$ and the current position i are needed to perform the full candidate list generation.

From there, since all the forced bit positions have been properly set, the process described for double error handling is performed on the remaining length of the packet. The following are the remaining steps:

Step 14: We introduce m', which corresponds to the remaining length of the packet. It is crucial to take into account the fact that the double-bit error correction must be performed with the current position taken into account to ensure the whole set of possibilities are considered.

Step 15: We perform a double-bit error correction on the updated syndrome s', comprising the $(N-2)$ LSB forced positions, for a packet length m'. The candidate error patterns comprise the 2 positions returned by Algorithm 2 and the forced positions contained in $\mathcal{F}$.

Step 16: Once the double error correction has been performed, we must update the forced error position, as illustrated in Algorithm 4, to test all possible $(N-2)$ forced positions.

Step 18: The main loop is performed for every number of errors k from N to 3 (i.e., when the error must be forced). When k reaches a value of 2, the last steps to perform are a double-bit error correction, followed by a single-bit error correction on the original syndrome s and payload length m, as shown in steps 20 and 21. We return the completed list of candidates E_N at step 22.

Algorithm 4 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 $\text{len}(\mathcal{F})$ 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} ← F_{k-1} + 1
3:      Return F' ← (F_1, ..., 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 ← F_i + 1
8:              j ← i
9:              while j < k − 1 do
10:                 F_{j+1} ← F_j + 1
11:                 j ← j + 1
12:             end while
13:             Return F' ← (F_1, ..., F_{k-1})
14:         end if
15:     end for
16: end if
```

D. CYCLES OF next ELEMENTS

In this subsection, we study some properties of the generated lookup table. Although this knowledge does not impact the functionality of the proposed algorithms, it provides valuable insights into the nature of the solutions to expect, and that could be further exploited. Although more complex, the approach has similarities with periodic sequence and shift registers (linear and non-linear feedback shift registers) over Galois Fields [27], which have been studied in the literature [28]–[30]. We can see in the flowchart of Figure 4 that there are two distinct variables for determining the syndromes, namely, a and b. In the last subsection, a was described as the main loop syndrome and b as the local loop syndrome. While a was incremented by 1 at each main loop, the syndrome value of b was successively updated to its *next* element until the *next* element had been processed (i.e., all the syndromes in the current cycle had been added to the table). Such cycles are observed for every generator polynomial. Figure 6 presents an example of this cycle. In the figure, we present the *next* element cycles for a generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$. It can be seen that since the generator polynomial parity is even, the sets of syndromes resulting from an odd or an even number of errors are disjoint. The value of each syndrome is expressed as a decimal value in Figure 6, where it can be seen that most parts of the table will be completed once these two cycles (i.e., two local loops on b) are performed.

A straightforward design to generate the table would consider the local odd and even loops only. However, we observed two exceptions to these cycles. In fact, two syndrome values are out of the cycle loops, and are represented by red circles in Figure 6. For the considered generator polynomial, these two syndrome values correspond to $s = 9$ and $s = 26$, respectively corresponding to $s = [01001]$ and $s = [11010]$ in binary vector representation, from MSB to LSB. We present the computation of the *next* element for both cases in Figure 7, where they are referred to as self-loop syndromes, types I and II. It can be seen that in both cases, the *next* element corresponds to the element itself. To understand why the *next* element is the syndrome itself and to make sure that there is no other case than these two, we performed an analysis of such syndromes.

For the case presented in Figure 7a, applied to even parity polynomials, and based on the operations needed to obtain the *next* element, it can be seen that:

$$\begin{aligned} &(s \ll 2) \oplus (1 \ll 1) = (s \ll 1) \oplus 1 \oplus g \\ \Longrightarrow\ &(s \ll 2) \oplus (s \ll 1) = g \oplus (1 \ll 1) \oplus 1. \end{aligned} \tag{6}$$

This equation can be expressed as:

$$s_{i-1} = g_{i+1} \oplus s_i \quad \forall\, (n-1) \geq i \geq 1 \tag{7}$$

with $s_{n-1} = 0$. Because $g(x)$ has even parity, it can be shown that $s_0 \neq g_1$.

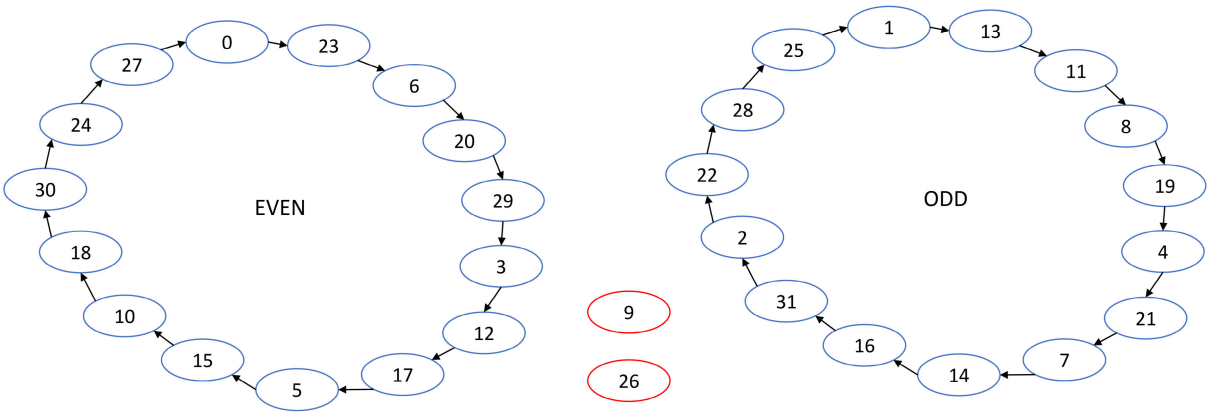


FIGURE 6. Example of cycles and exceptions when the generator polynomial is $g(x) = x^5 + x^4 + x^2 + 1$.

This equation can be applied to the generator polynomial to retrieve the even exception of this $g(x)$:

$$s_4 = 0, \quad s_3 = g_5 \oplus s_4 = 1, \; s_2 = g_4 \oplus s_3 = 0$$
$$s_1 = g_3 \oplus s_2 = 0, \quad s_0 = g_2 \oplus s_1 = 1 \tag{8}$$

At the end of the process, the only exception of type I is $s = [01001] = 9$, which corresponds to the one identified in Figure 6.

The other exception can be expressed in the same way, using the operations described in Figure 7b:

$$(s \ll 2) \oplus (1 \ll 1) = (s \ll 1) \oplus 1 \oplus g \oplus (g \ll 1)$$
$$\implies (s \ll 2) \oplus (s \ll 1) = (g \ll 1) \oplus g \oplus (1 \ll 1) \oplus 1 \tag{9}$$

which can be expressed as:

$$s_{i-1} = g_{i+1} \oplus g_i \oplus s_i \quad \forall\, (n-1) \geq i \geq 1 \tag{10}$$

with $s_{n-1} = 1$. Because $g(x)$ has even parity, it can be shown that $s_0 = g_1$. By applying this equation to the considered generator polynomial, the following is obtained:

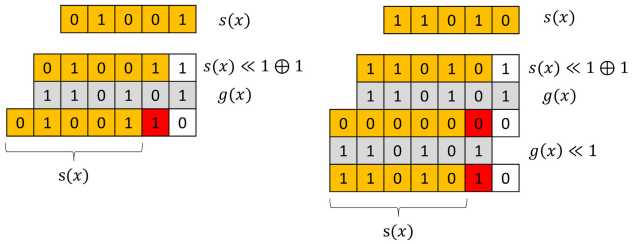
$$s_4 = 1, \quad s_3 = g_5 \oplus g_4 \oplus s_4 = 1, \; s_2 = g_4 \oplus g_3 \oplus s_3 = 0$$
$$s_1 = g_3 \oplus g_2 \oplus s_2 = 1, \quad s_0 = g_2 \oplus g_1 \oplus s_1 = 0 \tag{11}$$

At the end of the process, the only exception of type II is $s = [11010] = 26$, which also corresponds to the one identified in Figure 6.

We will now study the case of odd parity generator polynomials. Since the addition of such polynomials to a syndrome changes the parity, it can be seen in Figure 7a that the next element of a syndrome $s(x)$ cannot possibly be $s(x)$ itself. However, since the generator polynomial is added twice in 7b, it can be shown that the solution is:

$$s_i = g_{i+1} \quad \forall\, (n-1) \geq i \geq 0 \tag{12}$$

The solution can also be expressed as $s = (g \gg 1)$. Therefore, only type II self-loops exist for odd parity generator polynomials.



(a) Self-loop type I (b) Self-loop type II

FIGURE 7. Representation of the self-loop *next* elements for the two syndromes exceptions in Figure 6.

Studying these elements is important because doing so provides insight into the nature of the solutions to be expected. The *next* elements have the potential to generate numerous candidates. Let us assume that $N - 1$ bit positions have been forced and that the syndrome is one of these two *next* elements. If the associated P_1 leads to the remaining bit error being at position k within the packet, then the same $N - 1$ forced bits along with any position $k + i < m + n$ with $i \geq 1$ also constitute a valid error pattern. For example, this *next* element in Table 3 is at index 26. For this syndrome value, the single error position P_1 is 14, and the *next* element itself is 26. If after forcing $(N - 1)$ errors, we obtain this syndrome and the remaining length is 50 bits, we get a first candidate error pattern, with errors at a forced position and at $(F_{N-1}+1+P_1)$. In the very next step, the update syndrome is 26 once again. As the remaining length is now 49 bits, another candidate is found, with forced position F_{N-1} and position P_1 both increased by one. At each step, a new candidate is appended to the list until reaching the end of the message.

E. SYNDROMES WITH NO SOLUTION FOR SINGLE ERROR

We also observed an exception in the single error position search. Given the parity of both the generator polynomial and the syndrome, we are able to determine whether the number of errors that occurred in the packet is odd or even. Based on this knowledge, a particular entry of the table represented

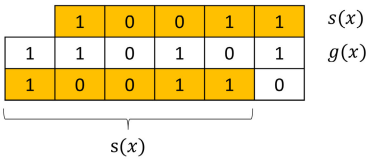


FIGURE 8. Single error correction method performed on syndrome $s(x) = x^4 + x + 1$ using a generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$ (even parity).

in Table 3 is worthy of interest. The syndrome $s = 19 = [10011]$ has an odd number of non-null coefficients, and is thus expected to have an associated single error position. However, Table 3 shows that there is no single error position for this syndrome, for any possible packet length. Figure 8 illustrates the single error correction applied to the syndrome. It can be clearly seen that at each step, the resulting syndrome corresponds to the originally computed one, and therefore, there is no possibility of having only one non-null coefficient at any step of the process.

Similarly to the *next* element cycles described previously, it can be seen that for this exception to be realized, the following equality must be met for an even parity generator polynomial:

$$(s \ll 1) = s \oplus g \implies (s \ll 1) \oplus s = g \tag{13}$$

which can be expressed as:

$$s_{i-1} = g_i \oplus s_i \quad \forall\, (n-1) \geq i \geq 1 \tag{14}$$

with $s_{n-1} = 1$. It can easily be shown that $s_0 = 1$ for even parity generator polynomials, and that the syndrome therefore never contains a single error pattern. Again, applying this equation to the generator polynomial used here yields:

$$\begin{aligned} &s_4 = 1, \quad s_3 = g_4 \oplus s_4 = 0, \; s_2 = g_3 \oplus s_3 = 0 \\ &s_1 = g_2 \oplus s_2 = 1, \quad s_0 = g_1 \oplus s_1 = 1 \end{aligned} \tag{15}$$

At the end of the process, the syndrome is $s = [10011] = 19$, which is the one identified in Table 3.

For odd parity generator polynomials, syndromes with no solution must belong to a cycle comprising an even number of syndromes, where an even parity syndrome leads to an odd parity syndrome after the addition of the generator polynomial, and vice versa. Let us now investigate the conditions for having a pair of syndromes with no solution (i.e., the shortest such cycles). Let s and s' be these two syndromes. They must meet the following expression:

$$\begin{aligned} &(s' \ll 1) \oplus s = g \text{ and } (s \ll 1) \oplus s' = g \\ &\qquad \implies (s' \oplus s) \ll 1 = s \oplus s' \end{aligned} \tag{16}$$

which can be expressed as:

$$s_{i-1} \oplus s'_{i-1} = s_i \oplus s'_i \quad \forall\, (n-1) \geq i \geq 1 \tag{17}$$

with $s_{n-1} = s'_{n-1}$. It follows that $s_i = s'_i, \forall i$, and therefore, there is no cycle of two elements, one leading to the other, for odd parity generator polynomials. Further investigation is required to conclude on the existence of longer cycles.

TABLE 4. Value of syndrome exceptions for commonly used generator polynomials (CRC-8-CCITT, CRC-16-CCITT, CRC-24-BLE and CRC-32-Ethernet).

	Self-loop (type I)	Self-loop (type II)	No single error
CRC-8	126	131	253
CRC-16	30,735	34,832	61,471
CRC-24	8,388,324	8,389,421	16,776,649
CRC-32	_____	2,187,366,107	_____

In Table 4, we present the decimal values of the syndrome exceptions for different generator polynomials. In it, the syndrome whose *next* element is itself is denoted as "Self-loop." As has been demonstrated, there are two such types of elements. A syndrome that does not provide a single error candidate is denoted "No single error". Since CRC-32-Ethernet uses an odd parity generator polynomial, there is no such syndrome identified, and it exhibits a type II self-loop.

Having this knowledge on the structure of cycles and exceptions in CRC error correction can help save computations if such a syndrome is spotted at the receiver. By identifying an exception, we can avoid unnecessary computations while ensuring the exhaustive list of error patterns is obtained.

IV. PERFORMANCE AND COMPLEXITY

In this section, we compare different performance aspects of the proposed CRC-based error correction method using an optimized table (which we will refer to as **CRC-ECOT**) to several other CRC-based error correction methods, listed next:

- **Arithmetic operations (CRC-ECA)**: the method described in [18], which generates the candidate list using logical operations on-the-fly, and does not require storing a table.
- **Explicit lookup table (CRC-ECEXP)**: the traditional lookup table approach [13], which is based on storing syndromes and their associated error positions. ECEXP explicit) means that the error positions are explicitly inserted in the lookup table. Note that this lookup table approach was recently used in a novel Polar SCL Decoding method [16].
- **Implicit lookup table (CRC-ECIMP)**: a proposed design of a lookup table approach similar to CRC-ECEXP, but where the error positions are not added to the table. The values of the error positions correspond to an implicit index (specific order in which the error positions are scanned) of the associated syndrome, which reduces the memory needed to implement such tables.
- **Exhaustive search (CRC-ECES)**: this corresponds to the arithmetic brute force scheme. No table is required in this method, but all the possible combinations of N error positions are successively tested to determine which ones lead to the computed syndrome.

Note that for our tests, we implemented the EC-ECOT version, as proposed in Algorithm 2, where the table size is smaller since steps 9 and 11 are not included.

A. COMPUTATIONAL COMPLEXITY

The methods compared have the same ability to correct multiple errors using the CRC syndrome. In terms of computational complexity, the CRC-ECOT method requires fewer operations as most of the computations are performed offline and then integrated into the table. We propose comparing the complexity of the method to that of the CRC-ECA and CRC-ECES approaches. Upon reception of a corrupted packet of m bits, the CRC-ECES method would test every error pattern up to N errors, i.e., it flips the error positions of the pattern and then computes the syndrome over the reconstructed packet. If the syndrome is null, a valid error pattern is found.

The complexity in this section is expressed as the number of additions to perform with g. In the case of CRC-ECES, m such operations are required for each long division. Thus, for the search of a single error, there are m long divisions to perform in order to test every possible error position, yielding m^2 operations. By extending this process to N errors, we obtain a global complexity of $O(m^{N+1})$.

The CRC-ECA method only performs one long division for a single error search, which can be performed through m additions of g. Extending this method to search for several errors requires setting $(N-1)$ forced positions, and a long division must be performed on the remaining length of the packet. A double-bit error search thus requires m^2 operations. Generalized to the search of N error patterns, it yields a global complexity of $O(m^N)$, as demonstrated in [18].

The complexity of the proposed CRC-ECOT method can be expressed through the complexity required to build the table and through the complexity required to perform the identification of the candidate error patterns. The former is highly complex, but is performed offline, prior to the communication. We will thus focus on the latter. For CRC-ECOT, the complexity up to double-bit error correction is extremely low thanks to the *next* element column integrated into the table. Single error correction requires a single lookup, while double error correction requires m table lookups. When the number of errors is increased, forced positions must be set. Thus, when searching for N errors, $(N-2)$ positions must be tested and the global complexity of the proposed approach will be $O(m^{N-2})$ additions of g times m table lookups. These gains are significant since the complexity increases significantly as N increases. Note that we could completely eliminate the arithmetic operations at the expense of a larger lookup table by storing $[s' \oplus g] \gg 1$ and $s' \gg 1$ for all syndrome values[3] in steps 9 and 11 of Algorithm 3. This would allow to further increase the speed for $N > 2$ by a new global complexity of $O(m^{N-1})$ table lookups.

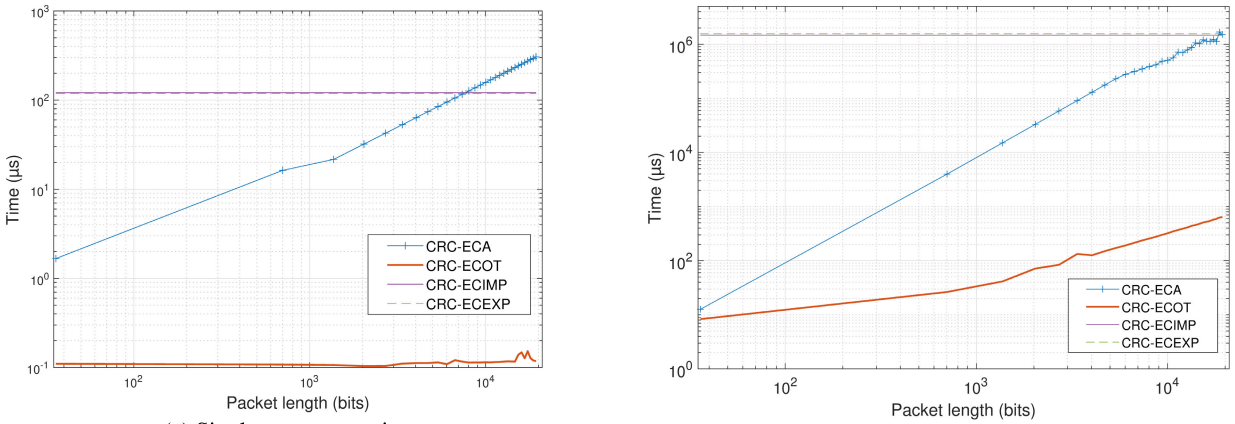
We tested a C implementation of the CRC-ECA, the CRC-ECOT and both the CRC-ECEXP and CRC-EXIMP approaches to compare their processing speeds on a Raspberry Pi model 4 [31], with a Broadcom BCM2711 processor, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz and 8 GB RAM. The test results are available in Figure 9, with log-log scales.

First, it can be seen that when searching for single errors, as in Figure 9a, the processing time per symbol of the CRC-ECOT is independent of the length of the packet. In fact, as the table is syndrome-indexed, when a corrupted packet is received, we only need to read the content of the table at the computed syndrome's entry, regardless of the packet's length.

On the other hand, the CRC-ECEXP and CRC-ECIMP lookup table-based methods must scan the packet in order to search for potential single error positions, which leads to higher processing times as the maximum packet length increases. In our example, the maximum packet length considered is 2500 bytes. The processing time per syndrome on the tested CPU for the proposed CRC-ECOT method is 100 ns, on average, whereas the CRC-ECA method's processing time ranges from 1.7 μs for the smallest payload (36 bits) to 300 μs for the largest payload considered. Both lookup table-based methods offer a constant processing time since they must scan the whole table for any computed syndrome. This processing time is 120 μs on the tested architecture. The proposed method's speedup for single error correction is thus 1200 times, as compared to these table approaches, and ranges from 17 to 3000 times faster, as compared to CRC-ECA, depending on the packet size.

The double error correction case illustrated in Figure 9b shows that again, CRC-ECOT processes each syndrome much faster than do all the other tested methods. Here, it can be seen that the processing time is not constant for CRC-ECOT as the packet length increases. For double error correction, we must consider the *next* element of each syndrome for the whole length of the packet. The processing time thus depends on the packet length. It is also interesting to note that as the packet length increases, the processing time gains also increase. When considering the smallest packet length (36 bits), the average processing time for double error correction is 8.3 μs for the proposed method and 12.6 μs for the arithmetic method, which gives a time ratio of 1.5 between the two methods. This ratio increases with the packet length, and is ultimately very significant for the largest payloads. The proposed method requires 642 μs to process a syndrome while the arithmetic method needs 1.5 s, yielding a time ratio of about 2300. Of course, these gains are due to the design of the proposed method as most of the computation is performed offline, prior to communication. CRC-ECIMP and CRC-ECEXP processing times for double-bit error correction are still constant at 1.5 s. The proposed method's speedup goes from 174,000 for the smallest packets to 2300 for the largest ones, as compared to these table approaches,

[3] Note that depending on the architecture on which the algorithm is implemented and the generator polynomial of interest, it may be less complex to perform $s' \gg 1$ than to retrieve it from a table.



(a) Single error correction (b) Double error correction

FIGURE 9. Comparison of the average processing time per syndrome for single and double error corrections for state-of-the-art CRC-based error correction [18] (CRC-ECA), the proposed optimized table (CRC-ECOT), the implicit table (CRC-ECIMP) and explicit table [13] (CRC-ECEXP) applied to CRC-16-CCITT.

depending on the packet size. Note that these speedups can be further increased by storing the values of steps 9 and 11 of Algorithm 3 in the table, but at the cost of increased memory requirements.

B. MEMORY REQUIREMENTS

We also compared the memory required to store the CRC-ECOT and CRC-ECEXP tables in Table 5, when using a packet of 1500 bytes, as it is the largest payload available in Ethernet (MTU). We compared these approaches for various generator polynomials:

- CRC-8-CCITT, where $g(x) = x^8 + x^2 + x + 1$.
- CRC-16-CCITT, used to protect the headers of 802.11 [3] and in low consumption 802.15.4 [32] communications, where $g(x) = x^{16} + x^{12} + x^5 + 1$.
- CRC-24-BLE, used to protect Bluetooth Low Energy [33] packets, where $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$.
- CRC-32-Ethernet, used to protect the entire packet in Ethernet [4] protocol, where $g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$.

The CRC-ECEXP approach calls for storage of a syndrome along with the corresponding error positions for every possible error case, yielding the following memory requirement:

$$M_{\text{CRC-ECEXP}} = \binom{m}{N} \times [\text{length}(s) + (2 \times N)] \text{ bytes} \quad (18)$$

where m is the bit length of the payload, length(s) corresponds to the byte length of the syndrome, and N is the maximum number of errors. We considered positions to be integer variables of 2 bytes, since the maximum single position is 12,000 in this implementation.

We also propose a comparison with the CRC-ECIMP method, which is an implementation requiring less memory than CRC-ECEXP. The implicit table approach consists in indexing a syndrome with its associated error position, which reduces the size of the table:

$$M_{\text{CRC-ECIMP}} = \binom{m}{N} \times \text{length}(s) \text{ bytes} \quad (19)$$

This strategy calls for knowledge of error position management when considering several errors (e.g., syndrome at index 12,001 corresponds to a double error at positions (0, 1)).

The proposed CRC-ECOT approach requires listing, for all the syndromes, the *next* element, whose size is always of length(s) bytes, as well as the error position P_1. Note that considering that the table is initialized to (-1), a negative element, the number of bits required to store P_1 is $\log_2(cycle + 1) + 1$. As most *cycle* lengths are $(2^{n-1} - 1)$ as shown in Table 5, it would take n bits to store P_1. It can be seen that the *cycle* length for CRC-32 is $2^n - 1$, yielding the need for an additional bit, for a total of $(n + 1)$ bits to store P_1 (i.e., 33 bits[4] for CRC-32). Except for such cases, the memory required can thus be expressed as:

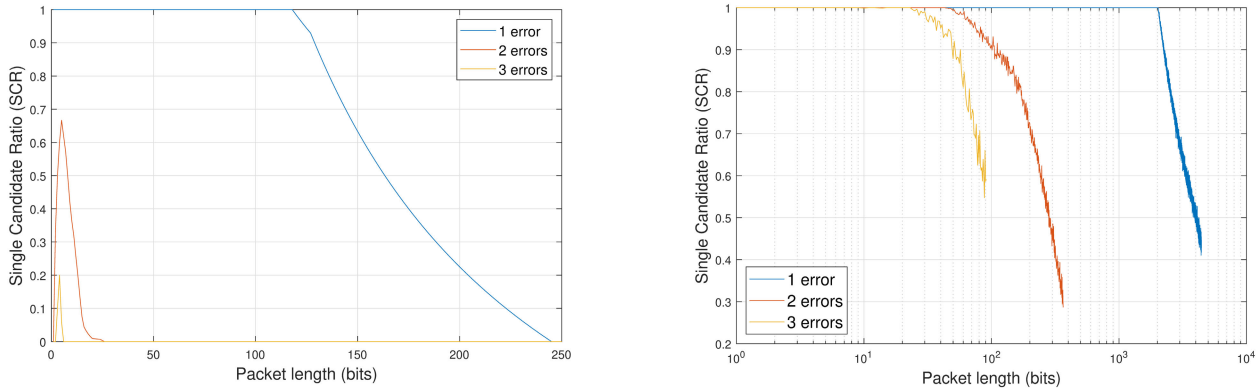
$$M_{\text{CRC-ECOT}} = \begin{cases} 2^n \times \text{length}(s) \text{ bytes}, & \text{if } N = 1 \\ 2^n \times 4 \times \text{length}(s) \text{ bytes}, & \text{if } N > 1 \end{cases} \quad (20)$$

It can first be seen in Table 5 that, unlike in CRC-ECEXP, where the table size is a function of the number of errors considered, the proposed approach has a fixed length for $(N > 1)$. In fact, for single error correction, the column comprising the *next* elements is not needed, and with double error correction, the columns can store both the *next* element and the results of steps 9 and 11 of Algorithm 3, which multiply the storage needed by 4. The table size for the proposed approach is less than for CRC-ECEXP, for all generator polynomial lengths n when $(N > 2)$. The proposed table is also smaller for $N = 2$ up to $n = 24$, and up to $n = 8$ for $N = 1$, thanks to the way in which cycles are handled in our method. It should be recalled that the arithmetic method does

[4]Note that some architectures will not support 33-bit numbers efficiently, and will consider 64 bits when exceeding $2^{32} - 1$, which would double the memory required.

TABLE 5. Comparison of the memory required for the tables in the CRC-ECEXP [13] and the proposed CRC-ECOT approaches. The packet considered here has a length of 1500 bytes. Note that the arithmetic method is table-free, and thus, its memory requirements remain negligible for any case described in this table.

Nb Errors N	CRC-8-CCITT (cycle = 2^7-1)		CRC-16-CCITT (cycle = $2^{15}-1$)		CRC-24-BLE (cycle = $2^{23}-1$)		CRC-32-Ethernet (cycle = $2^{32}-1$)	
	Lookup Table	Proposed	Lookup Table	Proposed	Lookup Table	Proposed	Lookup Table	Proposed
1	36 kB	256 B	48 kB	131 kB	60 kB	50.5 MB	72 kB	17.7 GB
2	360 MB	1 kB	432 MB	524 kB	504 MB	202 MB	576 MB	69.3 GB
3	2.02 TB	1 kB	2.30 TB	524 kB	2.60 TB	202 MB	2.88 TB	69.3 GB
4	7.77 PB	1 kB	8.64 PB	524 kB	9.50 PB	202 MB	10.4 PB	69.3 GB



(a) Single candidate ratio (SCR)

(b) Post-checksum single candidate ratio

FIGURE 10. Evolution of the SCR and post-checksum validation SCR for CRC-8-CCITT as a function of the packet length.

not need to store a table at all, but this is at the cost of much higher computational complexity. The best compromise will depend on the application and hardware on which the method is implemented. For instance, the proposed method is very appealing for small CRCs or for correcting a high number of errors, while the arithmetic method may be more appealing for single error correction when long CRCs (e.g., CRC-32) are used.

C. APPLICATION TO THE CORRECTION OF MULTIPLE ERRORS

The method proposed in this paper is able to output an exhaustive list of candidate error patterns having up to N errors. In practice, we assume that this method is applied in the context of relatively reliable communication channels, where corrupted packets tend to be mildly damaged. The authors in [34] demonstrated a realistic experimental scenario in which most damaged packets in a Bluetooth low energy (BLE) environment contained 3 errors or less. As our method is able to generate the list with a lower complexity, its use can be considered to increase the value of N to handle more error cases. However, the number of candidates increases significantly with N for a given packet length. In [18], we investigated the ratio of error patterns that would output a candidate list with a single entry, thus allowing the correction of the packet, over all possible error patterns for a given N, and termed this ratio the Single Candidate Ratio (SCR). We show that CRCs with low-degree generator polynomials never reach an SCR of 100% when considering multiple bit errors, and rapidly decrease as the packet length increases.

As an example, we propose to analyze the impact of a checksum cross-validation on the SCR for the generator polynomial used in CRC-8-CCITT. The SCR of this CRC is illustrated in Figure 10. It can be seen that when considering a single error, up to 127 bits (i.e., 2^7-1), the SCR is 100%, which illustrates the importance of the *cycle* length, as there is no list with multiple single error candidates for packets smaller than this *cycle* value. The SCR then rapidly decreases and reaches 0% for packets greater than 245 bits. When considering double- and triple-bit error patterns, the SCR is very low, even for the smallest packet values considered. It reaches 0% for lengths of 26 and 6 bits, for double and triple error patterns, respectively.

Implementing a checksum validation step, as in the UDP and TCP protocols, helps to significantly increase such ratios and achieve decent error correction rates on small packets, even when using CRC-8-CCITT. In Figure 10b, it can be seen that the SCR is noticeably higher for all the numbers of errors considered. The SCR remains at 100% when dealing with packet lengths of up to 41 bits and 11 bits for double and triple error patterns, respectively. Moreover, it can be seen that the SCR is still over 50% for double error patterns, up to a packet length of 270 bits. For single error correction, the SCR, which reaches 0% for packet sizes larger than 245 bits in Figure 10, remains at 100% up to a significant length of 2000 bits with checksum validation, which allows reconstruction of many more error cases. The method should perform even better with

an actual BLE system with small packets (up to 39 bytes, and now increased to a maximum of 255 bytes) protected by a strong CRC-24.

Thus, when increasing the number of errors, validation steps such as a checksum cross-validation will help maintain a high correction rate when the candidate list size increases significantly. This allows to take advantage of the processing speed gains of the proposed method and to consider larger values of N. Increasing N brings a lot of changes and challenges in the literature methods as they are designed for a specific number of errors. We demonstrated that table-based CRC-ECEXP and CRC-ECIMP produce intractable table sizes from $N = 3$. The complexity of CRC-ECA methods is greatly affected whenever N is increased. The proposed CRC-ECOT replaces the arithmetic operations with successive table lookups, thus reducing the global complexity. As well, the table size is easily managed and constant for $N \geq 2$, which makes it very appealing for multiple error correction.

V. CONCLUSION AND PERSPECTIVES

In this paper, we propose an optimized table-based method for performing multiple error correction based on the CRC syndrome. The approach offers a low complexity alternative to the state-of-the-art error correction method as it generates a table that contains precomputed operations required to perform error pattern searches, avoiding most to all arithmetic operations.

Thanks to offline table generation, the proposed approach achieves the same error correction performance as state-of-the-art approaches while providing computational savings and thus improving the processing speeds. We show through simulations that the proposed method achieves significant speed gains over the table-free arithmetic method, and is between 2300× and 3000× faster when generating the list of double and single error patterns, respectively.

Thus, reducing the complexity offers the possibility of increasing the number of errors to consider, while keeping the same processing time. Since this greatly increases the number of candidates in the output error pattern list, we present a validation step that increases the correction rate for lists containing many candidates. Other validation steps could be used such as those based on bit error probability. Future work will look at integrating the proposed CRC-based error correction solution into a complete cross-layer receiver architecture in order to benefit from other validation mechanisms available in the protocol stack. The objective is to further reduce the list size or to be able to determine the best candidate out of several in order to reconstruct the best signal quality at the receiver side (e.g., visual quality, in the case of video content transmission).

REFERENCES

[1] A. Houghton, *Error Coding for Engineers*. Boston, MA, USA: Springer, 2001.
[2] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 459–468.
[3] *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Standard 802.11, Dec. 2016.
[4] IEEE Standard Association, *IEEE Standard for Ethernet*, IEEE Standard 802.3-2018, 2018. [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html
[5] J. Luo, K. D. Bowers, A. Oprea, and L. Xu, "Efficient software implementations of large finite fields GF(2^n) for secure storage applications," *ACM Trans. Storage*, vol. 8, no. 1, p. 27, Feb. 2012.
[6] S. Lin and D. J. Costello, *Error Control Coding*. Upper Saddle River, NJ, USA: Prentice-Hall, 1983.
[7] W. W. Peterson and W. E. Weldon, *Error-Correcting Codes*, 2nd ed. Cambridge, MA, USA: MIT Press, 1972.
[8] J. E. Meggitt, "Error correcting codes and their implementation for data transmission systems," *IRE Trans. Inf. Theory*, vol. 7, no. 4, pp. 234–244, Oct. 1961.
[9] I. S. Reed, "A class of multiple-error-correcting codes and the decoding scheme," *IRE Trans.*, vol. 4, pp. 38–49, Sep. 1954.
[10] D. E. Müller, "Application of Boolean algebra to switching circuit design and to error detection," *Trans. I.R.E. Prof. Group Electron. Comput.*, vol. EC-3, no. 3, pp. 6–12, Sep. 1954.
[11] J. L. Massey, *Threshold Decoding*. Cambridge, MA, USA: MIT Press, 1963.
[12] T. Kasami, S. Lin, and W. W. Peterson, "Some results on cyclic codes which are invariant under the affine group and their applications," *Inf. Control*, vol. 11, nos. 5–6, pp. 475–496, Nov. 1967.
[13] S. Shukla and N. W. Bergmann, "Single bit error correction implementation in CRC-16 on FPGA," in *Proc. IEEE Field-Program. Technol.*, Brisbane, NSW, Australia, Dec. 2004, pp. 319–322.
[14] 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.
[15] 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.
[16] 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, 2019.
[17] V. Boussard, "CRC-based error correction methods and algorithms applied to video communications over vehicular and IoT wireless networks," Ph.D. dissertation, Université Polytechnique Hauts-de-France, Valenciennes, France, 2021.
[18] V. Boussard, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Table-free multiple bit-error correction using the CRC syndrome," *IEEE Access*, vol. 8, pp. 102357–102372, 2020.
[19] D. Hachenberger and D. Jungnickel, "Basis representation and arithmetics," in *Topics in Galois Fields. Algorithms and Computation in Mathematics*, vol. 29. Cham, Switzerland: Springer, 2020, pp. 355–425.
[20] 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.
[21] V. Boussard, F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Robust H.264 video decoding using CRC-based single error correction and non-desynchronizing bits validation," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2020, pp. 1098–1102.
[22] V. Boussard, S. Coulombe, F.-X. Coudoux, and P. Corlay, "Enhanced CRC-based correction of multiple errors with candidate validation," *Signal Process., Image Commun.*, vol. 99, Nov. 2021, Art. no. 116475.
[23] V. Boussard, S. Coulombe, F.-X. Coudoux, P. Corlay, and A. Trioux, "CRC-based multi-error correction of H.265 encoded videos in wireless communications," in *Proc. Int. Conf. Vis. Commun. Image Process. (VCIP)*, Dec. 2021, pp. 1–5.
[24] Y. Zhang and Q. Yuan, "A multiple bits error correction method based on cyclic redundancy check codes," in *Proc. 9th Int. Conf. Signal Process.*, Oct. 2008, pp. 1808–1810.
[25] 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)*, Cairns, QLD, Australia, Dec. 2018, pp. 1–7.
[26] A. M. Demirtas, A. R. Reibman, and H. Jafarkhani, "Performance of H.264 with isolated bit error: Packet decode or discard?" in *Proc. 18th IEEE Int. Conf. Image Process.*, Sep. 2011, pp. 949–952.

[27] D. Hachenberger and D. Jungnickel, "Shift register sequences," in *Topics in Galois Fields. Algorithms and Computation in Mathematics*, vol. 29. Cham, Switzerland: Springer, 2020.
[28] Y. Li, Z. Yang, K. Li, and L. Qu, "A new algorithm on the minimal rational fraction representation of feedback with carry shift registers," *Des., Codes Cryptogr.*, vol. 88, no. 3, pp. 533–552, Mar. 2020.
[29] S. K. Tripathi, B. Gupta, and K. K. S. Pandian, "The shortest register with non-linear update for generating a given finite or periodic sequence," *IEEE Commun. Lett.*, vol. 24, no. 6, pp. 1173–1177, Jun. 2020.
[30] U. Jetzek, *Galois Fields, Linear Feedback Shift Registers and Their Applications*. Munich, Germany: Carl Hanser Verlag GmbH & Company KG, 2018.
[31] *RaspberryPI4ModelB*. Accessed: Dec. 1, 2021. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[32] *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 Standard 802.15.4-2006, 2006.
[33] (2013). *Specification of the Bluetooth System. Core Version 4.1, Bluetooth SIG*. [Online]. Available: http://www.bluetooth.com.
[34] 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.

VIVIEN BOUSSARD (Member, IEEE) received the B.Sc. and M.Sc. degrees in broadcast engineering from the Université Polytechnique Hauts-de-France, Valenciennes, France, in 2015 and 2017, respectively, and the Ph.D. degree in engineering from the École de Technologie Supérieure, Montréal, Canada, and the Université Polytechnique Hauts-de-France, in 2021. In 2021, he joined Dacast, where his current research interest includes the optimization of video communication systems.

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. 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 a Senior Research Engineer and a Program Manager with the Audiovisual Systems Laboratory. In 2004, he joined the École de technologie supérieure (ÉTS), where he currently carries out research and development on video processing and systems, compression, and transcoding. From 2009 to 2018, he was the Vantrix Industrial Research Chair in video optimization. He is a Professor with the Department of Software and IT Engineering, ÉTS, a constituent of the Université du Québec Network.

FRANÇOIS-XAVIER COUDOUX (Senior Member, IEEE) received the M.S. and Ph.D. degrees in electrical engineering from the Université Polytechnique Hauts-de-France, Valenciennes, France, in 1991 and 1994, respectively. Since 2004, he has been a Professor with UMR 8520, Department of Opto-Acousto-Electronics, Institute of Electronics, Microelectronics, and Nanotechnologies, Valenciennes. 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 the Université Polytechnique Hauts-de-France, Valenciennes, France, in 1994. Since 2016, he has been a Professor with UMR 8520, Department of Opto-Acousto-Electronics, Institute of Electronics, Microelectronics, and Nanotechnologies, France. His current research interests include telecommunications, multimedia delivery over wired and wireless networks, and optimal quality of service for video transmission.

Erratum to "CRC-Based Correction of Multiple Errors Using an Optimized Lookup Table"

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

August 2023

In [1] three errors occurred:

- In Table 1., single-error correction is applied to CRC-8-SAE-J1850 of generator polynomial $g(x) = x^8 + x^4 + x^3 + x^2 + 1$ instead of CRC-8-CCITT of generator polynomial $g(x) = x^8 + x^2 + x + 1$.
- In section IV.B., the memory requirements for CRC-ECEXP and CRC-ECIMP are respectively

$$M_{\text{CRC-ECEXP}} = \binom{m+n}{N} \times [\text{length}(s) + 2 \times N] \text{ bytes} \tag{18}$$

$$M_{\text{CRC-ECIMP}} = \binom{m+n}{N} \times \text{length}(s) \text{ bytes} \tag{19}$$

  where $m + n$ is the bit-length of the packet.

- In Algorithm 2, line 8: $m + n - 1$ should be replaced by $m + n - 2$
- In Algorithm 2, line 11: $m + n - F1$ should be replaced by $m + n - F1 - 1$.

References

[1] Vivien Boussard, Stéphane Coulombe, François-Xavier Coudoux, and Patrick Corlay. CRC-Based Correction of Multiple Errors Using an Optimized Lookup Table. *IEEE Access*, 10:23931–23947, 2022.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Digital Object Identifier 10.1109/ACCESS.2019.DOI

CRC-Based Correction of Multiple Errors Using an Optimized Lookup Table

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: S. Coulombe (e-mail: stephane.coulombe@etsmtl.ca).

This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant and UPHF.

ABSTRACT

In this paper, we propose a new approach to perform multiple error correction in wireless communications over error-prone networks. It is based on the cyclic redundancy check syndrome, using an optimized lookup table that avoids performing arithmetic operations. This method is able to achieve the same correction performance as the state-of-the-art approaches while significantly reducing the computational complexity. The table is designed to allow multiple bit error correction simply by navigating within it. Its size is constant when considering more than two errors, which represents a tremendous advantage over earlier lookup table-based approaches. Simulation results of a C implementation performed on a Raspberry Pi 4 show that the proposed method is able to process single and double error corrections of large payloads in 100 ns and 642 µs, respectively, while it would take 300 µs and 1.5 s, respectively, with the state-of-the-art CRC multiple error correction technique. This represents a speedup of nearly 3000× for single error and 2300× for double error correction, respectively. Compared to table-based approaches, the proposed method offers a speedup of nearly 1200× for single error and 2300× for double error correction under the same conditions. We also show that when multiple candidate error patterns are present, numerous errors can be corrected by adding a checksum cross-validation step.

INDEX TERMS Cyclic Redundancy Check, Error Correction, Lookup Table, Checksum Control, Wireless Communication.

I. INTRODUCTION

In wireless communications, cyclic redundancy checks (CRCs) [1], [2] are widely adopted in order to enhance the communication reliability between a transmitter and a receiver. Indeed, CRCs are broadly used at different layers of the protocol stack of a data transmission as they detect transmission errors at the receiver. For instance, CRCs are present at the physical layer of widely deployed wireless protocols such as 802.11 [3] to protect the header of the packet, and at the Medium Access Control (MAC) layer, to protect the entire packet. An example of the latter is the 802.3 Ethernet protocol [4]. In general, CRCs are only used to detect whether a transmitted packet has bit errors, in which case the erroneous packet is discarded.

CRCs are computed from two main components: the protected bitstream, i.e., the message or data to transmit, which we will call the payload, denoted $d_T(x)$ for transmitted data, and a generator polynomial, denoted $g(x)$. Both $d_T(x)$ and $g(x)$ are binary polynomials. The generator polynomial is a constant binary polynomial, which is defined according to the transmission standard used. There are various generator polynomials in use, differentiated by their length and their number of non-null coefficients. Larger generator polynomials will lead to stronger error detection capabilities, but at the cost of higher packet overheads.

The typical transmission and reception of a CRC-protected data packet follow three main steps [1]. Firstly, at the transmitter, the payload $d_T(x)$ is left-shifted by n positions, where n is the degree of the generator polynomial, to produce the transmitted packet $p_T(x) = d_T(x).x^n$. This result is then divided by the generator polynomial $g(x)$ to obtain the remainder of this division, denoted $r_T(x)$. Secondly, $r_T(x)$ is

appended to the payload to be sent, and occupies the n right-most positions of the transmitted message. The transmitted message is thus $d_T(x).x^n + r_T(x)$, where $+$ is the addition which corresponds to an *exclusive or* (XOR) between two binary polynomials[1], and has a total length of $m + n$ bits, where m is the payload length of the message. Thirdly, at the receiver, the received packet, denoted p_R, is divided by the generator polynomial $g(x)$ in order to check if an error occurred during the transmission. If the message is intact, the remainder of the division by $g(x)$, called the computed CRC syndrome $s(x)$, will produce a result equal to zero since $r_T(x)$ has been added to generate an entire multiple of $g(x)$. On the other hand, a non-zero CRC syndrome indicates an error in the transmitted message.

In currently deployed systems and methods, conventional CRC error management involves discarding a packet when an error is detected at the receiver; that is, when the remainder is not equal to zero at the receiver. These approaches do not allow for packets to be corrected. In such systems, a received corrupted packet is processed as a lost packet. For example, a packet having a single bit in error, and which would otherwise be a good packet, will be detected as an erroneous packet and will be entirely discarded. Discarding entire packets having only one or a few errors leads to a significant loss of valuable information which could have been exploited if the errors had been corrected.

Therefore, various methods, described in Section II, were developed to exploit the CRC to perform error correction in a packet [6]–[16]. These methods include Meggitt decoding [8] and its practical variation called *error-trapping* which is capable of correcting, with high efficiency, single errors, short double errors and burst errors [9]–[12]. This is achieved using specialized circuitry. However, the approach cannot handle errors which are not close to each other in a packet. To avoid complex computational circuitry to identify the error positions, some approaches consist in storing the different syndromes produced by the error patterns in a lookup table [1]. This is typically performed for small packets and for a single error to maintain the size of these lookup tables reasonable. For instance, in [15], the authors propose a fixed latency serial transceiver with single bit error correction using a lookup table. However, when considering challenging applications such as video streaming, where packet length may be large and where multiple errors may occur in a same packet, this kind of solution becomes impracticable. In this paper, we propose an alternative and efficient method for CRC-based multiple error correction using an optimized lookup table. The proposed method is not limited to a specific packet size and is able to handle multiple errors. It is based on unpublished research work from [17]. In our method, most of the required operations are done offline and then stored in a table. The table is designed to allow the results for single and double error patterns to be accessed by simply navigating through the table, which results in significant processing time reductions versus table-free methods. The proposed approach can also be applied to identify any number of errors and at any position, with significant speedups. This paper does not intend to compete with existing error-correcting codes (e.g., turbo, LDPC, polar), but rather provides a new error-correction method that takes advantages of the widely used CRC present in the protocol stack. In fact, the proposed method can be used in addition to the codes mentioned above to reinforce the robustness of transmitted data against channel errors in a cross-layer context. The following are the benefits and contributions of the proposed method:

- **Significant speed gains**: The proposed approach is designed to allow most to all of the arithmetic operations required in the state-of-the-art table-free error correction approach [18] to be performed offline and stored in a table. This design contributes to greatly reducing the processing time of the proposed method.
- **Fixed-length tables**: State-of-the-art table-based approaches define distinct tables for each number of errors and maximum packet size considered. Furthermore, these tables grow in size exponentially with the number of errors considered, making them impractical when considering 3 or more errors. In contrast, the tables derived and used for the correction of multiple bits in the proposed approach are fixed for a given generator polynomial, and therefore possess a fixed length, regardless of the number of errors or packet size under consideration. We also propose another version of the table for the special case of a single error correction, which requires less memory storage.
- **Analysis of the syndromes and of the table structure**: We provide equations to identify, for any generator polynomial, syndromes with special properties (e.g., syndromes for which there is no valid error pattern comprising a single erroneous bit). We also highlight the cyclic properties of the table-based process which produces syndrome elements looping on themselves.
- **Applicability to multiple error correction**: Because of its large speed gains and reasonable table sizes, the proposed method is well-positioned to correct multiple errors. Also, unlike state-of-the-art methods [8]–[12] only handling errors occurring in bursts, the proposed method can handle multiple errors regardless of their individual positions in the packet. However, when multiple errors are considered, and especially when the burst and maximum packet size conditions are relaxed, the candidate error patterns leading to the computed syndrome are numerous, making the identification of the true error pattern challenging. We show that we can significantly reduce the list of candidates, even to the point of having a single one and being able to correct the packet, by performing an additional validation step in the form of testing of the candidates with the checksum found in UDP and TCP protocols.

[1]Binary packets of length m belong to the Galois Field GF(2^m), where the addition is performed as the bitwise XOR [5].

This paper is mainly based on the theory described and discussed in [18], with which the reader is expected to be familiar. The rest of the paper is organized as follows. In Section II, we introduce related works on CRC error correction covering lookup table methods and state-of-the-art CRC multiple error correction. In Section III, we present the concepts and implementation of the proposed optimized table, as well as its use in multiple error correction systems. We also analyze the structure of the table and syndromes with special properties. In Section IV, we evaluate the performance of the proposed approach in terms of processing speed and memory usage, as compared to existing methods. In Section V, we conclude and give an overview of future research works. We assume that the reader is familiar with the notations and concepts described in our previous works [18], especially those related to the Galois Field GF(2) [19] and its generalization to GF(2^m).

II. RELATED WORKS

Several works have explored the error correction possibilities of error detection codes, such as CRC [8]–[16] or checksums [20]. They can be categorized as table-based and table-free approaches. As described in [1] and more recently in [13], [14], [16], the table-based approach consists in computing a lookup table (LUT) prior to communication. In this scheme, each LUT entry contains the pre-computed syndrome corresponding to a specific single error position in the received packet. An example of such a table for CRC-8-CCITT is given in Table 1. A search for the computed syndrome is performed in the table. If a match is found, the bit at the corresponding position is flipped. The number of entries in the lookup table is based on the size of the expected payload, and is constant. It has been applied to other generator polynomials by [15]. Such tables have been recently exploited in improved successive cancellation list (SCL) decoding schemes of Polar codes [16]. This method is fast when conducted on small packets, but suffers from some serious disadvantages. Firstly, the approach assumes that a single error occurred in the packet, yielding a miscorrection probability in severe channel conditions, since a highly corrupted packet can produce the same syndrome as a single error. Furthermore, to support the correction of multiple errors, methods based on this approach must store the syndromes of all combinations of error positions. Consequently, memory requirements grow exponentially with the number of errors considered, limiting their use to small packet sizes and few errors in practice. For instance, as shown in [16], a table conceived to correct all single and double errors in packets having bit lengths of 128 bits requires 128 entries for single error positions and 8128 entries to cover all double error positions.

Table-free approaches rely on on-the-fly arithmetic operations instead of pre-computed lookup tables to perform error correction. They include Meggitt decoding [8] and error-trapping [9]–[12] for which multiple error correction is possible only when errors are concentrated in a region

TABLE 1: Example of lookup table for single error correction as proposed in the literature [13], applied to CRC8-SAE-J1850 of generator polynomial $g(x) = x^8+x^4+x^3+x^2+1$, considering a 10-bit payload.

Error position	Associated syndrome
0	0000 0001
1	0000 0010
2	0000 0100
3	0000 1000
4	0001 0000
5	0010 0000
6	0100 0000
7	1000 0000
8	0001 1101
9	0011 1010
10	0111 0100
11	1110 1000
12	1100 1101
13	1000 0111
14	0001 0011
15	0010 0110
16	0100 1100
17	1001 1000

of the packet not exceeding the CRC size, meaning that they cannot correct errors which are located far apart in the packet. In [18], we introduced a generic table-free multiple error correction. This approach generates an exhaustive list of all error patterns, regardless of where they are located in the packet, corresponding to the computed syndrome up to a predetermined (desired) number of errors. The data packet includes a payload $d_T(x)$ and cyclic redundancy check (CRC) information. The latter is calculated using a generator function or polynomial $g(x)$. The method generates an exhaustive list of valid error patterns containing N errors or less, by exploiting the definition of CRC computation. The list can comprise one or several candidates depending on $g(x)$, the CRC syndrome and the packet length. When the list contains several candidates, various methods can be applied to identify the error pattern within the list that actually occurred. For instance, in [21], we proposed using a UDP checksum and a non-desynchronizing bits validation steps to eliminate invalid candidates in the context of error-prone transmission of H.264 baseline profile compressed video streams. In [22], we combined a UDP checksum and video decoding validation steps for H.264 and H.265 video communications over 802.11p and Bluetooth Low Energy wireless networks. We demonstrated the possibility of correcting up to 3 errors in a video packet. In [23], we extended our work to estimate the number of candidates in the list and applied our method to correct up to 5 errors. All the approaches led to substantial video quality improvements. Another example of using additional information to perform CRC-based multiple bit error correction is found in [24] where the authors identify the bits having a high error probability to determine the most likely error patterns. This shows that although the proposed

method generates several candidate error patterns, depending on the targeted application, it is possible to eliminate all but one candidates using additional information present at other layers of the protocol stack or from the application itself.

We now summarize the main approach described in [18]. We can express the syndrome computed at the receiver as:

$$s(x) = (d_T(x).x^n + r_T(x) + e(x)) \bmod g(x) \quad (1)$$

where $e(x)$ represents the potential error pattern that corrupted the packet during the transmission, and where erroneous positions in $e(x)$ are identified by values of 1. From this definition, it is clear that the result of $(d_T(x).x^n + r_T(x)) \bmod g(x)$ is zero, as $r_T(x)$ is the remainder of the division of $d_T(x)$ by $g(x)$, and:

$$s(x) = e(x) \bmod g(x) \quad (2)$$

If we isolate the error vector, $e(x)$, we obtain:

$$e(x) = s(x) + q(x).g(x) \quad (3)$$

where $q(x)$ can be any binary polynomial of the highest degree $(m-1)$, with m being the payload length. As the total number of possible values of $q(x)$ is too high (there are 2^m such polynomials), the method proposes focusing on lightly corrupted packets and then building $q(x)$ term by term. This is achieved by canceling the Least Significant Bit (LSB) term of the current $e(x)$, at each step of the process, through the addition of properly left-shifted $g(x)$ (i.e., so that its LSB term is aligned with the term of $e(x)$ to cancel[2]). Adding $g(x)$ aligned at any position maintains the class equivalence of (3) and produces error pattern candidates having the same computed syndrome at each step [18]. The number of non-null coefficients in the polynomial $e(x)$ corresponds to the number of errors in the current candidate. The method thus only keeps candidates when the number of non-null coefficients in $e(x)$ is equal to or less than a predefined number N. To handle the correction of multiple bit errors, the method forces term values in $e(x)$ throughout the process (i.e., it sets such terms to 1 or turns them into a 1 by adding $g(x)$ aligned with their positions). Once $(N-1)$ positions have been forced, single error management is performed on the remaining length of the packet to locate the last error. If this last error does not exist, it means that the forced term values did not correspond to a valid error pattern for the given syndrome (i.e., an error pattern leading to the computed syndrome).

Although this method requires very little memory space, one drawback with it, however, is that it is very complex when considering several errors in a packet. This complexity, measured in the number of additions involving $g(x)$, is $O(m^{N-1})$, where m represents the payload length and N the number of errors considered. It thus grows exponentially with N.

[2]In this paper, we assume that $g_n = g_0 = 1$ as observed in practice.

III. PROPOSED METHOD

While the state-of-the-art method discussed in the last section aims at generating the list of valid error patterns with arithmetic operations, the proposed method does the same by exploiting tables, and accordingly avoids most arithmetic operations. Unlike previous methods [13], where tables are sparse and contain error positions only for CRC syndromes compatible with a certain packet size (e.g., that associated with a standard), the proposed tables provide error patterns for every possible syndrome value. In addition to being usable for any packet size, these tables can be indexed directly by syndrome value instead of being searched, as was the case in the previous case. Indeed, the process of searching for a pattern in a table, as in the previous approaches, is computationally intensive. This method thus provides enhanced speed performance, but at the cost of higher memory requirements. However, the proposed tables are solely dependent on the generator polynomial of interest, and are fixed for any number of errors N, while the previous methods lead to tables whose sizes grow exponentially with N.

In what follows, we will use the notation a to represent the binary vector associated with a binary polynomial $a(x)$. The i-th element (LSB-wise) of this vector will be denoted a_i. The addition of polynomials $a(x)$ and $g(x)$, denoted $a(x) + g(x)$, will become the XOR of vectors a and g, denoted $a \oplus g$. The left shift of $g(x)$ by n positions, denoted $g(x).x^n$ will be denoted $g \ll n$. The addition of two vectors should be interpreted as an XOR operation.

A. SINGLE ERROR CORRECTION

The state-of-the-art method achieves CRC syndrome simplification through successive additions (XORs) of the current error pattern vector (initialized to the computed syndrome) with left-shifted versions of the generator polynomial, and an updating of the error vector e with the result of the addition until a single-bit error pattern appears in e (i.e., a single-bit is set to 1). In the proposed method, we aim to avoid these computations by storing the location (relative distance) of the single error corresponding to each syndrome value in a table. Because the table is directly indexed by these syndrome values, the error location can be accessed directly by using the syndrome as an index in the table. The first step of the proposed method is thus to generate the table for single error correction, following the steps illustrated in the flowchart given in Figure 1.

Step 1: The table is initialized to -1. We choose this initialization value as it cannot be a valid entry in the list, contrary to 0, which would correspond to a single error at position 0. The size of the table corresponds to the size of the possible set of syndromes, (i.e., $2^n - 1$, where n is the bit length of the syndrome s).

Step 2: The syndrome is then initialized to 0, which is equivalent to null vector $\mathbf{0}$, although this value is not of interest as it would indicate that there is no error in the packet.

Step 3: Next, the single error position P_1 associated with the syndrome s for a given polynomial g is determined. The

sub-steps to perform are illustrated in a separate flowchart in Figure 2, where it can be seen that the state-of-the-art single error algorithm is actually performed in steps 1, 2, 4, 5 and 6. Figure 2 presents an error handling process similar to the CRC-based single error correction available in the literature, where the packet length m is set to the largest packet size before single error periodicity (i.e., we denote this length as the cycle length of a generator polynomial). The first step is to initialize e to m zeros and set its LSB value to s. Then, for each syndrome, we thus count the number of non-null values in the error vector e and check if this number equals 1. If that is the case, the corresponding error position is set in the corresponding entry of the table. If not, the LSB non-null value is canceled and the error vector is checked again, until we reach the cycle length. The cycle length for any generator polynomial is equal to or less than $2^n - 1$, and can be retrieved experimentally by determining the distance between two single error patterns having the same syndrome.

Step 4: The single error position provided by step 3 is stored in the table.

Step 5: If the whole set of possible syndromes has been tested (final value of s reached), the process stops. If not, the syndrome is updated by incrementing its value by 1.

At the end of the process, a table similar to Table 2 is obtained, and shows an example for the polynomial generator $g(x) = x^5 + x^4 + x^2 + 1$. The index column is actually implicit and does not need to be stored; in fact, it was added here simply for better readability. In this table, P_1 denotes the degree of the single error position (i.e., the single non-null position in e). For instance, looking at the table, the error position is x^{10} when the computed syndrome is 7. It can be seen that half the indexes indicate a position P_1 of (-1). This notation means that there is no single error position or

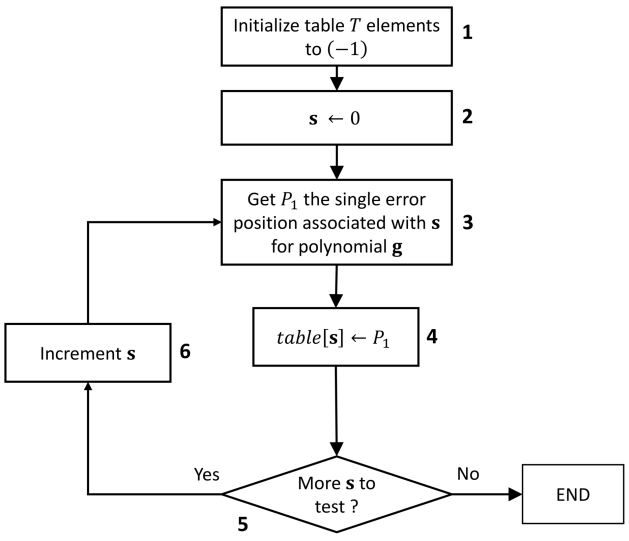


FIGURE 1: Flowchart representing the steps to build the table containing the whole list of syndromes along with their associated single error positions

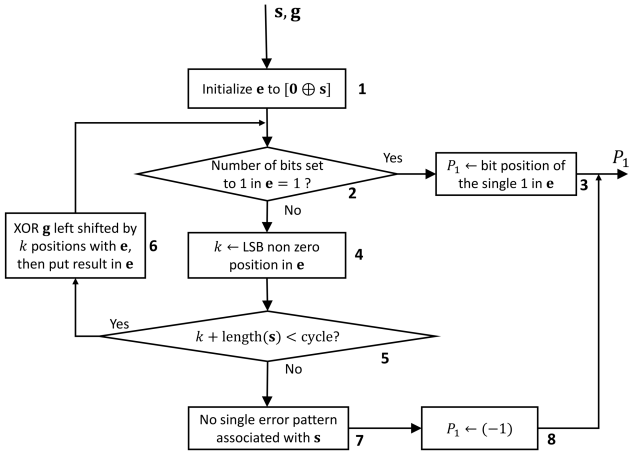


FIGURE 2: Flowchart representing the steps to obtain the single error position from a computed syndrome (step 3 in Fig 1). In step 1, **0** represents the null vector

TABLE 2: Single error position table generated for a CRC-5 of generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$

Index	P_1	Index	P_1
0	-1	16	4
1	0	17	-1
2	1	18	-1
3	-1	19	-1
4	2	20	-1
5	-1	21	5
6	-1	22	8
7	10	23	-1
8	3	24	-1
9	-1	25	9
10	-1	26	14
11	7	27	-1
12	-1	28	12
13	13	29	-1
14	11	30	-1
15	-1	31	6

solution associated with the corresponding syndrome value. Since the generator polynomial's parity is even, it mainly corresponds to syndromes with even parity. Of note, when the generator polynomial has even parity, syndrome values with even parity cannot lead to odd parity error patterns, and thus, they cannot lead to single error solutions. We will see later in this paper that there is also a specific syndrome for which no solution exists when the generator polynomial has even parity.

The steps for identifying candidate single error positions when receiving a corrupted packet are illustrated in Algorithm 1. The method proceeds by simply checking the value of P_1 in the table at the index corresponding to the computed syndrome s, as shown in step 2. If P_1 is different from (-1) and indicates a value less than the length of the packet, a candidate is appended to the list. A search for more valid

candidates is conducted, considering the cycle length of the generator polynomial used. For example, in Table 2, the cycle length is $(2^4-1)=15$. Thus, the position P_1 ranges from 0 to 14, producing corresponding syndromes obtained from the Index column in the table (e.g. syndromes of 1 and 26 correspond to $P_1=0$ and $P_1=14$, respectively). If an error occurs at position 15, as it corresponds to position $P_1=(0+cycle)$, it will produce a syndrome equal to 1. For a packet of length 50, when computing syndrome 1 (i.e., the same as position $P_1=0$), the error can thus be at positions 0, 15, 30 and 45. Since valid single error positions are cyclic, we test all possible values up to the length of the received packet. An error correction method may correct the packet if a single candidate is returned by Algorithm 1 or if one of them passes additional validations such as a UDP or TCP checksum [25], [26].

The proposed single error correction approach differs from the state-of-the-art lookup table approaches as it considers the whole set of possible syndromes that can occur, regardless of the packet length. Moreover, current lookup table approaches are sorted in a bit error position-wise order, making it mandatory to scan the table to retrieve the error position. By sorting the table in a syndrome-wise order, we do not have to scan the table as it can be indexed with the computed syndrome value.

B. DOUBLE ERROR CORRECTION

In order to handle double error correction, the strategy proposed in the state-of-the-art CRC-based multiple error correction method is to force a first error and then search for the second using the single error method [18]. As the first error is forced at a specific position F_1 by setting all lower positions to 0 and this position to 1 through successive conditional additions of g at required positions, the resulting syndrome is expected to correspond to a single error syndrome. Checking the position of this error by applying the single error method will indicate whether the remaining error at position P_1 (as well as the other single errors at positions separated by the cycle length for sufficiently large packets) occurs within the packet, which would lead to a new candidate error pattern with errors at positions (P_1+1+F_1) and F_1. Thus, to consider the entire set of possibilities, single error management must be called for each possible location of the first error. At the end of each verification, the LSB error is moved by one bit toward the most significant bit (MSB), from the LSB position where $F_1=0$ to the end of the payload at position $F_1=m-1$. This repositioning is conducted in two steps:

- The previous bit position tested (former F_1, forced to 1) is set to 0 by XORing g at this position.
- The following bit toward the MSB is considered as the new first error (i.e., $F_1 \leftarrow F_1+1$), and is either kept at 1 if it was 1 or forced to 1 by XORing g at this position if it was 0.

The process is repeated until the forced bit position reaches the position m, corresponding to the length of the protected data (position $m-1$ is the last position processed).

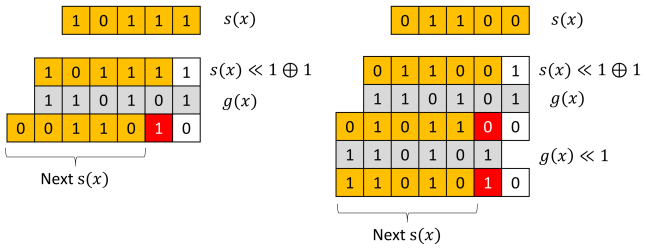
It can be seen that based on these steps, the succession of syndromes is always the same for a given generator polynomial, and therefore we propose, in this method, to store the value of the next syndrome based on the current one. More specifically, the *next* element s' of a given syndrome s is:

$$next = \begin{cases} (s' \oplus g) \gg 1 & \text{if } s_0 = 0 \\ s' \gg 1 & \text{if } s_0 = 1 \end{cases} \quad (4)$$

where s' is defined as:

$$s' = [((s \ll 1) \oplus 1) \oplus g] \gg 1 \quad (5)$$

The form of (5) actually corresponds to an updating of the forced error position by canceling the previously forced one and setting the new forced position to 1. These steps are illustrated in Figure 3, where both cases discussed are



(a) Next element case 1 (b) Next element case 2

FIGURE 3: Illustration of the next elements generation for the two possible cases. In case 1, the newly forced value is already set to 1 after the first XOR of $g(x)$. In case 2, the newly forced position is 0 after the first XOR, which requires a second XOR of $g(x)$ to force this position to 1. The red boxes represent the new position to force.

Algorithm 1 SingleErrorCorrection($T[2^n]$,s,n,m,$cycle$)

Inputs:

$T[2^n]$: indexed table containing P_1 for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial

Output:

E_1: the list of valid error positions for single-bit error

```
E1 ← {}
P1 ← T[s]
if P1 ≠ −1 then
    while (P1 < m + n) do
        Add P1 to E1
        P1 ← P1 + cycle
    end while
end if
Return E1
```

presented. In Figure 3a, the syndrome is first left-shifted by one position and 1 is appended as the LSB, which represents the formerly forced position. We cancel this position by XORing the generator polynomial. As the new forced position (represented by a red box) is already set to 1, no further step is required, and we simply consider the next syndrome. On the other hand, in Figure 3b, the new forced position is not set to 1 after the first XOR. In this case, therefore, another XOR with the generator polynomial is needed to produce the next syndrome. In the example of Figure 3a, it is clear that starting from the syndrome $s = [10111] = 23$, we cancel the previously forced value by XORing the generator polynomial and leave the new position to force as 1, identified as a red box. We thus obtain the next element for the syndrome equal to 26, which is $s = [00110] = 6$. Linking a syndrome to the next will significantly reduce the complexity of the approach since the arithmetic operations will be pre-computed and stored in the reference table.

In [18], it was shown that throughout the process of identifying error patterns with N or fewer errors, which operates from the LSB to the MSB, only a range of n bits can contain non-zero values since the lower positions are already eliminated and the higher positions, initialized to 0, have not yet been altered. The proposed method exploits this property by considering only values within this range (sliding window) and processing them as the syndromes of interest. In the case of a double error, the forced error position will alter the syndrome value on which our subsequent single error position determination is based. Essentially, we consider the effect of forcing the error position on the syndrome used for identifying the remaining error position. This latter position becomes relative to the forced position. It should be noted that the forced error position, represented by a red box in Figure 3a, is not part of the syndrome considered for single-bit error determination, but is implicitly present throughout the process described in (5).

The following are the steps for generating the table containing both the single error position and the next element, as illustrated in Figure 4:

Step 1: The whole table is initialized to (-1). The number of entries in the table is the same as for the single error table previously described, with the difference lying in the number of columns. For each row, another column containing the next element of the current syndrome is appended.

Step 2: We initialize a, a variable representing the first syndrome of the loop, to 1.

Step 3: We now begin to complete the table. In order to avoid unnecessary computations, we first check that the current table position has yet not been processed and that the current values of a and g are able to produce a candidate.

Step 4: If that is the case, we initialize the next element to 0 and set a local syndrome b to the value of a.

Step 5: We first fill the single error position through to steps described in the previous section (see Figure 2).

Step 6: The next element is identified for the current syndrome b and the generator polynomial g. The steps to determine the next element are described in a separate flowchart in Figure 5. To generate the next element, we cancel the previously forced bit position by XORing g and set the new forced position through a new XOR if necessary.

Step 7: We store the single error position and the next element in the table.

Step 8: The local syndrome b is then updated to the next position computed in step 6.

Step 9: If the next element has not yet been processed, we keep on computing its associated single error position and next element. If the next element has already been processed, the current cycle of $next$ has looped, which means that we have to increment a and start a new loop.

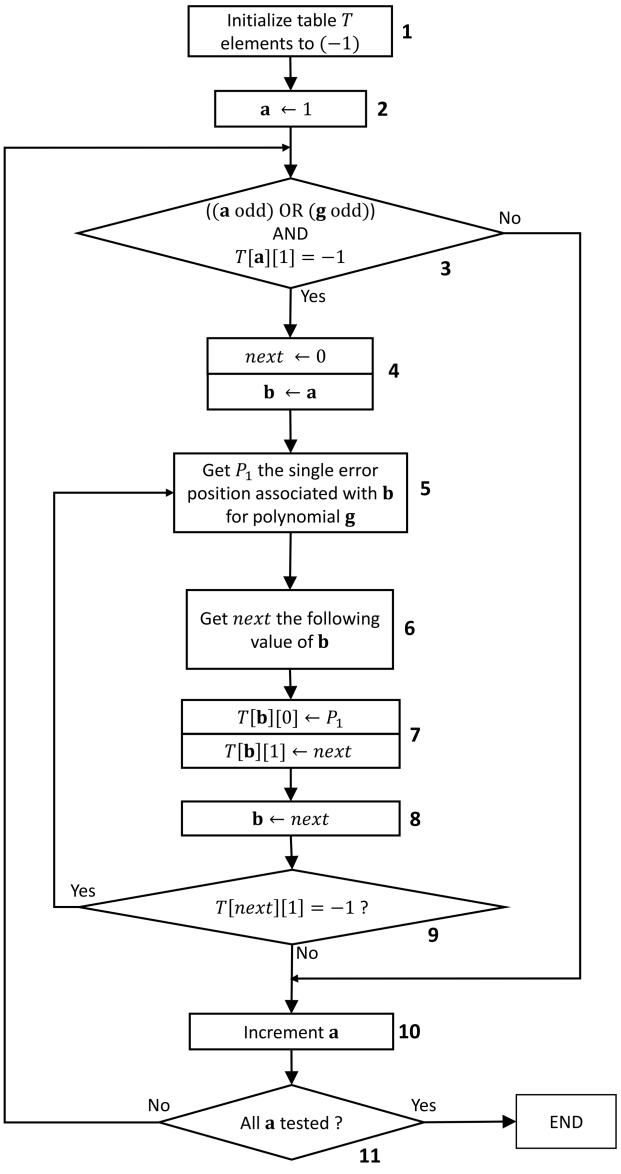


FIGURE 4: Flowchart representing the steps to generate the table T containing single error position and next element to handle double error correction

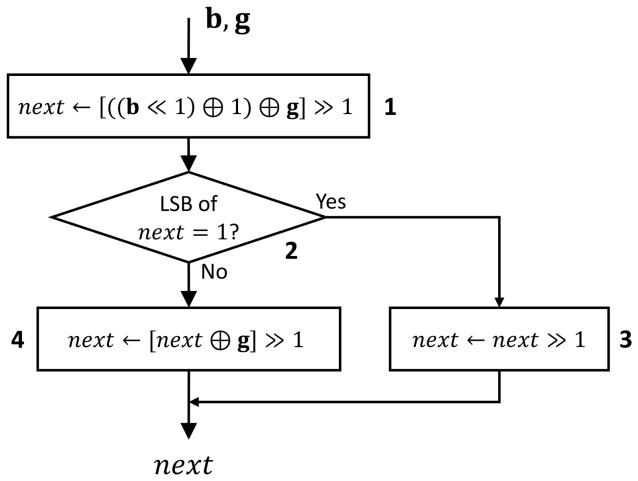


FIGURE 5: Flowchart representing the steps to follow to generate the next element from the computed syndrome and the generator polynomial (step 6 of Fig. 4, illustrated in Fig. 3)

Step 10: Once every value of a has been tested, the process is ended as the table is complete.

Table 3 shows the complete table at the end of the process for the generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$. The P_1 column corresponds to the relative distance of the remaining single error, and the $next$ column stores the next syndrome to be tested (i.e., the syndrome after the forced position is updated). Algorithm 2 illustrates the different steps for performing the correction of double-bit errors when the table is generated:

Step 1: The candidate list containing the candidates with 2 errors is initialized as an empty set.

Step 2: A vector s' of length n bits is created, and it will correspond to the updated version of the original syndrome s.

Step 3-7: As each $next$ element from the table has been generated given that an error had already been forced, the first error must be added prior to navigation within the table. Thus, if the LSB of the syndrome s is 0, an XOR with g must be performed to force the first position. Since this forced bit is implicit and not part of the syndrome, a right shift is performed in both cases. Note that in practice, such operations can also be stored in the table. Adding two columns to store these results would double the memory storage needed but avoid any arithmetic operation.

Step 8: At this point, we consider the first error at position 0, and perform the loop over all possible forced positions, from 0 to $m + n - 1$.

Step 9-10: The relative single error position is accessed from table T. Two conditions must be met to consider the candidate as valid. First, the position P_1 must indicate a relative distance (i.e., the single error position for a syndrome value equal to the index), as tested in step 10, and then the position of the error indicated by P_1 must be in the range of

Algorithm 2 TwoErrorCorrection($T[2^n][2]$,s,n,m,$cycle$)

Inputs:

$T[2^n][2]$: indexed table containing P_1 and $next$ elements for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial

Output:

E_2: the list of valid error positions for double-bit error

```
1:  E2 ← {}
2:  Let s' be a vector of length n
3:  if s ∧ 1 = 0 then
4:      s' ← [s ⊕ g] ≫ 1 // Can be stored in a table
5:  else
6:      s' ← s ≫ 1 // Can be stored in a table
7:  end if
8:  for F1 = 0 to m + n − 2 do
9:      P1 ← T[s'][0]
10:     if P1 ≠ −1 then
11:         while (P1 < m + n − F1 − 1) do
12:             Add (P1 + 1 + F1, F1) to E2
13:             P1 ← P1 + cycle
14:         end while
15:     end if
16:     s' ← T[s'][1]
17: end for
18: Return E2
```

the packet.

Step 11: As the distance P_1 is relative to the current forced position F_1, $P_1 < m + n$ is not enough to guarantee that the error pattern is possible. Thus, considering a relative packet size of $m + n - F_1$, we ensure that we identify only valid error patterns.

Step 12: When both conditions described in Step 10 and Step 11 are met, a candidate comprising the forced position F_1 and the remaining error position P_1+1+F_1 is appended to the list.

Step 13: As the single error position is cyclic, we should test every possible error pattern. At each loop, we add the $cycle$ length to the position P_1 and check if the resulting value is within the range of the packet. If not, the next forced position must be tested.

Step 16: Once a forced position has been processed, the updated syndrome, corresponding to the syndrome resulting from the next forced position is accessed from the table. It corresponds to the second column of the current syndrome index, as illustrated in Table 3.

C. N ERROR CORRECTION

For N errors, the generalization of the strategy used for double error correction is performed, as illustrated in Algo-

TABLE 3: Table generated for double-bit error correction for a CRC-5 of generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$

Index	P_1	$next$	Index	P_1	$next$
0	-1	23	16	4	31
1	0	13	17	-1	5
2	1	22	18	-1	30
3	-1	12	19	-1	4
4	2	21	20	-1	29
5	-1	15	21	5	7
6	-1	20	22	8	28
7	10	14	23	-1	6
8	3	19	24	-1	27
9	-1	9	25	9	1
10	-1	18	26	14	26
11	7	8	27	-1	0
12	-1	17	28	12	25
13	13	11	29	-1	3
14	11	16	30	-1	24
15	-1	10	31	6	2

rithm 3. The concept is as follows. $(N-2)$ bit values must be forced to 1 at each step (initialized to the $(N-2)$ LSBs of the syndrome). Then, the double error method is performed on the remaining bits of the packet. The number of errors, N, is then decreased and when it reaches 1, single error correction is performed on the computed syndrome.

For each forced bit error position (in the double error approach after forcing the $(N-2)$ bits), the $next$ element (forcing the next bit toward the MSB) can be accessed from the reference table to reduce the computational complexity and avoid arithmetic operations. Therefore, we look at every possible combination of forcing $(N-2)$ bit values to 1 within the $m-1$ first bits (LSB) of the packet. Let the forced bits be at positions $F_1, F_2, \ldots, F_{N-2}$ with $F_1 < F_2 < \ldots < F_{N-2}$ (sorted by increasing the bit position). Forcing some bits to 1 means that the bits at positions $F_1, F_2, \ldots, F_{N-2}$ are set to 1 and the other bit positions below position F_{N-2} are set to 0. With this definition, it can be noted that all the bits with a position below F_{N-2} are actually forced (to a value of 0 or 1). Steps 4 to 12 of Algorithm 3 illustrate this forcing process.

Steps 1-2: First, we initialize both the candidate list to an empty set, and k, the local number of errors to consider, to its maximum value N.

Step 3: This step and the following ones are repeated for every value of k for which $N \geq k > 2$.

Step 4-5: At step 4, the set of forced bits $\mathcal{F}$ is initialized to the $(k-1)$ LSB positions. Step 5 shows that the process continues until $\mathcal{F}$ is set to the $(m+n)$ MSB positions. In step 8, & and || represent the *logical and* and *logical or* operations, respectively. In this approach, "forced bits" means the positions forced to 1, representing the $(N-2)$ errors placed. However, it should be understood that if a bit within the range of forced bit positions is not forced to 1, then it is forced to 0. More generally, we focus on the positions forced to 1 because the algorithm strives to set the other positions to 0 during its elimination process.

Algorithm 3 NErrorCorrection($T[2^n][2]$,s,n,m,$cycle$,N)

Inputs:

$T[2^n][2]$: indexed table containing P_1 and $next$ elements for each syndrome
s: the syndrome vector
n: the length of the syndrome vector
m: the length of the payload vector
$cycle$: the cycle length of the generator polynomial
N: the maximum number of errors to consider

Output:

E_N: the list of valid error patterns up to N errors

1: $E_N \leftarrow \{\}$
2: $k \leftarrow N$
3: **while** $k > 2$ **do**
4: $\quad\mathcal{F} \leftarrow (0, \ldots, k-2)$
5: $\quad$**while** $\mathcal{F} \neq (m+n-k+1, \ldots, m+n-1)$ **do**
6: $\quad\quad s' \leftarrow s$
7: $\quad\quad$**for** $i = 0$ to F_{k-2} **do**
8: $\quad\quad\quad$**if** $(s'_1 = 0$ & $i \in \mathcal{F})$ || $(s'_1 = 1$ & $i \notin \mathcal{F})$ **then**
9: $\quad\quad\quad\quad s' \leftarrow (s' \oplus g) \gg 1$ // Can be stored in a table
10: $\quad\quad\quad$**else**
11: $\quad\quad\quad\quad s' \leftarrow (s' \gg 1)$ // Can be stored in a table
12: $\quad\quad\quad$**end if**
13: $\quad\quad$**end for**
14: $\quad\quad m' \leftarrow m - (F_{k-2} + 1)$
15: $\quad\quad$Add TwoErrorCorrection(T,s',n,m',$cycle$), $\mathcal{F}$ to E_N
16: $\quad\quad\mathcal{F} \leftarrow$UpdateForcedPosition($\mathcal{F}$,$m$)
17: $\quad$**end while**
18: $\quad k \leftarrow k-1$
19: **end while**
20: Add TwoErrorCorrection(T,s,n,m,$cycle$), $\mathcal{F}$ to E_N
21: Add SingleErrorCorrection($T[][0]$,s,n,m,$cycle$) to E_N
22: Return E_N

Step 6: We start by initializing the binary vector s', representing the updated syndrome, to s.

Steps 7-11: Forcing of bit positions must be achieved through the successive addition of the generator polynomial vector and an accumulation in the updated syndrome s' in order to obtain the desired result.

The syndrome cannot simply be ignored and the bit values changed at desired positions. Rather, the equivalence relationship with the original syndrome must be maintained, i.e., only shifted versions of g may be added to it. This is done by starting at bit 0 of the syndrome, and if its value is the desired value, then nothing is done for that position. Otherwise, g must be added at that position (i.e., XOR performed), which contains 1 at its LSB, to modify it. The decision as to whether or not to perform the XOR is illustrated in step 8 of Algorithm 3.

As the method successively processes the next positions from LSB (bit 0) to MSB (bit F_{N-2}) in a similar fashion, it is

important that g be added at each step at the current position (i.e., g should be XORed at step 9 when processing position i) and the syndrome value when the conditions of step 8 are met until position F_{N-2} is processed. Otherwise, the syndrome is right-shifted by one position at each step, as shown in step 11.

In practice, these operations can also be stored in a table when searching for N error patterns. Thus, two additional columns would be added, doubling the required storage, but every computation would be stored in the table. Only the forced bit positions set $\mathcal{F}$ and the current position i are needed to perform the full candidate list generation.

From there, since all the forced bit positions have been properly set, the process described for double error handling is performed on the remaining length of the packet. The following are the remaining steps:

Step 14: We introduce m', which corresponds to the remaining length of the packet. It is crucial to take into account the fact that the double-bit error correction must be performed with the current position taken into account to ensure the whole set of possibilities are considered.

Step 15: We perform a double-bit error correction on the updated syndrome s', comprising the $(N-2)$ LSB forced positions, for a packet length m'. The candidate error patterns comprise the 2 positions returned by Algorithm 2 and the forced positions contained in $\mathcal{F}$.

Algorithm 4 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 $\text{len}(\mathcal{F})$ 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**

Step 16: Once the double error correction has been performed, we must update the forced error position, as illustrated in Algorithm 4, to test all possible $(N-2)$ forced positions.

Step 18: The main loop is performed for every number of errors k from N to 3 (i.e., when the error must be forced). When k reaches a value of 2, the last steps to perform are a double-bit error correction, followed by a single-bit error correction on the original syndrome s and payload length m, as shown in steps 20 and 21. We return the completed list of candidates E_N at step 22.

D. CYCLES OF $NEXT$ ELEMENTS

In this subsection, we study some properties of the generated lookup table. Although this knowledge does not impact the functionality of the proposed algorithms, it provides valuable insights into the nature of the solutions to expect, and that could be further exploited. Although more complex, the approach has similarities with periodic sequence and shift registers (linear and non-linear feedback shift registers) over Galois Fields [27], which have been studied in the literature [28]–[30]. We can see in the flowchart of Figure 4 that there are two distinct variables for determining the syndromes, namely, a and b. In the last subsection, a was described as the main loop syndrome and b as the local loop syndrome. While a was incremented by 1 at each main loop, the syndrome value of b was successively updated to its $next$ element until the $next$ element had been processed (i.e., all the syndromes in the current cycle had been added to the table). Such cycles are observed for every generator polynomial. Figure 6 presents an example of this cycle. In the figure, we present the $next$ element cycles for a generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$. It can be seen that since the generator polynomial parity is even, the sets of syndromes resulting from an odd or an even number of errors are disjoint. The value of each syndrome is expressed as a decimal value in Figure 6, where it can be seen that most parts of the table will be completed once these two cycles (i.e., two local loops on b) are performed.

A straightforward design to generate the table would consider the local odd and even loops only. However, we observed two exceptions to these cycles. In fact, two syndrome values are out of the cycle loops, and are represented by red circles in Figure 6. For the considered generator polynomial, these two syndrome values correspond to $s = 9$ and $s = 26$, respectively corresponding to $s = [01001]$ and $s = [11010]$ in binary vector representation, from MSB to LSB. We present the computation of the $next$ element for both cases in Figure 7, where they are referred to as self-loop syndromes, types I and II. It can be seen that in both cases, the $next$ element corresponds to the element itself. To understand why the $next$ element is the syndrome itself and to make sure that there is no other case than these two, we performed an analysis of such syndromes.

For the case presented in Figure 7a, applied to even parity polynomials, and based on the operations needed to obtain

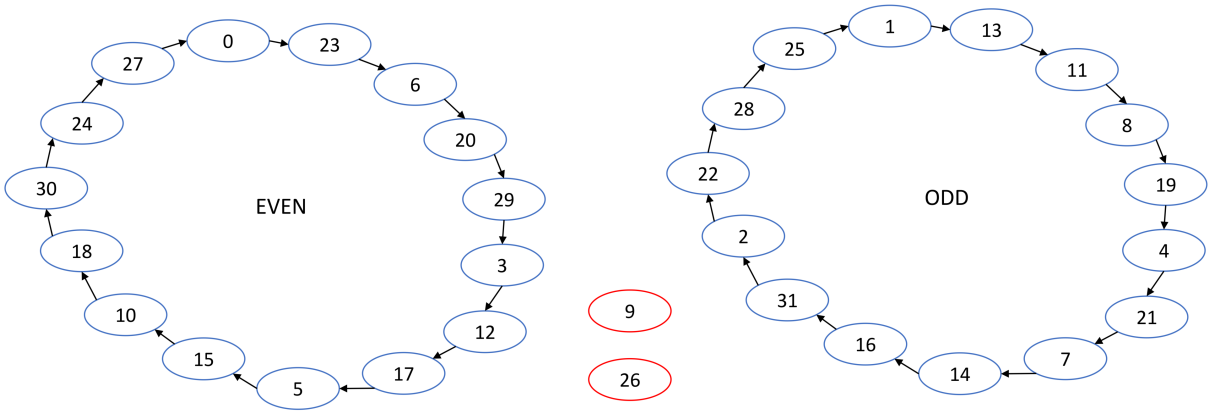


FIGURE 6: Example of cycles and exceptions when the generator polynomial is $g(x) = x^5 + x^4 + x^2 + 1$

the *next* element, it can be seen that:

$$\begin{aligned}&(s \ll 2) \oplus (1 \ll 1) = (s \ll 1) \oplus 1 \oplus g \\ \implies &(s \ll 2) \oplus (s \ll 1) = g \oplus (1 \ll 1) \oplus 1.\end{aligned} \tag{6}$$

This equation can be expressed as:

$$s_{i-1} = g_{i+1} \oplus s_i \quad \forall\,(n-1) \geq i \geq 1 \tag{7}$$

with $s_{n-1} = 0$. Because $g(x)$ has even parity, it can be shown that $s_0 \neq g_1$.

This equation can be applied to the generator polynomial to retrieve the even exception of this $g(x)$:

$$\begin{aligned}s_4 = 0, \quad s_3 = g_5 \oplus s_4 = 1, \quad s_2 = g_4 \oplus s_3 = 0 \\ s_1 = g_3 \oplus s_2 = 0, \quad s_0 = g_2 \oplus s_1 = 1\end{aligned} \tag{8}$$

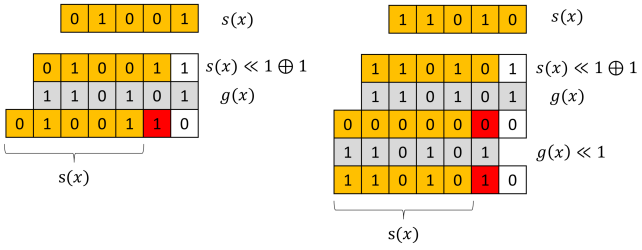
At the end of the process, the only exception of type I is $s = [01001] = 9$, which corresponds to the one identified in Figure 6.

The other exception can be expressed in the same way, using the operations described in Figure 7b:

$$\begin{aligned}&(s \ll 2) \oplus (1 \ll 1) = (s \ll 1) \oplus 1 \oplus g \oplus (g \ll 1) \\ \implies &(s \ll 2) \oplus (s \ll 1) = (g \ll 1) \oplus g \oplus (1 \ll 1) \oplus 1\end{aligned} \tag{9}$$

which can be expressed as:

$$s_{i-1} = g_{i+1} \oplus g_i \oplus s_i \quad \forall\,(n-1) \geq i \geq 1 \tag{10}$$



(a) Self-loop type I (b) Self-loop type II

FIGURE 7: Representation of the self-loop *next* elements for the two syndromes exceptions in Figure 6

with $s_{n-1} = 1$. Because $g(x)$ has even parity, it can be shown that $s_0 = g_1$. By applying this equation to the considered generator polynomial, the following is obtained:

$$\begin{aligned}s_4 = 1, \quad s_3 = g_5 \oplus g_4 \oplus s_4 = 1, \quad s_2 = g_4 \oplus g_3 \oplus s_3 = 0 \\ s_1 = g_3 \oplus g_2 \oplus s_2 = 1, \quad s_0 = g_2 \oplus g_1 \oplus s_1 = 0\end{aligned} \tag{11}$$

At the end of the process, the only exception of type II is $s = [11010] = 26$, which also corresponds to the one identified in Figure 6.

We will now study the case of odd parity generator polynomials. Since the addition of such polynomials to a syndrome changes the parity, it can be seen in Figure 7a that the next element of a syndrome $s(x)$ cannot possibly be $s(x)$ itself. However, since the generator polynomial is added twice in 7b, it can be shown that the solution is:

$$s_i = g_{i+1} \quad \forall\,(n-1) \geq i \geq 0 \tag{12}$$

The solution can also be expressed as $s = (g \gg 1)$. Therefore, only type II self-loops exist for odd parity generator polynomials.

Studying these elements is important because doing so provides insight into the nature of the solutions to be expected. The *next* elements have the potential to generate numerous candidates. Let us assume that $N-1$ bit positions have been forced and that the syndrome is one of these two *next* elements. If the associated P_1 leads to the remaining bit error being at position k within the packet, then the same $N-1$ forced bits along with any position $k+i < m+n$ with $i \geq 1$ also constitute a valid error pattern. For example, this *next* element in Table 3 is at index 26. For this syndrome value, the single error position P_1 is 14, and the *next* element itself is 26. If after forcing $(N-1)$ errors, we obtain this syndrome and the remaining length is 50 bits, we get a first candidate error pattern, with errors at a forced position and at $(F_{N-1}+1+P_1)$. In the very next step, the update syndrome is 26 once again. As the remaining length is now 49 bits, another candidate is found, with forced position F_{N-1} and position P_1 both increased by one. At each step, a new

candidate is appended to the list until reaching the end of the message.

E. SYNDROMES WITH NO SOLUTION FOR SINGLE ERROR

We also observed an exception in the single error position search. Given the parity of both the generator polynomial and the syndrome, we are able to determine whether the number of errors that occurred in the packet is odd or even. Based on this knowledge, a particular entry of the table represented in Table 3 is worthy of interest. The syndrome $s = 19 = [10011]$ has an odd number of non-null coefficients, and is thus expected to have an associated single error position. However, Table 3 shows that there is no single error position for this syndrome, for any possible packet length. Figure 8 illustrates the single error correction applied to the syndrome. It can be clearly seen that at each step, the resulting syndrome corresponds to the originally computed one, and therefore, there is no possibility of having only one non-null coefficient at any step of the process.

Similarly to the *next* element cycles described previously, it can be seen that for this exception to be realized, the following equality must be met for an even parity generator polynomial:

$$(s \ll 1) = s \oplus g \implies (s \ll 1) \oplus s = g \tag{13}$$

which can be expressed as:

$$s_{i-1} = g_i \oplus s_i \quad \forall\,(n-1) \geq i \geq 1 \tag{14}$$

with $s_{n-1} = 1$. It can easily be shown that $s_0 = 1$ for even parity generator polynomials, and that the syndrome therefore never contains a single error pattern. Again, applying this equation to the generator polynomial used here yields:

$$\begin{aligned} s_4 = 1, \quad s_3 = g_4 \oplus s_4 = 0, \quad s_2 = g_3 \oplus s_3 = 0 \\ s_1 = g_2 \oplus s_2 = 1, \quad s_0 = g_1 \oplus s_1 = 1 \end{aligned} \tag{15}$$

At the end of the process, the syndrome is $s = [10011] = 19$, which is the one identified in Table 3.

For odd parity generator polynomials, syndromes with no solution must belong to a cycle comprising an even number of syndromes, where an even parity syndrome leads to an odd parity syndrome after the addition of the generator polynomial, and vice versa. Let us now investigate the conditions for having a pair of syndromes with no solution (i.e., the shortest

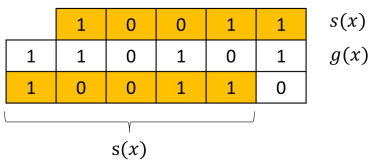


FIGURE 8: Single error correction method performed on syndrome $s(x) = x^4 + x + 1$ using a generator polynomial $g(x) = x^5 + x^4 + x^2 + 1$ (even parity)

TABLE 4: Value of syndrome exceptions for commonly used generator polynomials (CRC-8-CCITT, CRC-16-CCITT, CRC-24-BLE and CRC-32-Ethernet)

	Self-loop (type I)	Self-loop (type II)	No single error
CRC-8	126	131	253
CRC-16	30,735	34,832	61,471
CRC-24	8,388,324	8,389,421	16,776,649
CRC-32	_____	2,187,366,107	_____

such cycles). Let s and s' be these two syndromes. They must meet the following expression:

$$\begin{aligned} &(s' \ll 1) \oplus s = g \text{ and } (s \ll 1) \oplus s' = g \\ \implies &(s' \oplus s) \ll 1 = s \oplus s' \end{aligned} \tag{16}$$

which can be expressed as:

$$s_{i-1} \oplus s'_{i-1} = s_i \oplus s'_i \quad \forall\,(n-1) \geq i \geq 1 \tag{17}$$

with $s_{n-1} = s'_{n-1}$. It follows that $s_i = s'_i, \forall i$, and therefore, there is no cycle of two elements, one leading to the other, for odd parity generator polynomials. Further investigation is required to conclude on the existence of longer cycles.

In Table 4, we present the decimal values of the syndrome exceptions for different generator polynomials. In it, the syndrome whose *next* element is itself is denoted as "Self-loop". As has been demonstrated, there are two such types of elements. A syndrome that does not provide a single error candidate is denoted "No single error". Since CRC-32-Ethernet uses an odd parity generator polynomial, there is no such syndrome identified, and it exhibits a type II self-loop.

Having this knowledge on the structure of cycles and exceptions in CRC error correction can help save computations if such a syndrome is spotted at the receiver. By identifying an exception, we can avoid unnecessary computations while ensuring the exhaustive list of error patterns is obtained.

IV. PERFORMANCE AND COMPLEXITY

In this section, we compare different performance aspects of the proposed CRC-based error correction method using an optimized table (which we will refer to as **CRC-ECOT**) to several other CRC-based error correction methods, listed next:

- **Arithmetic operations (CRC-ECA)**: the method described in [18], which generates the candidate list using logical operations on-the-fly, and does not require storing a table.
- **Explicit lookup table (CRC-ECEXP)**: the traditional lookup table approach [13], which is based on storing syndromes and their associated error positions. ECEXP (explicit) means that the error positions are explicitly inserted in the lookup table. Note that this lookup table approach was recently used in a novel Polar SCL Decoding method [16].

- **Implicit lookup table (CRC-ECIMP)**: a proposed design of a lookup table approach similar to CRC-ECEXP, but where the error positions are not added to the table. The values of the error positions correspond to an implicit index (specific order in which the error positions are scanned) of the associated syndrome, which reduces the memory needed to implement such tables.
- **Exhaustive search (CRC-ECES)**: this corresponds to the arithmetic brute force scheme. No table is required in this method, but all the possible combinations of N error positions are successively tested to determine which ones lead to the computed syndrome.

Note that for our tests, we implemented the EC-ECOT version, as proposed in Algorithm 2, where the table size is smaller since steps 9 and 11 are not included.

A. COMPUTATIONAL COMPLEXITY

The methods compared have the same ability to correct multiple errors using the CRC syndrome. In terms of computational complexity, the CRC-ECOT method requires fewer operations as most of the computations are performed offline and then integrated into the table. We propose comparing the complexity of the method to that of the CRC-ECA and CRC-ECES approaches. Upon reception of a corrupted packet of m bits, the CRC-ECES method would test every error pattern up to N errors, i.e., it flips the error positions of the pattern and then computes the syndrome over the reconstructed packet. If the syndrome is null, a valid error pattern is found.

The complexity in this section is expressed as the number of additions to perform with g. In the case of CRC-ECES, m such operations are required for each long division. Thus, for the search of a single error, there are m long divisions to perform in order to test every possible error position, yielding m^2 operations. By extending this process to N errors, we obtain a global complexity of $O(m^{N+1})$.

The CRC-ECA method only performs one long division for a single error search, which can be performed through m additions of g. Extending this method to search for several errors requires setting $(N-1)$ forced positions, and a long division must be performed on the remaining length of the packet. A double-bit error search thus requires m^2 operations. Generalized to the search of N error patterns, it yields a global complexity of $O(m^N)$, as demonstrated in [18].

The complexity of the proposed CRC-ECOT method can be expressed through the complexity required to build the table and through the complexity required to perform the identification of the candidate error patterns. The former is highly complex, but is performed offline, prior to the communication. We will thus focus on the latter. For CRC-ECOT, the complexity up to double-bit error correction is extremely low thanks to the $next$ element column integrated into the table. Single error correction requires a single lookup, while double error correction requires m table lookups. When the number of errors is increased, forced positions must be set. Thus, when searching for N errors, $(N-2)$ positions must be tested and the global complexity of the proposed approach will be $O(m^{N-2})$ additions of g times m table lookups. These gains are significant since the complexity increases significantly as N increases. Note that we could completely eliminate the arithmetic operations at the expense of a larger lookup table by storing $[s' \oplus g] \gg 1$ and $s' \gg 1$ for all syndrome values[3] in steps 9 and 11 of Algorithm 3. This would allow to further increase the speed for $N > 2$ by a new global complexity of $O(m^{N-1})$ table lookups.

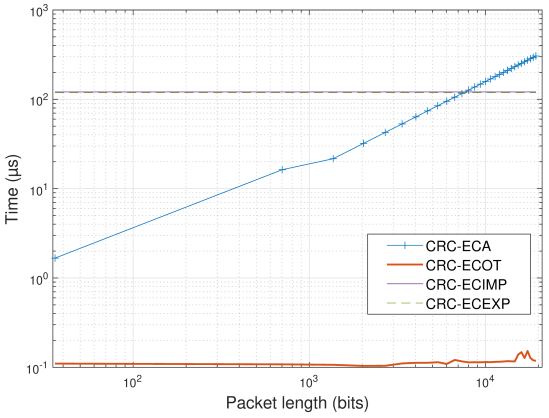
We tested a C implementation of the CRC-ECA, the CRC-ECOT and both the CRC-ECEXP and CRC-EXIMP approaches to compare their processing speeds on a Raspberry Pi model 4 [31], with a Broadcom BCM2711 processor, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz and 8 GB RAM. The test results are available in Figure 9, with log-log scales.

First, it can be seen that when searching for single errors, as in Figure 9a, the processing time per symbol of the CRC-ECOT is independent of the length of the packet. In fact, as the table is syndrome-indexed, when a corrupted packet is received, we only need to read the content of the table at the computed syndrome's entry, regardless of the packet's length.

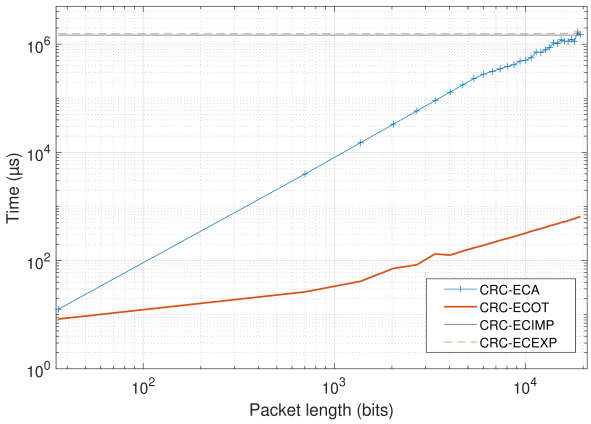
On the other hand, the CRC-ECEXP and CRC-ECIMP lookup table-based methods must scan the packet in order to search for potential single error positions, which leads to higher processing times as the maximum packet length increases. In our example, the maximum packet length considered is 2500 bytes. The processing time per syndrome on the tested CPU for the proposed CRC-ECOT method is 100 ns, on average, whereas the CRC-ECA method's processing time ranges from 1.7 µs for the smallest payload (36 bits) to 300 µs for the largest payload considered. Both lookup table-based methods offer a constant processing time since they must scan the whole table for any computed syndrome. This processing time is 120 µs on the tested architecture. The proposed method's speedup for single error correction is thus 1200 times, as compared to these table approaches, and ranges from 17 to 3000 times faster, as compared to CRC-ECA, depending on the packet size.

The double error correction case illustrated in Figure 9b shows that again, CRC-ECOT processes each syndrome much faster than do all the other tested methods. Here, it can be seen that the processing time is not constant for CRC-ECOT as the packet length increases. For double error correction, we must consider the $next$ element of each syndrome for the whole length of the packet. The processing time thus depends on the packet length. It is also interesting to note that as the packet length increases, the processing time gains also increase. When considering the smallest packet length (36 bits), the average processing time for double error correction is 8.3 µs for the proposed method and 12.6 µs for the arithmetic method, which gives a time ratio of 1.5 between the two methods. This ratio increases with the packet length, and is ultimately very significant for the

[3]Note that depending on the architecture on which the algorithm is implemented and the generator polynomial of interest, it may be less complex to perform $s' \gg 1$ than to retrieve it from a table.



(a) Single error correction



(b) Double error correction

FIGURE 9: Comparison of the average processing time per syndrome for single and double error corrections for state-of-the-art CRC-based error correction [18] (CRC-ECA), the proposed optimized table (CRC-ECOT), the implicit table (CRC-ECIMP) and explicit table [13] (CRC-ECEXP) applied to CRC-16-CCITT

largest payloads. The proposed method requires 642 µs to process a syndrome while the arithmetic method needs 1.5 s, yielding a time ratio of about 2300. Of course, these gains are due to the design of the proposed method as most of the computation is performed offline, prior to communication. CRC-ECIMP and CRC-ECEXP processing times for double-bit error correction are still constant at 1.5 s. The proposed method's speedup goes from 174,000 for the smallest packets to 2300 for the largest ones, as compared to these table approaches, depending on the packet size. Note that these speedups can be further increased by storing the values of steps 9 and 11 of Algorithm 3 in the table, but at the cost of increased memory requirements.

B. MEMORY REQUIREMENTS

We also compared the memory required to store the CRC-ECOT and CRC-ECEXP tables in Table 5, when using a packet of 1500 bytes, as it is the largest payload available in Ethernet (MTU). We compared these approaches for various generator polynomials:

- CRC-8-CCITT, where $g(x) = x^8 + x^2 + x + 1$.
- CRC-16-CCITT, used to protect the headers of 802.11 [3] and in low consumption 802.15.4 [32] communications, where $g(x) = x^{16} + x^{12} + x^5 + 1$.
- CRC-24-BLE, used to protect Bluetooth Low Energy [33] packets, where $g(x) = x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$.
- CRC-32-Ethernet, used to protect the entire packet in Ethernet [4] protocol, where $g(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$.

The CRC-ECEXP approach calls for storage of a syndrome along with the corresponding error positions for every possible error case, yielding the following memory requirement:

$$M_{\text{CRC-ECEXP}} = \binom{m+n}{N} \times [\text{length}(s) + (2 \times N)] \text{ bytes} \tag{18}$$

where $m + n$ is the bit length of the packet, $\text{length}(s)$ corresponds to the byte length of the syndrome, and N is the maximum number of errors. We considered positions to be integer variables of 2 bytes, since the maximum single position is 12,000 in this implementation.

We also propose a comparison with the CRC-ECIMP method, which is an implementation requiring less memory than CRC-ECEXP. The implicit table approach consists in indexing a syndrome with its associated error position, which reduces the size of the table:

$$M_{\text{CRC-ECIMP}} = \binom{m+n}{N} \times \text{length}(s) \text{ bytes} \tag{19}$$

This strategy calls for knowledge of error position management when considering several errors (e.g., syndrome at index 12,001 corresponds to a double error at positions $(0, 1)$).

The proposed CRC-ECOT approach requires listing, for all the syndromes, the *next* element, whose size is always of $\text{length}(s)$ bytes, as well as the error position P_1. Note that considering that the table is initialized to (-1), a negative element, the number of bits required to store P_1 is $\log_2(cycle + 1) + 1$. As most *cycle* lengths are $(2^{n-1} - 1)$ as shown in Table 5, it would take n bits to store P_1. It can be seen that the *cycle* length for CRC-32 is $2^n - 1$, yielding the need for an additional bit, for a total of $(n + 1)$ bits to store P_1 (i.e., 33 bits[4] for CRC-32). Except for such cases,

[4]Note that some architectures will not support 33-bit numbers efficiently, and will consider 64 bits when exceeding $2^{32} - 1$, which would double the memory required.

TABLE 5: Comparison of the memory required for the tables in the CRC-ECEXP [13] and the proposed CRC-ECOT approaches. The packet considered here has a length of 1500 bytes. Note that the arithmetic method is table-free, and thus, its memory requirements remain negligible for any case described in this table

Nb Errors N	CRC-8-CCITT (cycle = 2^7-1)		CRC-16-CCITT (cycle = $2^{15}-1$)		CRC-24-BLE (cycle = $2^{23}-1$)		CRC-32-Ethernet (cycle = $2^{32}-1$)	
	Lookup Table	Proposed	Lookup Table	Proposed	Lookup Table	Proposed	Lookup Table	Proposed
1	36 kB	256 B	48 kB	131 kB	60 kB	50.5 MB	72 kB	17.7 GB
2	360 MB	1 kB	432 MB	524 kB	504 MB	202 MB	576 MB	69.3 GB
3	2.02 TB	1 kB	2.30 TB	524 kB	2.60 TB	202 MB	2.88 TB	69.3 GB
4	7.77 PB	1 kB	8.64 PB	524 kB	9.50 PB	202 MB	10.4 PB	69.3 GB

the memory required can thus be expressed as:

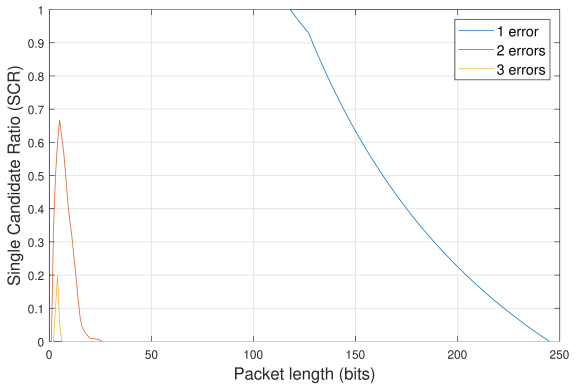
$$M_{\text{CRC-ECOT}} = \begin{cases} 2^n \times \text{length}(s) \text{ bytes}, & \text{if } N = 1 \\ 2^n \times 4 \times \text{length}(s) \text{ bytes}, & \text{if } N > 1 \end{cases} \quad (20)$$

It can first be seen in Table 5 that, unlike in CRC-ECEXP, where the table size is a function of the number of errors considered, the proposed approach has a fixed length for $(N > 1)$. In fact, for single error correction, the column comprising the $next$ elements is not needed, and with double error correction, the columns can store both the $next$ element and the results of steps 9 and 11 of Algorithm 3, which multiply the storage needed by 4. The table size for the proposed approach is less than for CRC-ECEXP, for all generator polynomial lengths n when $(N > 2)$. The proposed table is also smaller for $N = 2$ up to $n = 24$, and up to $n = 8$ for $N = 1$, thanks to the way in which cycles are handled in our method. It should be recalled that the arithmetic method does not need to store a table at all, but this is at the cost of much higher computational complexity. The best compromise will depend on the application and hardware on which the method is implemented. For instance, the proposed method is very appealing for small CRCs or for correcting a high number of errors, while the arithmetic method may be more appealing for single error correction when long CRCs (e.g., CRC-32) are used.

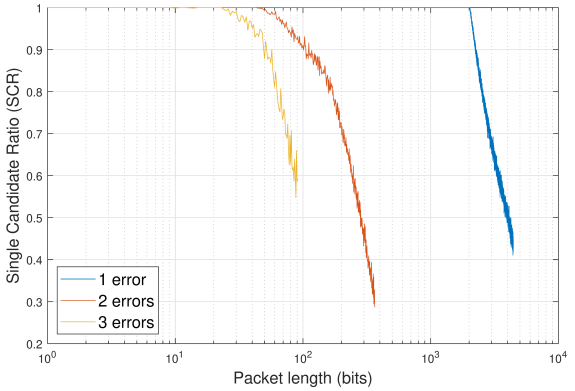
C. APPLICATION TO THE CORRECTION OF MULTIPLE ERRORS

The method proposed in this paper is able to output an exhaustive list of candidate error patterns having up to N errors. In practice, we assume that this method is applied in the context of relatively reliable communication channels, where corrupted packets tend to be mildly damaged. The authors in [34] demonstrated a realistic experimental scenario in which most damaged packets in a Bluetooth low energy (BLE) environment contained 3 errors or less. As our method is able to generate the list with a lower complexity, its use can be considered to increase the value of N to handle more error cases. However, the number of candidates increases significantly with N for a given packet length. In [18], we investigated the ratio of error patterns that would output a candidate list with a single entry, thus allowing the correction of the packet, over all possible error patterns for a given N, and termed this ratio the Single Candidate Ratio (SCR). We show that CRCs with low-degree generator polynomials never reach an SCR of 100% when considering multiple bit errors, and rapidly decrease as the packet length increases.

As an example, we propose to analyze the impact of a checksum cross-validation on the SCR for the generator polynomial used in CRC-8-CCITT. The SCR of this CRC is illustrated in Figure 10a. It can be seen that when considering a single error, up to 127 bits (i.e., $2^7 - 1$), the SCR is 100%,



(a) Single candidate ratio (SCR)



(b) Post-checksum single candidate ratio

FIGURE 10: Evolution of the SCR and post-checksum validation SCR for CRC-8-CCITT as a function of the packet length

which illustrates the importance of the $cycle$ length, as there is no list with multiple single error candidates for packets smaller than this $cycle$ value. The SCR then rapidly decreases and reaches 0% for packets greater than 245 bits. When considering double- and triple-bit error patterns, the SCR is very low, even for the smallest packet values considered. It reaches 0% for lengths of 26 and 6 bits, for double and triple error patterns, respectively.

Implementing a checksum validation step, as in the UDP and TCP protocols, helps to significantly increase such ratios and achieve decent error correction rates on small packets, even when using CRC-8-CCITT. In Figure 10b, it can be seen that the SCR is noticeably higher for all the numbers of errors considered. The SCR remains at 100% when dealing with packet lengths of up to 41 bits and 11 bits for double and triple error patterns, respectively. Moreover, it can be seen that the SCR is still over 50% for double error patterns, up to a packet length of 270 bits. For single error correction, the SCR, which reaches 0% for packet sizes larger than 245 bits in Figure 10a, remains at 100% up to a significant length of 2000 bits with checksum validation, which allows reconstruction of many more error cases. The method should perform even better with an actual BLE system with small packets (up to 39 bytes, and now increased to a maximum of 255 bytes) protected by a strong CRC-24.

Thus, when increasing the number of errors, validation steps such as a checksum cross-validation will help maintain a high correction rate when the candidate list size increases significantly. This allows to take advantage of the processing speed gains of the proposed method and to consider larger values of N. Increasing N brings a lot of changes and challenges in the literature methods as they are designed for a specific number of errors. We demonstrated that table-based CRC-ECEXP and CRC-ECIMP produce intractable table sizes from $N = 3$. The complexity of CRC-ECA methods is greatly affected whenever N is increased. The proposed CRC-ECOT replaces the arithmetic operations with successive table lookups, thus reducing the global complexity. As well, the table size is easily managed and constant for $N \geq 2$, which makes it very appealing for multiple error correction.

V. CONCLUSION AND PERSPECTIVES

In this paper, we propose an optimized table-based method for performing multiple error correction based on the CRC syndrome. The approach offers a low complexity alternative to the state-of-the-art error correction method as it generates a table that contains precomputed operations required to perform error pattern searches, avoiding most to all arithmetic operations.

Thanks to offline table generation, the proposed approach achieves the same error correction performance as state-of-the-art approaches while providing computational savings and thus improving the processing speeds. We show through simulations that the proposed method achieves significant speed gains over the table-free arithmetic method, and is between 2300× and 3000× faster when generating the list of double and single error patterns, respectively.

Thus, reducing the complexity offers the possibility of increasing the number of errors to consider, while keeping the same processing time. Since this greatly increases the number of candidates in the output error pattern list, we present a validation step that increases the correction rate for lists containing many candidates. Other validation steps could be used such as those based on bit error probability. Future work will look at integrating the proposed CRC-based error correction solution into a complete cross-layer receiver architecture in order to benefit from other validation mechanisms available in the protocol stack. The objective is to further reduce the list size or to be able to determine the best candidate out of several in order to reconstruct the best signal quality at the receiver side (e.g., visual quality, in the case of video content transmission).

REFERENCES

[1] A. Houghton, "Error coding for engineers," Springer, Boston, 2001.
[2] P. Koopman, "32-bit cyclic redundancy codes for Internet applications, "*Proceedings International Conference on Dependable Systems and Networks*, pp. 459-468, 2002.
[3] IEEE 802.11: Part 11: "Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," Dec. 2016.
[4] IEEE Standard Association, "IEEE 802.3-2018 - IEEE Standard for Ethernet," 2018, [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html
[5] J. Luo, K. D. Bowers, A. Oprea, and L. Xu, "Efficient software implementations of large finite fields GF(2^n) for secure storage applications," in *ACM Transactions on Storage*, vol. 8, no. 1, 27 pages, Feb. 2012.
[6] S. Lin and D.J. Costello, "Error control coding," Prentice-Hall, 1983.
[7] W.W. Peterson and W. E. Weldon, "Error-Correcting Codes," MIT Press, Second Edition, 1972.
[8] J. E. Meggitt, "Error Correcting Codes and Their Implementation," IRE Trans. Inf. Theory, IT-7, pp. 232-244, Oct. 1961.
[9] I. S. Reed, "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme," IRE Trans., IT-4, pp. 38-49, Sept. 1954.
[10] D. E. Muller, "Applications of Boolean Algebra to Switching Circuit Design and to Error Detection," IRE Trans., EC-3, pp. 6-12, Sept. 1954.
[11] J. L. Massey, Threshold Decoding, MIT Press, Cambridge, Mass. 1963.
[12] T. Kasami, L. Lin, and W. W. Peterson, "Some Results on Cyclic Codes Which Are Invariant under the Affine Group and Their Applications," Inf. Control, 2(5 and 6), pp. 475-496, Nov. 1968.
[13] 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.
[14] 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.
[15] A. S. Aiswarya and G. Anu, "Fixed Latency Serial Transceiver with Single Bit Error Correction on FPGA," in *2017 International Conference on trends in Electronics and Informatics (ICEI)*, 11-12 May 2017.
[16] 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, Dec. 2019.
[17] V. Boussard, "CRC-based error correction methods and algorithms applied to video communications over vehicular and IoT wireless networks," [Unpublished doctoral dissertation], Université Polytechnique Hauts-de-France / École de technologie supérieure, 2021.
[18] V. Boussard, S. Coulombe, F. Coudoux and P. Corlay, "Table-Free Multiple Bit-Error Correction Using the CRC Syndrome," *IEEE Access*, vol. 8, pp. 102357-102372, 2020.
[19] D. Hachenberger and D. Jungnickel, "Basis Representation and Arithmetics," in *Topics in Galois Fields. Algorithms and Computation in Mathematics*, vol. 29, pp. 355-425, Springer, 2020.

[20] F. Golaghazadeh, S. Coulombe, F.-X. Coudoux, P. Corlay, "Checksum Filtered List Decoding Applied to H.264 and H.265 Video Error Correction," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 8, pp. 1993-2006, Aug. 2018.
[21] V. Boussard, F. Golaghazadeh, S. Coulombe, F. Coudoux and P. Corlay, "Robust H.264 Video Decoding Using CRC-Based Single Error Correction and Non-Desynchronizing Bits Validation," in *2020 IEEE International Conference on Image Processing (ICIP)*, pp. 1098-1102, 2020.
[22] V. Boussard, F. Golaghazadeh, S. Coulombe, F. Coudoux and P. Corlay, "Enhanced CRC-based Correction of Multiple Errors with Candidate Validation," *Signal Processing: Image Communication*, vol.99, pp. 116475, ISSN 0923-5965, 2021.
[23] V. Boussard, S. Coulombe, F. Coudoux, P. Corlay and A. Trioux, "CRC-Based Multi-Error Correction of H.265 Encoded Videos in Wireless Communications," in *2021 IEEE Visual Communications and Image Processing (VCIP)*, pp. 1-5, 2021.
[24] Y. Zhang and Q. Yuan, "A multiple bits error correction method based on cyclic redundancy check codes," in *2008 9th International Conference on Signal Processing*, pp. 1808-1810, 2008.
[25] F. Golaghazadeh, S. Coulombe, F. Coudoux and P. Corlay, "The Impact of H.264 Non-Desynchronizing Bits on Visual Quality and its Application to Robust Video Decoding," in *2018 12th International Conference on Signal Processing and Communication Systems (ICSPCS)*, Cairns, Australia, pp. 1-7, 2018.
[26] A. Demirtas, A. Reibman, and H. Jafarkhani, "Performance of H.264 with isolated bit error: Packet decode or discard?" in *Proceedings IEEE 18th International Conference on Image Processing (ICIP)*, pp. 949–952, 2011.
[27] D. Hachenberger and D. Jungnickel, "Shift Register Sequences," in *Topics in Galois Fields. Algorithms and Computation in Mathematics*, vol. 29, pp. 427-487, Springer, 2020.
[28] Y. Li, Z. Yang, Z. Li et al. "A New Algorithm on the Minimal Rational Fraction Representation of Feedback with Carry Shift Registers," in *Design, Codes and Cryptography*, vol. 88, pp. 533-552, 2020.
[29] S. K. Tripathi, B. Gupta, K. K. S. Pandian, "The Shortest Register With Non-Linear Update for Generating a Given Finite or Periodic Sequence," in *IEEE Communications Letters*, vol. 24, no. 6, pp. 1173-1177, 2020.
[30] U. Jetzek, "Galois Fields, Linear Feedback Shift Registers and their Applications," Carl Hanser Verlag GmbH & Company KG, 2018.
[31] RaspberryPI4ModelB.[Online].Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.
[32] "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.
[33] Specification of the Bluetooth system. Core Version 4.1, Bluetooth SIG,2013. [Online]. Available: http://www.bluetooth.com.
[34] E. Tsimbalo, X. Fafoutis, R. J. Piechocki, "CRC Error Correction in IoT Applications," *IEEE Transactions on Industrial Informatics*, vol. 13, no.1, pp. 361-369, Feb. 2017.

VIVIEN BOUSSARD received the B.Sc. and M.Sc. degrees in broadcast engineering from Université Polytechnique Hauts-de-France, Valenciennes, France, in 2015 and 2017, respectively, and the Ph.D. degree in engineering from École de Technologie Supérieure, Montréal, Canada, and Université Polytechnique Hauts-de-France, Valenciennes, France, in 2021. He joined Dacast, in 2021, where his current research interests include the optimization of video communication systems.

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.