CRC and Hamming Codes - Basic Examples

Daniel E. Lucani

September 11, 2019

1 Cyclic Redundancy Check (CRC)

Let us consider a block of data B with n bits, where B can be expressed as a polynomial $B(x) = b_0x^0 + b_1x^1 + ... + b_{n-1}x^{n-1}$ and b_i is the i-th bit of B. We consider b_{n-1} as the Most Significant Bit (MSB) of B. Computing a `CRC-m` (i.e., m bits long) requires computing the long division of $B(x)$ with a generator polynomial $g(x) = g_0x^0 + g_1x^1 + ... + g_mx^{m1}$. The residue of this division is the `CRC` value.

One of the properties of CRC's is that

$$\texttt{CRC}(A \oplus B) = \texttt{CRC}(A) \oplus \texttt{CRC}(B).$$

This means that pre-computing all n-bit sequences with a single non-zero bit allows us to compute any sequence of n-bits by XORing the appropriate `CRC` sequences. More explicitly,

$$\begin{aligned} \texttt{CRC}(B) = \quad & b_{n-1}\texttt{CRC}(1000...0) \oplus b_{n-2}\texttt{CRC}(0100...0)\oplus && (1)\\ & b_{n-3}\texttt{CRC}(0010...0) \oplus b_{n-4}\texttt{CRC}(0001...0)\oplus && (2)\\ ... & && (3)\\ & b_1\texttt{CRC}(00...010) \oplus b_0\texttt{CRC}(00...001)\oplus && (4) \end{aligned}$$

1.1 CRC-3

For a `CRC-3`, the generator polynomial is $g^3(x) = x^3 + x + 1$. Let us consider $n = 7$ bit sequences with a single non-zero bit first. Note that a typical `CRC` accepts any n as input. Our choice is strategic to allow us to do the comparison to Hamming codes. However, the same procedure applies for a general n.

$$\begin{aligned} & (0000001) \to x^0 && (5)\\ \Longrightarrow\ & \texttt{CRC}(0000001) = x^0/g^3(x) && (6)\\ \Longrightarrow\ & \texttt{CRC}(0000001) = x^0 \to (001) && (7) \end{aligned}$$

$$\begin{aligned} & (0000010) \to x^1 && (8)\\ \Longrightarrow\ & \texttt{CRC}(0000010) = x^1/g^3(x) && (9)\\ \Longrightarrow\ & \texttt{CRC}(0000010) = x^1 \to (010) && (10) \end{aligned}$$

[1]we will use + and $\oplus$ interchangeably here.

$$(0000100) \rightarrow x^2 \tag{11}$$
$$\implies \texttt{CRC}(0000100) = x^2/g^3(x) \tag{12}$$
$$\implies \texttt{CRC}(0000100) = x^2 \rightarrow (100) \tag{13}$$

$$(0001000) \rightarrow x^3 \tag{14}$$
$$\implies \texttt{CRC}(0001000) = x^3/g^3(x) = x^3/(x^3+x+1) \tag{15}$$
$$\implies \texttt{CRC}(0001000) = x+1 \rightarrow (011) \tag{16}$$

because

$$\begin{array}{r|l} x^3 & x^3+x+1 \\ \cline{2-2} x^3+x+1 & 1 \\ \hline \mathbf{x+1} & \end{array}$$

$$(0010000) \rightarrow x^4 \tag{17}$$
$$\implies \texttt{CRC}(0010000) = x^4/g^3(x) = x^4/(x^3+x+1) \tag{18}$$
$$\implies \texttt{CRC}(0010000) = x^2+x \rightarrow (110) \tag{19}$$

because

$$\begin{array}{r|l} x^4 & x^3+x+1 \\ \cline{2-2} x^4+x^2+x & x \\ \hline x^2+x & \end{array}$$

$$(0100000) \rightarrow x^5 \tag{20}$$
$$\implies \texttt{CRC}(0100000) = x^5/g^3(x) = x^5/(x^3+x+1) \tag{21}$$
$$\implies \texttt{CRC}(0100000) = x^2+x+1 \rightarrow (111) \tag{22}$$

because

$$\begin{array}{r|l} x^5 & x^3+x+1 \\ \cline{2-2} x^5+x^3+x^2 & x^2 \qquad +1 \\ \cline{1-1} x^3+x^2 & \\ x^3 \qquad +x+1 & \\ \cline{1-1} x^2+x+1 & \end{array}$$

Finally, we calculate the CRC value for the sequence (1000000) that corresponds to the MSB as

$$(1000000) \rightarrow x^6 \tag{23}$$
$$\implies \texttt{CRC}(1000000) = x^6/g^3(x) = x^6/(x^3+x+1) \tag{24}$$
$$\implies \texttt{CRC}(1000000) = x^2+1 \rightarrow (101) \tag{25}$$

$$
\begin{array}{l|l}
x^6 & x^3+x+1 \\ \cline{2-2}
x^6+x^4+x^3 & x^3 \quad +x+1 \\ \cline{1-1}
\quad x^4+x^3 & \\
\quad x^4 \quad +x^2+x & \\ \cline{1-1}
\qquad x^3+x^2+x & \\
\qquad x^3 \quad +x+1 & \\ \cline{1-1}
\qquad\quad x^2 \quad +1 &
\end{array}
$$

Polynomial	Bit Sequence	`CRC-3`
x^0	(0000001)	(001)
x^1	(0000010)	(010)
x^2	(0000100)	(100)
x^3	(0001000)	(011)
x^4	(0010000)	(110)
x^5	(0100000)	(111)
x^6	(1000000)	(101)

Table 1: Bit sequences with one non-zero bit and `CRC-3` associated to it

We can summarize the `CRC-3` results in Table 1.1.

Computing the CRC for other 7-bit sequences: This means that a sequence (1001001) will have

$$
\begin{aligned}
\texttt{CRC}(1001001) = & \quad \texttt{CRC}(1000000) \oplus \texttt{CRC}(0001000) \oplus \texttt{CRC}(0000001) && (26)\\
= & \quad (101) \oplus (011) \oplus (001) && (27)\\
= & \quad (111). && (28)
\end{aligned}
$$

For sanity check, we can calculate the `CRC-3` using the standard procedure (long division) of the sequence $(1001001) \rightarrow x^6 + x^3 + 1$ with $g^3(x)$ such that

$$
\begin{array}{l|l}
x^6 \quad +x^3 \qquad +1 & x^3+x+1 \\ \cline{2-2}
x^6+x^4+x^3 & x^3 \qquad +x \\ \cline{1-1}
\quad x^4 \qquad +1 & \\
\quad x^4 \quad +x^2+x & \\ \cline{1-1}
\qquad\quad x^2+x+1 &
\end{array}
$$

which results in a residue of $x^2 + x + 1 \rightarrow (111)$ as expected.

More generally, for any input sequence $B(x) = b_0x^0 + b_1x^1 + ... + b_6x^6$ or $B = (b_0, ..., b_6)$, we can use Eq. 1:

$$
\begin{aligned}
\texttt{CRC}(B) = & \quad b_0\texttt{CRC}(0000001) \oplus b_1\texttt{CRC}(0000010) \oplus b_2\texttt{CRC}(0000100) && (29)\\
& \oplus b_3\texttt{CRC}(0001000) \oplus b_4\texttt{CRC}(0010000) \oplus b_5\texttt{CRC}(0100000) \oplus b_6\texttt{CRC}(1000000) && (30)\\
= & \quad b_0(001) \oplus b_1(010) \oplus b_2(100) \oplus b_3(011) \oplus b_4(110) \oplus b_5(111) \oplus b_6(101) && (31)\\
= & \quad (00b_0) \oplus (0b_10) \oplus (b_200) \oplus (0b_3b_3) \oplus (b_4b_40) \oplus (b_5b_5b_5) \oplus (b_60b_6). && (32)
\end{aligned}
$$

Matrix perspective: Another way to understand this is that for any input sequence $B(x) = b_0x^0 + b_1x^1 + ... + b_6x^6$ or $B = (b_6, ..., b_0)$, we can calculate the $CRC - 3$ as a vector $C = (c_0, c_1, c_2)$:

$$C = B\boldsymbol{H} \tag{33}$$

where $\boldsymbol{H}$ corresponds to the elements of `CRC-3` in the above table:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{34}$$

Connection to Tofino Switches: $\boldsymbol{H}$ is the matrix of values that the Tofino switch will load into the hardware components when given a polynomial $g(x) = x^3 + x + 1$ (or $0x3$ for the function call input). The Tofino switch will use a similar technique as depicted above (i.e., computing the CRC's for a sequence of n bits with only 1 non-zero element) to compute the values. This is likely done in the background (i.e., hidden to the user of the CRC function).

2 Hamming Codes

Hamming Codes are block codes that convert k bit messages into n bit messages by adding m parity bits. More specifically, $n = 2^m - 1$ bits and $k = n - m = 2^m - m - 1$ bits. Hamming codes are generated using a generator matrix $\boldsymbol{G}$ as follows

$$\boldsymbol{G} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}$$

where the size of the matrix is $k \times n$.

Connection to CRC #1: a particular construction of vnG can be achieved using a generator polynomial and the fact that Hamming codes are *cyclic codes*. This means that each row of the vnG can be generated by a shifted version of the previous row.

Example: For the case of a $(n, k) = (7, 4)$ code that has $m = 3$ parity bits, the generator polynomial is $g^3(x) = g_3x^3 + g_2x^2 + g_1x^1 + g_0x^0 = x^3 + x + 1$ (or $(1, 0, 1, 1)$), the same generator polynomial used for `CRC-3`. In this case,

$$\boldsymbol{G} = \begin{bmatrix} g_3 & g_2 & g_1 & g_0 & 0 & 0 & 0 \\ 0 & g_3 & g_2 & g_1 & g_0 & 0 & 0 \\ 0 & 0 & g_3 & g_2 & g_1 & g_0 & 0 \\ 0 & 0 & 0 & g_3 & g_2 & g_1 & g_0 \end{bmatrix}.$$

which means

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}.$$

Such a code can be transformed to *systematic* form, where the original message is embedded directly into the codeword and the parity bits are clearly separated from it. This is achieved by performing row operations to eliminate components until the first k columns are equivalent to an identity matrix. In general, this looks like

$$\boldsymbol{G}_s = \begin{bmatrix} 1 & 0 & \cdots & 0 & p_{0,0} & p_{0,1} & \cdots & p_{0,n-k-1} \\ 0 & 1 & \cdots & 0 & p_{1,0} & p_{1,1} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{I}_k & \boldsymbol{P} \end{bmatrix}.$$

Example: For the case of a $(n, k) = (7, 4)$, its systematic matrix form is obtained by eliminating the effects of the lower two rows on the first two rows. The resulting matrix is

$$\boldsymbol{G}_s = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

where the *parity matrix* $\boldsymbol{P}$ of size $k \times m$ is

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

A systematic codeword allows for easy decoding, since it is immediate what the message was, assuming that no errors occurred. This will be the convention we use in our implementations and future dicusssions.

The *parity-check matrix*, $\boldsymbol{H}$ of size $m \times n$:

$$\boldsymbol{H} = \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k-1,0} & h_{n-k-1,1} & \cdots & h_{n-k-1,n-1} \end{bmatrix},$$

or in systematic form,

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{P}^T & \boldsymbol{I}_{n-k} \end{bmatrix}.$$

Example: For the case of a $(n, k) = (7, 4)$, $\boldsymbol{H}$ is given by

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

or alternatively

$$\boldsymbol{H}^T = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

which is the same as in the $CRC - 3$ case studied before for $n = 7$ bits.

The pair of generator matrix and parity-check matrix is related as:

$$\boldsymbol{G}\boldsymbol{H}^T = \mathbf{0}.$$

The message $\boldsymbol{m}$ $(1 \times k)$ is encoded into the codeword $\boldsymbol{c}$ $(1 \times n)$ by

$$\boldsymbol{c} = \boldsymbol{m}\boldsymbol{G}.$$

For decoding, the received sequence is $\boldsymbol{B} = \boldsymbol{c} + \boldsymbol{e}$, where $\boldsymbol{e}$ is the error pattern that occurred when transitting the data (all-zero $\boldsymbol{e}$ means there are no errors). The matrix $\boldsymbol{H}$ can be used to detect whether any errors occured, by calculating the *syndrome* vector, $\boldsymbol{s}$:

$$\boldsymbol{s} = \boldsymbol{B}\boldsymbol{H}^T = (\boldsymbol{c} + \boldsymbol{e})\boldsymbol{H}^T = \boldsymbol{c}\boldsymbol{H}^T + \boldsymbol{e}\boldsymbol{H}^T = \boldsymbol{m}\boldsymbol{G}\boldsymbol{H}^T + \boldsymbol{e}\boldsymbol{H}^T = \boldsymbol{e}\boldsymbol{H}^T.$$

The syndrome vector is all-zero if no errors occurred. It specifies the error that happened, and may be used to correct that error, e.g. by using a look-up table of the associated error in a syndrome-error table. Note that there are many other decoding algorithms.

In order to determine the bit in error and its associated syndrome, we can simply calculate $\boldsymbol{s}$ for all $\boldsymbol{e}$ with a single non-zero bit. For example, $\boldsymbol{e} = (0000001)$ results in

$$\boldsymbol{e}\boldsymbol{H}^T = (0,0,0,0,0,0,1) \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (0,0,1)$$

For $\boldsymbol{e} = (1000000)$ results in

$$\boldsymbol{e}\boldsymbol{H}^T = (1,0,0,0,0,0,0) \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (1,0,1)$$

Basically, the position of the bit in error has a direct correspondance to the row in $\boldsymbol{H}^T$ and that row provides the specific syndrome obtained. This result can be summarized as Table 2. Interestingly, this table has a one-to-one correspondence with the table for `CRC-3`. It also tells us that to calculate $\boldsymbol{B}\boldsymbol{H}^T$ we can use exactly the same procedure as for `CRC-3`, basically, XORing the syndrome values (bit-by-bit) that correspond to each non-zero element in $\boldsymbol{B}$.

Remark: More generally, as long as (a) the generator polynomial for Hamming is used as parameter for the CRC generator polynomial and (b) the size of the input data to be computed is $n = 2^m - 1$ bits as consistent with a Hamming code, then the same conclusions will be valid for other `CRC-m` and the corresponding $(n, k) = (2^m - 1, 2^m - m - 1)$ Hamming code.

Other generator Polynomials for Hamming Codes:

Bit in error	Bit Sequence	Syndrome
0	(0000001)	(001)
1	(0000010)	(010)
2	(0000100)	(100)
3	(0001000)	(011)
4	(0010000)	(110)
5	(0100000)	(111)
6	(1000000)	(101)

Table 2: Hamming Code $(7, 4)$ Syndromes

Code	Generator Polynomial	Parameter for Tofino	Corresponding `CRC-m`
$(7, 4)$	$x^3 + x + 1$	$0x3$	CRC-3-GSM
$(15, 11)$	$x^4 + x + 1$	$0x3$	CRC-4-ITU
$(31, 26)$	$x^5 + x^2 + 1$	$0x05$	CRC-5-USB
$(63, 57)$	$x^6 + x + 1$	$0x03$	CRC-6-ITU
$(127, 120)$	$x^7 + x^3 + 1$	$0x09$	CRC-7
$(255, 247)$	$x^8 + x^4 + x^3 + x^2 + 1$	$0x1D$	CRC-8-SAE J1850

Table 3: Generator Polynomials

3 Generalized Deduplication (GDD)

In the scenario of approximate deduplication, the process is reversed to that presented for Hamming codes. Meaning, the encoding (transformation) process of GDD corresponds to Hamming decoding, while the decoding (inverse transformation) process corresponds to Hamming encoding.

3.1 GDD Encoding

The original data to be transformed is $\boldsymbol{B} = (b_{n-1}, b_{n-2}, ..., b_0)$ and will be converted into a pair $(\boldsymbol{m}, \boldsymbol{s})$ of *basis* and *deviation*, $\boldsymbol{m} = (m_{k-1}, ..., m_0)$ and $\boldsymbol{s} = (s_{m-1}, ..., s_0)$, which correspond to the message and syndrome of Hamming codes, respectively.

Let us assume that the code we use is systematic.

`STEP 1 (Calculate Syndrome/deviation):`

To achieve this, we compute

$$\boldsymbol{s} = \boldsymbol{B}\boldsymbol{H}^T \tag{35}$$

Remark: This step can be performed in the Tofino switches by using the appropriate `CRC` on the data. This will directly produce the syndrome/deviation needed.

`STEP 2 (Use Syndrome to Calculate the Basis):`

If $\boldsymbol{s} = \boldsymbol{0}$, then $(m_{k-1}, ..., m_0) = (b_{n-1}, b_{n-2}, ..., b_{n-1-k})$. If $\boldsymbol{s} \neq \boldsymbol{0}$, then it is key to identify the bit to be flipped in the original $\boldsymbol{B}$ sequence.

More specifically, this can be done by constructing a look-up table, such as the one in Table 2, and adding the case of a syndrome of all-zero bits (e.g., $\boldsymbol{s} = (000)$) with a bit sequence of (0000000) to apply to the original data. This allows us to have a single path for processing all data.

Using the value already computed for $\boldsymbol{s}$, we can identify the error bit sequence associated to it. After applying the bit error sequence $\boldsymbol{e}$ to $\boldsymbol{B}$ we obtain $\boldsymbol{B'} = \boldsymbol{e} \oplus \boldsymbol{B}$. Then, we can assign the $(m_{k-1}, ..., m_0) = (b'_{n-1}, b'_{n-2}, ..., b'_{n-1-k})$. Basically, reading out the first (MSB) k bits as the basis.

Example: Let us assume the system receives $\boldsymbol{B} = (0000001)$. Then,

$$\boldsymbol{s} = \boldsymbol{B}\boldsymbol{H}^T = (0,0,0,0,0,0,1)\begin{bmatrix}1&0&1\\1&1&1\\1&1&0\\0&1&1\\1&0&0\\0&1&0\\0&0&1\end{bmatrix} = (0,0,1)$$

This means that $\boldsymbol{s} = (001)$. Using Table 2, we identify that the bit sequence to correct the bit error is $\boldsymbol{e} = (0000001)$. We can then proceed to XOR this error to the received sequence

$$\boldsymbol{e} \oplus \boldsymbol{B} = (0000000)$$

. From this, it is clear that the basis will be the first four bits $(0,0,0,0)$.

Remark: Note that for our implementation we do not really need to have all values on the table. A table with only k-bit sequences (instead of n-bit sequences) would be sufficient to contain the useful cases that require a modification of the k MSBs. A number of syndromes only require a change to the parity bits, which is not really interesting for determining the basis.

Remark: If we need to reduce memory usage, we can introduce logic that does not store the sequences themselves, but only the position to be flipped in $\boldsymbol{B}$. This would significantly reduce the amount of memory used in the table. Basically, instead of storing n bits for each sequence of the total $n+1$ syndrome values for a total of $\approx n^2 + n$ bits, we can store m bits to signal the position of the bit error for all $n+1$ syndromes, for a total of $\approx mn + m$ bits. As a first example, for $m = 7$ we can have $n = 2^7 - 1 = 127$-bit chunks of data and the raw look up table would require roughly 1399 bits in size while the improved version would require 896 bits. For $m = 13$ bits (chunks of 1 KB of data), it would require 67100673 bits (8+ MB) while the other approach would require only 106496 bits (13.3 KB).

`STEP 3 (Calculate ID of Basis):`

In order to reduce the amount of data sent, we calculate the ID for the basis. This is performed using a hash or `CRC`.

For the Tofino switches, we can rely on a sufficiently large `CRC`, e.g., `CRC-32` or larger if needed.

Remark: Clearly, gains can only occur if the size of the ID is smaller than the size of the basis being processed. For example, for a `CRC-32`, we would have 32 bits of ID. This would only provide gains for transformations with Hamming codes of $m > 5$ bits. For example, $m = 6$ bits corresponds to chunks of 63 bits and basis of size 57 bits. Thus, instead of sending 63 bits we would send $32 + 6 = 38$ bits if there is a potential to compress. Thus, the top gain for this would be around 65 %. For $m = 7$ and chunks of 127 bits, the gain can be as much as $3,25$ times (less bits transmitted).

For $m = 13$ bits, the potential is larger. Instead of sending 8191 bits, we would be able to send $32 + 13 = 45$ bits when there is a chance to compress. This means a top gain of 182+ times. Of course, the requirement to achieve this gain is larger, meaning, more data needs to be stored in the other end (e.g., other switch or Cloud) to take advantage of it. Keeping records of small chunks of 63 or 127 bits is typically simpler and more matches are expected.

This is a parameter that needs to be evaluated.

Further optimization: The switch can also use a different ID instead of the `CRC-32` value. For example, the switch and its intended destination could agree to have a link between the CRC value and a specific ID of smaller size. If the switch can store 2^{16} ID's, the destination could assign a 16 bit to each 32-bit CRC value. This means that the process and gains could be higher. Using our previous example, the top gains for $m = 6, 7, 13$ would be $2,86$ times, $5,52$ times and 282 times, respectively.

Similarly, if the switch could only hold 2^{10} values in the table for a given source, only 10 bits of ID are necessary.

This approach can be particularly good if we would like to use CRC's or hashes with larger number of bits but will like to keep the overhead low.

```
STEP 4 (Determine if ID is in the list of IDs of the switch
for the specific destination):
```

This is the simplest mechanism to use. Basically, the switch keeps track of the list of IDs already available at the intended destination. If the destination has them, they can undo the operation. This list could be updated/modified over time (but in a much longer time frame than typical operations). The switches operation would be stateless from this perspective.

In fact, for the switch the operation would require keeping a look-up table with IDs of 32 bits (for `CRC-32`). Depending on switch memory and other requirements (e.g., maximum number of destinations active, more commonly used destinations, destinations with the most traffic), we can determine the size of the look-up per destination.

There are other mechanisms to do this step, but require additional memory to be stored in the switch.

3.2 GDD Decoding

The objective of GDD decoding is to transform from a $(\boldsymbol{m}, \boldsymbol{s})$ representation or $(ID_m, \boldsymbol{s})$ representation to the original data $\boldsymbol{B}$. Let us assume we start in form $(ID_m, \boldsymbol{s})$, which is the most general and only one conducent to reduction of traffic.

STEP 1 (Search for $\boldsymbol{m}$ based on ID):

The encoder only transforms to a given ID value if it knows that the receiver is able to convert back this step. Basically, the receiver needs to have a look-up table that matches the ID to a $\boldsymbol{m}$ value of k bits.

STEP 2 (Transform basis $\boldsymbol{m}$ to error-free data chunk):

The next step is to use $\boldsymbol{m}$ and multiply it with the generator matrix G_s to produce the error-free data chunk $\boldsymbol{B}'$:

$$\boldsymbol{B}' = \boldsymbol{m}\boldsymbol{G}_s.$$

From a practical perspective, it is only required for the system to perform a product of the $\boldsymbol{m}$ with the parity matrix P and append it to the bits in $\boldsymbol{m}$. The reason is that $\boldsymbol{G}_s$ corresponds to a systematic code, which means that the k most significant bits of $\boldsymbol{B}'$ are exactly the k-bits in $\boldsymbol{m}$.

In other words,

$$\boldsymbol{B}' = [\boldsymbol{m}, \boldsymbol{p}]$$

, where $\boldsymbol{p}$ is the vector of size $1 \times m$ containing the added parity bits, and where

$$\boldsymbol{p} = \boldsymbol{m}\boldsymbol{P}.$$

Implementation in Tofino: To implement this in a simple fashion in the Tofino switch by using the `CRC-m` primitives, this becomes quite simple. In fact, it is equivalent to the encoding step. The main change is that we would input to the system not only the $\boldsymbol{m}$ sequence but we would zero pad it with m additional zeros.

As an example for the $(7, 4)$ code, if we received $\boldsymbol{m} = (1111)$ we could compute the `CRC-3` with the same polynomial as for the encoding but receiving as input the sequence $[\boldsymbol{m}, 0, 0, 0] \rightarrow$ (1111000). The effect of the last three bits is zero.

$$[\boldsymbol{m}, 0, 0, ..., 0]\boldsymbol{H}^T = (1, 1, 1, 1, 0, 0, 0) \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (1, 1, 1)$$

The m-bit output from the `CRC` (in our example (111)) is then used to fill the last m bits of the sequence. In the previous example, this means that the input to the CRC function $((1, 1, 1, 1, 0, 0, 0))$ produces (111) and these are replaced in the last $m = 3$ bits to generate $\boldsymbol{B}' = (1, 1, 1, 1, 1, 1, 1)$.

`STEP 3 (Use syndrome` $\boldsymbol{s}$ `to apply the necessary modification):`

Using the value $\boldsymbol{s}$, we can search in Table 2 to find the bit that requires flipping in the $\boldsymbol{B}'$ sequence computed in Step 2.

For example, if $\boldsymbol{s} = (101)$, this means that the MSB needs to be flipped in $\boldsymbol{B}'$ order to recover the orignal data $\boldsymbol{B}$. This means that $\boldsymbol{B} = (0111111)$.

This step and its considerations is highly related to Step 2 in the encoding process.