Towards Synthesis of Application-Specific Forward Error Correction (FEC) Codes

Jedidiah McClurg
Colorado State University
USA

Lauren Zoe Baker*
Colorado School of Mines
USA

Ronaldo Canizales*
Colorado State University
USA

Dilochan Karki*
Colorado State University
USA

ABSTRACT

Forward error correction (FEC) is a key component of modern high-bandwidth networks. Typically implemented at the physical layer, FEC attaches error-correcting codes to blocks of transmitted data, allowing some corrupted blocks to be repaired without retransmission. We outline a synthesis-based approach for automatic exploration of the FEC-code design space, focusing on Hamming codes. We formally verify the correctness of a Hamming (128, 120) code used for FEC in the recent 802.3df Ethernet standard, and provide preliminary evidence that our prototype synthesizer can leverage user-provided formal properties to generate FEC codes that are highly robust, efficiently implementable, and tuned to support specific data formats such as IEEE floating points.

CCS CONCEPTS

• **Networks** → **Error detection and error correction**; **Programmable networks**.

KEYWORDS

forward error correction, program synthesis

ACM Reference Format:
Jedidiah McClurg, Lauren Zoe Baker, Ronaldo Canizales, and Dilochan Karki. 2024. Towards Synthesis of Application-Specific Forward Error Correction (FEC) Codes. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696886

*Equal contribution

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696886

1 INTRODUCTION

Block codes such as Hamming [13], Reed-Solomon [33], low-density parity check (LDPC) [10], etc. have been used for error detection and correction in communication and data storage since the advent of digital computing. Recently, these types of block codes are making a strong resurgence in the form of *forward error correction (FEC)*, which has become a key component of modern networks as speeds increase in optical and cellular links. For example, a (128, 120) Hamming code (120-bit data plus 8-bit check) has been extensively analyzed [3, 29, 31] and adopted alongside KP4 (a Reed-Solomon code) in the recent 802.3df standard (400- and 800-Gb/s Ethernet). A huge amount of manual design went into constructing this Hamming FEC code, including low-level analysis related to hardware implementation [4, 40], but detailed arguments were also made in favor of an alternative Reed-Solomon FEC code [39]. The questions that motivated this paper were twofold: (1) ***How can we provide formal guarantees about the "right" FEC code?*** (2) ***How can we help to automate the FEC design process?***

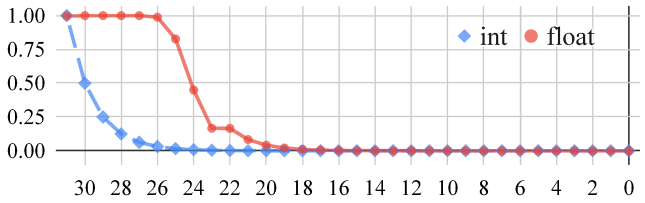

Figure 1: Avg. magnitude of num. error vs. bit position

A key observation is that the "right" code is heavily dependent on the type of data being transmitted. Consider Figure 1, which shows the (normalized) average magnitude of numeric error across all possible numeric 32-bit integers and floating points when a bit at each index is flipped. This suggests that a floating-point-specific FEC code may be able to better balance between robustness and computation/transmission overhead by focusing primarily on the upper 8 bits,

especially since many distributed applications have some degree of tolerance to low-magnitude numeric error, e.g., certain scientific computing simulations [19], distributed machine learning (ML) training [27], etc. Conceivably, we could use one FEC code for integers and another for floating points, tuned to their per-bit characteristics, and existing work could allow us to dynamically exchange codes [18].

In this paper, we outline a prototype system which is the first to use counterexample-guided inductive synthesis (CEGIS) to automatically explore the FEC design space, focusing specifically on Hamming codes. Our tool supports user-specifiable properties, allowing application-specific design of FECs, and can function as both a verifier and synthesizer. For example, we can efficiently verify that the 802.3df (128,120) Hamming code has minimum distance 3. We show that our tool can automatically extend the standard Hamming (7,4) code to much higher robustness, while minimizing the necessary additional check bits. We synthesize a standard Hamming code with minimum distance 3 that is usable for 32-bit data, and synthesize an alternative floating-point-specific code with increased overall error rate, but drastically decreased average magnitude of error, as well as significantly fewer check bits. Finally, we show how our tool can *optimize* codes by reducing the number of used bits, and demonstrate the encode/check performance improvement and compressibility advantages of this optimization.

2 BACKGROUND

Many codes have been proposed for FEC (see [37] for an overview of FEC and survey of specific codes). In this paper, we focus on Hamming codes, but the techniques developed here could be readily applied to other types of block codes.

2.1 Hamming Codes

The following error detection and correction method was first described by Hamming [13]. Although the coding scheme can be applied in arbitrary finite fields, the 2-element finite field GF(2) is most commonly used, due to the prevalence of binary data. An (n, k) Hamming code encodes each k-bit *data word* $\vec{d}$ into an n-bit *codeword* $\vec{w}$. This computation is performed via matrix multiplication $\vec{w} = \vec{d}\,G$. The $k \times n$ *generator matrix* G is of the form $G = (I_k \mid P)$, where I_k is the $k \times k$ identity matrix, and P is a $k \times (n-k)$ *coefficient matrix* that determines the $(n-k)$ *check bits* within the codeword. In this paper, we will denote a generator matrix as G_c^k, where k is the data length, and $c = (n-k)$ is the number of check bits. To determine whether a codeword $\vec{w}$ has been corrupted, we perform matrix multiplication $\vec{b} = (H\,\vec{w}^T)^T$, where $H = (-P^T \mid I_{(n-k)})$ is a $(n-k) \times n$ *check matrix*. In the finite field GF(2), $-P^T = P^T$, i.e., simply the transpose of the coefficient matrix P. If $\vec{b} = \vec{0}$, the codeword is *valid* (no error detected). If $\vec{b}^T$ equals a column j of H, this indicates a single error at position j of the codeword, allowing error correction. Multi-bit errors can be detected but not corrected.

$$\begin{pmatrix}0&0&1&1\end{pmatrix}\left(\begin{array}{cccc|ccc}1&0&0&0&1&0&1\\0&1&0&0&1&1&0\\0&0&1&0&1&1&1\\0&0&0&1&0&1&1\end{array}\right)=\left(\begin{array}{cccc|ccc}0&0&1&1&1&0&0\end{array}\right)$$

$$\left(\begin{array}{cccc|ccc}1&1&1&0&1&0&0\\0&1&1&1&0&1&0\\1&0&1&1&0&0&1\end{array}\right)\left(\begin{array}{c}0\\0\\1\\1\\\hline 1\\0\\0\end{array}\right)=\begin{pmatrix}0\\0\\0\end{pmatrix}$$

Figure 2: Hamming encode ($\vec{d}\,G = \vec{w}$) and check ($H\,\vec{w}^T = \vec{b}^T$) example calculations

Fig. 2 shows how Hamming $(7, 4)$ encoding works. Here, 4-bit data word $\vec{d}$ is multiplied by generator G_3^4 to form 7-bit codeword $\vec{w}$. Note that in GF(2), *addition* behaves as `xor`, and *multiplication* behaves as `and`, so to produce the first bit of the codeword, we bitwise-`and` data `0011` with the first generator column `1000` to get `0000`, then `xor` across these bits to get $(((0 \oplus 0) \oplus 0) \oplus 0) \oplus 0 = 0$ (this `xor` operation produces `1` when an *odd* number of bits are set). To check codeword $\vec{w}$, we multiply $H\,\vec{w}^T$ to get 3-bit check bits $\vec{b}^T$.

2.2 Hamming Code Robustness

An *undetectable error* occurs when bit-errors have altered a valid codeword into another valid codeword. The *minimum distance* $md(G)$ of a generator G is the smallest number of bit-errors needed to transform any valid codeword into another valid codeword, i.e., the smallest number of bit errors needed to produce an undetectable error. For example, the Figure 2 generator G_3^4 has minimum distance 3. The probability P_u of an undetected error is given by the formula

$$P_u(G_c^k) = \sum_{j=m}^{n} \binom{n}{j} \cdot p^j \cdot (1-p)^{n-j} \approx \binom{n}{m} \cdot p^m,$$

where $n = k + c$ is the codeword length, $m = md(G)$ is the minimum distance, and p is the channel bit-error probability. Altering characteristics of the generator such as the number of check bits and minimum distance can change its *robustness*, i.e., its likelihood of detecting errors. In Section 3, we describe an automated approach for producing robust generators, and provide preliminary experimental evidence demonstrating the utility of this approach in Section 4.

$$
\begin{array}{rcll}
n & \in & \mathbb{Z} & \textit{(integer constant)} \\
r & \in & \mathbb{R} & \textit{(real constant)} \\
f & ::= & \texttt{len}_\texttt{d} \mid \texttt{len}_\texttt{c} \mid \texttt{len}_1 \mid \texttt{md} & \textit{(function)} \\
\phi & ::= & \texttt{true} \mid \texttt{false} \mid c \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi & \textit{(property)} \\
 & & \mid \phi \Rightarrow \phi \mid \texttt{minimal}(e) \mid \texttt{maximal}(e) & \\
e & ::= & n \mid r \mid e + e \mid e - e \mid e * e \mid -e & \textit{(expression)} \\
 & & \mid \texttt{G}_e(e,e) \mid \texttt{len}_\mathcal{G} \mid \texttt{len}_\texttt{w} \mid \texttt{w}(e) \mid \texttt{sum}_\texttt{w} \mid f(\texttt{G}_e) & \\
c & ::= & e \neq e \mid e = e \mid e > e \mid e < e & \textit{(condition)}
\end{array}
$$

Figure 3: Property language for Hamming generators

3 SPECIFYING AND SYNTHESIZING GENERATORS

We outline a program synthesis approach to automate the construction of application-specific Hamming codes.

3.1 Property Language

Figure 3 details our specification language for properties of generators. Numeric expressions e include value at a specific location within a specific generator $\texttt{G}_e(e,e)$, number of generators $\texttt{len}_\mathcal{G}$ (we use $\mathcal{G}$ to represent the set of all generators), number of weights $\texttt{len}_\texttt{w}$ (discussed in §3.2), value of a specific weight $\texttt{w}(e)$, weighted sum of bit-error probabilities $\texttt{sum}_\texttt{w}$, minimum distance of generator $\texttt{md}(\texttt{G}_e)$, data length of generator $\texttt{len}_\texttt{d}(\texttt{G}_e)$, check length of generator $\texttt{len}_\texttt{c}(\texttt{G}_e)$, and number of set bits (ones) in the generator's coefficient matrix $\texttt{len}_1(\texttt{G}_e)$. The two pseudo-properties $\texttt{minimal}(e)$ and $\texttt{maximal}(e)$ indicate that numeric expression e should be minimized/maximized during synthesis. As an example,

$$\texttt{len}_\mathcal{G} = 1 \wedge \texttt{len}_\texttt{d}(\texttt{G}_0) = 4 \wedge \texttt{len}_\texttt{c}(\texttt{G}_0) \leq 4 \\ \wedge\ \texttt{md}(\texttt{G}_0) = 3 \wedge \texttt{minimal}(\texttt{len}_\texttt{c}(\texttt{G}_0)),$$

instructs the synthesizer to produce a single G_0 having 4-bit data length, at most 4 check bits, and minimum distance 3, while also minimizing the number of check bits.

3.2 SMT-Based Property Checking

Satisfiability modulo theories (SMT) [11] extends Boolean satisfiability checking with the ability to efficiently handle various theories such as arithmetic. All properties specified via the Figure 3 property language can be translated into formulas within the decidable SMT theory *quantifier-free uninterpreted functions and linear real arithmetic* (QF_UFLRA). We begin by declaring function symbols

$$len_\mathcal{G} \in \mathbb{Z}, g : \mathbb{Z}\times\mathbb{Z}\times\mathbb{Z} \to \mathbb{B}, h : \mathbb{Z}\times\mathbb{Z}\times\mathbb{Z} \to \mathbb{B},$$
$$len_d : \mathbb{Z} \to \mathbb{Z}, len_c : \mathbb{Z} \to \mathbb{Z}, md : \mathbb{Z} \to \mathbb{Z}, len_1 : \mathbb{Z} \to \mathbb{Z},$$
$$data_0 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B}, enc_0 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B}, check_0 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B},$$
$$data_1 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B}, enc_1 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B}, check_1 : \mathbb{Z}\times\mathbb{Z} \to \mathbb{B},$$
$$len_w \in \mathbb{Z}, sum_w \in \mathbb{R}, w : \mathbb{Z} \to \mathbb{R}, map : \mathbb{Z} \to \mathbb{Z}.$$

The g, h, $data_k$, enc_k, $check_k$ symbols represent matrices/vectors associated with the generators, with the first parameter identifying a specific generator. For example, $g(0,2,5)$ represents the Boolean value at position $(2,5)$ of generator G_0.

Given user-defined upper-bound constants $L_\mathcal{G}, L_d, L_c$, and L_w, we assert well-formedness constraints. For example, (1) ensures that the identity matrix of each G_i is set properly, and (2) ensures that the coefficient matrix of each H_i is set to properly match G_i. These formulas are implicitly existentially-quantified — the SMT solver will search for a *satisfying assignment*, giving a concrete function for each symbol.

$$\bigwedge_{i=0}^{L_\mathcal{G}} \bigwedge_{y=0}^{L_d} \bigwedge_{x=0}^{L_d} ((x < len_d(i) \wedge y < len_d(i)) \Rightarrow g(i,y,x) = (x = y)) \tag{1}$$

$$\bigwedge_{i=0}^{L_\mathcal{G}} \bigwedge_{y=0}^{L_d} \bigwedge_{x=0}^{L_d+L_c} ((y < len_c(i) \wedge x < len_d(i)) \Rightarrow h(i,y,x) = g(i, x, y + len_d(i))) \tag{2}$$

There are various other straightforward well-formedness constraints, including ensuring that $enc_k = data_k\, G$ and that $check_k = (H\, enc_k^T)^T$. Due to space limitations, we elide these. Constraint (3) ensures that len_1 properly counts the number of set bits (ones) in the coefficient portion of each generator.

$$\bigwedge_{i=0}^{L_\mathcal{G}} \Big(\sum_{y=0}^{L_d} \sum_{x=0}^{L_d+L_c} ite(i < len_\mathcal{G} \wedge g(i,y,x) \wedge y < len_d(i) \wedge len_d(i) \leq x < len_d(i) + len_c(i), 1, 0) \Big) = len_1(i) \tag{3}$$

The user can specify len_w weights w. In this case, the data words of the generators are considered to be permutations of bits from a larger len_w-bit word where real-valued weights determine the relative criticality level of each bit (further details in §4.3). Constraints (4) and (5) allow *map* to assign each of the len_w bits to a specific generator. In constraint (6), $chooseTimesPow(n, m)$ represents the approximation for undetected error probability from §2.2, i.e., $\binom{n}{m} \cdot p^m$ (we pre-compute this for the possible values of n, m, p, with p being a user-specifiable constant), and this ensures that sum_w tracks the weighted sum of the undetected error probabilities resulting from the mapping. of bits to generators.

$$\bigwedge_{j=0}^{L_w} 0 \leq map(j) < len_\mathcal{G} \tag{4}$$

$$\bigwedge_{i=0}^{L_\mathcal{G}} (i < len_\mathcal{G} \Rightarrow len_d(i) = \sum_{j=0}^{L_w} ite(map(j) = i, 1, 0)) \tag{5}$$

$$\Big(\sum_{j=0}^{L_w} w(j) \cdot chooseTimesPow(len_d(map(j)) + len_c(map(j)), md(map(j)))\Big) = sum_w \quad (6)$$

Constraint (7) ensures that the check bits are set to zero for the two codewords corresponding to each generator. This is used in combination with constraint (8) to ensure that the verifier can check constraints related to minimum distance.

$$\bigwedge_{i=0}^{L_G} \bigwedge_{k=0}^{1} (i < len_G \Rightarrow \neg \bigvee_{j=0}^{L_c} j < len_c(i) \wedge check_k(i,j)) \quad (7)$$

Formula ϕ_{md} states that there are two distinct codewords that differ by fewer bits than the minimum distance.

$$\phi_{md} = \bigvee_{i=0}^{L_G} (i < len_G \wedge (0 < \Big(\sum_{x=0}^{L_d+L_c} ite(x \geq len_d(i) + len_c(i) \vee enc_0(i,x) = enc_1(i,x), 0, 1)\Big) < md(i))) \quad (8)$$

As we will see in Section 3.3, the SMT solver can be used as either a *synthesizer*, in which case we assert a conjunction of constraints and ask for a satisfying assignment that solves them, or as a *verifier*, in which case we assert the *negation* of a conjunction of properties, and ask for a satisfying assignment corresponding to a property violation (*counterexample*). When we are given a list of properties $props = \psi_0, \cdots, \psi_k$ to verify, if one of them refers to the minimum distance, we append $\neg\phi_{md}$ to *props*, and the verifier then checks the assertion $\neg(\psi_0 \wedge \cdots \wedge \psi_k \wedge \neg\phi_{md})$, which is equivalent to $\neg\psi_0 \vee \cdots \vee \neg\psi_k \vee \phi_{md}$. In other words, the verifier will search for a satisfying assignment that either causes one of the properties to be violated, or satisfies ϕ_{md}.

3.3 Checking Quantified Properties

We have seen how existentially-quantified properties can be efficiently checked using an SMT solver. In general, checking more complex quantifier alternations is undecidable, but we will consider how to efficiently check arbitrary properties of the form $\exists k \in \mathcal{K}. \forall i \in \mathcal{I}. \phi(i,k)$ utilizing an approach known as counterexample-guided inductive synthesis (CEGIS) [36, 35]. In this approach, we utilize two SMT solvers operating in a loop. The first solver functions as a synthesizer, trying to satisfy the property

$$\exists k \in \mathcal{K}. \ \wedge_{i \in I} \phi(i,k)$$

for an initially-empty finite set I. If the property is satisfiable, the k from the satisfying assignment is provided to the other solver, which functions as a verifier, handling the property $\forall i \in \mathcal{I}. \phi(i,k)$ by checking that its negation $\exists i \in \mathcal{I}. \neg\phi(i,k)$ is unsatisfiable. If this negation is instead *satisfiable*, the i from the satisfying assignment (counterexample) is added to the set I, and the synthesize-verify loop repeats.

Algorithm 1: Synthesis Algorithm

```
Input: list of properties props; list of optimization
       constraints opts; solver timeout timeout
Result: generator G satisfying props, or ⊥ on failure
// initialize SMT solvers (§3.2)
(syn, ver) ← initSolvers(props, opts); G ← ⊥
for o ∈ opts do
    while o.canBeFurtherOptimized() do
        syn.push()
        syn.assert(o.getNextBound())
        G′ ← ⊥; t ← time()
        while time() − t < timeout do
            G″ ← ⊥; syn.push()
            status ← syn.checkSat(timeout)
            if status = SAT then G″ ← syn.model()
            else break // synthesizer failed
            syn.pop(); ver.push()
            ver.assert(makeAssertion(G″))
            status ← ver.checkSat(timeout)
            if status ≠ SAT then
                G′ ← G″
                break // verifier succeeded
            else syn.assert(makeCex(G″))
            ver.pop()
        if G′ ≠ ⊥ then
            o.success(); G ← G′
        else o.failure()
        syn.pop()
return G
```

3.4 Synthesis Algorithm

Algorithm 1 shows pseudocode for our synthesis approach. The *initSolvers* function asserts well-formedness constraints such as (1)-(2) into both solvers, asserts (3)-(6) into *syn*, and asserts (7) into *ver*. All properties in *props* except those referring to minimum distance are asserted into *syn*, and the remaining ones are asserted into *ver* as mentioned in the discussion of ϕ_{md} in Section 3.2. The function *makeAssertion* produces an assertion matching the synthesizer's current candidate G'', and *makeCex* produces an assertion ensuring that the synthesizer will not synthesize this G'' again.

4 IMPLEMENTATION AND EXPERIMENTS

We built a prototype implementation of the synthesis framework outlined in Section 3. Our implementation contains

min_dist	*check_len*	iterations	time (s)	RAM (GB)
8	**12**	11,395	151.80	1.52
7	**12**	9,046	121.65	1.53
6	**8**	15,109	183.86	1.56
5	**7**	12,334	121.77	1.57
4	**5**	15,662	126.02	1.55
3	**3**	682	5.16	0.81
2	**2**	637	4.72	0.81

Table 1: Generators with given minimum distance

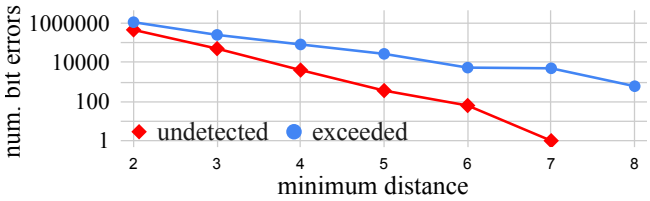

Figure 4: Generator robustness

around 2,500 lines of Java code, and utilizes two instances of the Z3 [7] SMT solver (version 4.8.11) for the *synthesizer* and *verifier* components. Our code compiles and runs on Ubuntu 22.04.4 Linux using OpenJDK 11.0.23.

For all experiments, we used a Dell OptiPlex 7080 workstation with the following hardware configuration: 10-core (20-thread) Intel i9-10900K CPU at 3.70GHz, 128 GB DDR4 RAM, and a 2TB PCIe NVME Class 40 SSD.

4.1 Verifying Real-World Generators

Algorithm 1 can also be used as a stand-alone verifier, in which case, optimization constraints are ignored, the synthesizer steps are skipped, and all *props* are provided to the verifier. We used our tool to formally verify that the 802.3df Hamming (128, 120) code generator provided in [4] has minimum distance 3. Runtime was 14.40 s, and maximum RAM used was 1.38 GB. To demonstrate *negations* of properties, we also verified that the (128, 120) code does *not* have minimum distance 4. This runtime was 122.58 s, and maximum RAM used was 1.37 GB.

4.2 Synthesizing Robust Generators

In this experiment, we measured the ability of our synthesizer to produce generators with higher levels of robustness. We fixed a data length of 4, as in the commonly-used Hamming (7, 4) code, but explored other minimum distances from 8 down to 2, minimizing the check length for each value m of minimum distance via the following property:

$$\texttt{len}_\texttt{d}(\texttt{G}_0){=}m \wedge 2 \leq \texttt{len}_\texttt{c}(\texttt{G}_0) \leq 14 \wedge \texttt{minimal}(\texttt{len}_\texttt{c}(\texttt{G}_0)).$$

We used a solver timeout of 120 s. Table 1 shows results and synthesizer performance. For example, for minimum distance 4, we synthesized the following generator G_5^4:

$$G_5^4 = \left(\begin{array}{cccc|ccccc} 1 & 0 & 0 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 0 & 1 & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & 0 & 1 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 0 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}\right)$$

To test the robustness of each of our synthesized generators, we produced a sequence of random 10,000,000 four-bit data words. For each of these data words, we used a given generator G to encode it, and then randomly flipped each bit in the codeword with channel bit-error probability $p = 0.1$. Figure 4 shows the results. The upper line shows the number of codewords that had more bit-flips than the generator's minimum distance, and this closely matches the theoretical undetected error count, $P_u(G)$·10,000,000 (Section 2.2). The lower line shows the actual undetected codeword errors — codewords transformed into *other codewords* by bit errors.

As seen in the figure, our synthesized generator G_{12}^4 having minimum distance 8 reduced the number of undetected corrupted codewords to zero.

4.3 Synthesizing a Floating-Point-Specific Generator

As we saw in Figure 1, different types of data can be affected in different ways by bit errors. In this experiment, we leveraged the characteristics of IEEE floating-point data to synthesize a floating-point-specific generator which reduced the average numeric magnitude of undetected errors, and utilized only a small number of added check bits.

We considered a 32-bit data word as two 16-bit words. For 16-bit data, we first synthesized a generator with a single check bit and minimum distance of 2. After 479 iterations (6.09 s), our synthesizer produced the following generator G_1^{16}, which functions in exactly the same way as an *even-parity bit*, allowing detection of all single-bit errors.

$$G_1^{16} = \left.\begin{pmatrix} 1 & \cdots & 0 & \mathbf{1} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & \mathbf{1} \end{pmatrix}\right\} 16 \text{ rows}$$

We then synthesized a generator G_6^{16} with 6 check bits and minimum distance 3, which took 210 iterations (6.66 s). Finally, we used the data from Figure 1 to produce *weights* for each of the upper 16 bits of a 32-bit IEEE floating point, representing the relative average magnitude of numeric errors resulting from each bit being flipped:

$$w_i = 100, 100, 100, 100, 99, 98, 82, 45, 17, 17, 8, 4, 2, 1, 1, 1.$$

We increased the number of generators to *two* (one with 5 check bits and min. dist. 3, and the other with 1 check bit and min. dist. 2), and allowed mapping $map : \mathbb{N} \rightarrow \{0, 1\}$ of each bit i within the 16-bit data to a generator. Starting with an

generators	check	undetect.	avg. err.	non-num.
$G_1^{16}\ G_1^{16}$	2	2,333,996	$2.14 \cdot 10^{36}$	5744
$G_6^{16}\ G_6^{16}$	12	12,383	$1.59 \cdot 10^{36}$	21
$G_5^8\ G_1^8\ G_1^{16}$	7	585,979	$0.24 \cdot 10^{36}$	248

Table 2: Float32-specific generator robustness

initial bound of 1000, we asked the synthesizer to minimize $\texttt{sum}_\texttt{w}$ (§3.2), where $md(map(i))$ is the minimum distance of the generator that bit i is currently mapped to:

$$sum_w = \sum_{i=0}^{15} w_i \cdot \binom{len_d(map(i)) + len_c(map(i))}{md(map(i))} \cdot p^{md(map(i))}.$$

After 997 iterations (355.48 s), the synthesizer produced G_5^8 and G_1^8, and a mapping where the upper 8 bits of the data are mapped to G_5^8 and lower 8 were mapped to G_1^8.

Similar to our approach in Section 4.2, we randomly generated 10,000,000 32-bit data words, making sure that each represented a *numeric* floating point value. For each combinations of generators listed in Table 2, we encoded the data and flipped each bit of the resulting codewords with probability $p = 0.1$. Table 2 shows that our Float32-specific combination $G_5^8\ G_1^8\ G_1^{16}$ containing the generators synthesized using per-bit weights had more undetected errors than the approach having solely minimum distance 3, but less than the simple parity-bit approach. Additionally, it has 7 check bits, more than the 2 for the parity approach, but fewer than the 12 for the minimum distance 3 approach. Most importantly, when considering the *magnitude* of the numeric error caused by bit flips within the floating-point values, our Float32-specific combination had the lowest average magnitude, while also providing a number of *non-numeric* errors (numeric data being corrupted into *not-a-number* or *infinite* values) between the other two approaches.

4.4 Synthesizing High-Performance Generators

In this experiment, we optimized generators by minimizing the number of set bits in their coefficient matrix:

$$\texttt{md}(\texttt{G}_k){=}m \ \wedge \texttt{minimal} \left(\sum_{j=0}^{d} \sum_{i=d}^{d+c} \texttt{G}_k(j,i) \right).$$

Specifically, we synthesized a generator G_{17}^{32} with minimum distance $m = 3$, and minimized the number of set bits in the coefficient matrix from 200 down to 100. At 1210 iterations (419.06 s), the synthesizer timed out on a sum equal to 118, resulting in 82 generators having sums between 119 and 200.

For each of these generators, we automatically emitted a C implementation that could perform efficient encoding and checking for that specific generator using `and` and `xor` bitwise operators. Each C program cycled through all possible 32-bit words in increments of 21, resulting in 204,522,253 total data words — the program performed encode/check for each. We used GCC 11.4.0 to compile each program, using both the `-O0` and `-O3` compiler optimization levels. Performance results are shown in Figure 5 — each runtime data point is an average across 5 runs of the program. Note that minimizing the number of ones can also have benefits at the hardware level by decreasing the gate count [34].

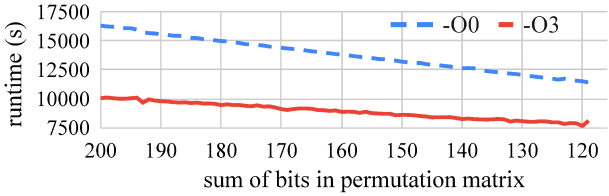

Figure 5: Encode/check performance of generators

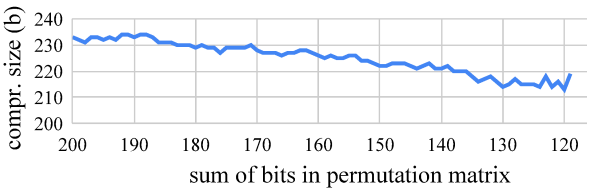

Figure 6: Compressability of generators

Besides affecting performance, the sum of bits in the permutation matrix also influences *compressibility*. For each of the 82 generators, we iterated through each column of the permutation matrix, writing the bits into a file. As a basic demonstration regarding compressibility, we created a GZIP-compressed TAR archive from each of these binary files, and measured the resulting archive sizes (see Figure 6).

5 RELATED WORK

Program Synthesis for Networks. Network programming languages [2, 26, 22] and associated program synthesis techniques have been used to solve a variety of orthogonal problems, such as synthesis of network updates [25, 23, 17] and synthesis of control-plane operations [5]. Our work on synthesis of FEC codes is complementary to these approaches.

Application-Specific FEC Codes. There are various projects that have manually designed application-specific codes, e.g., for 100G Ethernet [6], low-delay 5G [15], and repair of 802.11 packets [14]. Our work seeks to automate such design.

Verification of FEC Codes. There are works on verification of error-correction codes (ECCs) at the hardware level [8], including approaches using BDDs [12], computational geometry [20], and first-order logic (via the ACL2 theorem

prover) [28]. In contrast, our approach uses an SMT-based verification/synthesis technique.

Synthesis of FEC Codes. There are works on synthesis of Trellis codes [9] and Reed-Solomon codes [21], but these do not benefit from an efficient counterexample-guided implementation and user-specifiable properties. There is also work on exhaustive exploration of CRC polynomials [16], but this does not provide formal guarantees. Work on synthesis of Hamming-based DRAM ECCs [30] uses repeated calls to a SAT solver, but does not allow the customizability and flexibility of our SMT-based CEGIS approach.

Alternative Uses for Hamming Codes. Hamming codes have seen other uses in networking. For example, recent works have used Hamming codes for synthesis of congestion control algorithms [1] and in-network compression [38].

6 FUTURE WORK

There are many potential lines of work that can build on our prototype approach. We outline two potential directions.

Scaling Synthesis to Other Codes

There are many ways the CEGIS approach used in our prototype tool could be optimized. For example, we could seek to produce *smaller (more general)* counterexamples, rather than using the entire candidate generator matrix, to speed up the synthesize-verify loop. Additionally, we are exploring integration with other approaches for scaling up synthesis, such as using rewrite rules [24], and synthesizing *merge functions* [32] to combine multiple codes together.

Synthesizing Codes with Multi-Bit Error Correction.

As mentioned in Sec. 2.1, a single-bit error in a Hamming codeword can be detected by obtaining the check bits $\vec{b}$, and observing whether $\vec{b}$ matches a column in the check matrix. If there are *multiple* bit-errors, $\vec{b}$ will be the sum of multiple columns, meaning the bit-error positions can no longer be uniquely identified. Consider the Figure 2 example, but assume the highlighted bits of the codeword are flipped.

$$\begin{pmatrix}0&0&1&1\end{pmatrix}\left(\begin{array}{cccc|ccc}1&0&0&0&1&0&1\\0&1&0&0&1&1&0\\0&0&1&0&1&1&1\\0&0&0&1&0&1&1\end{array}\right)=\begin{pmatrix}0&\mathbf{0}&1&1&1&0&\mathbf{0}\end{pmatrix}$$

$$\left(\begin{array}{cccc|ccc}1&1&1&0&1&0&0\\0&1&1&1&0&1&0\\1&0&1&1&0&0&1\end{array}\right)\left(\begin{array}{c}0\\\mathbf{1}\\1\\1\\\hline 1\\0\\\mathbf{1}\end{array}\right)=\begin{pmatrix}1\\1\\0\end{pmatrix}+\begin{pmatrix}0\\0\\1\end{pmatrix}=\begin{pmatrix}1\\1\\1\end{pmatrix}$$

In this case, we cannot discern this two-bit error from a single-bit error, since the sum of the corresponding check matrix columns matches another column. For this reason, Hamming codes are typically only used to detect single-bit errors. However, by increasing the number of check bits and carefully adjusting the generator, we can ensure that *each pair of columns in the check matrix has a unique sum.* The following extends the previous example with 8 additional check bits to exhibit this property. This new generator still has minimum distance 3 (verified by our tool), but can now be used to detect both 1-bit and 2-bit errors.

$$H=\left(\begin{array}{cccc|cccccccccc}1&1&1&0&1&0&0&0&0&0&0&0&0&0\\0&1&1&1&0&1&0&0&0&0&0&0&0&0\\1&0&1&1&0&0&1&0&0&0&0&0&0&0\\\mathbf{1}&\mathbf{0}&\mathbf{0}&\mathbf{0}&0&0&0&1&0&0&0&0&0&0\\\mathbf{0}&\mathbf{1}&\mathbf{0}&\mathbf{0}&0&0&0&0&1&0&0&0&0&0\\\mathbf{0}&\mathbf{0}&\mathbf{1}&\mathbf{0}&0&0&0&0&0&1&0&0&0&0\\\mathbf{0}&\mathbf{0}&\mathbf{0}&\mathbf{1}&0&0&0&0&0&0&1&0&0&0\\\mathbf{1}&\mathbf{0}&\mathbf{0}&\mathbf{0}&0&0&0&0&0&0&0&1&0&0\\\mathbf{0}&\mathbf{1}&\mathbf{0}&\mathbf{0}&0&0&0&0&0&0&0&0&1&0\\\mathbf{0}&\mathbf{0}&\mathbf{1}&\mathbf{0}&0&0&0&0&0&0&0&0&0&1\\\mathbf{0}&\mathbf{0}&\mathbf{0}&\mathbf{1}&0&0&0&0&0&0&0&0&0&1\end{array}\right)$$

Based on this observation, we are working to add *number of correctable bit errors* as a property in our synthesizer, which may allow us to correct multi-bit errors using fewer check bits than the above manually-crafted check matrix.

7 CONCLUSION

In this paper, we present a novel approach for constructing FEC codes automatically via program synthesis. Our tool allows the user to tune Hamming codes for specific applications, through the use of customizable formal specifications.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their detailed comments. This work is supported by the National Science Foundation under Grant No. 2318970.

REFERENCES

[1] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2022. Automating network heuristic design and analysis. *HotNets*. ACM, 8–16.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. Netkat: semantic foundations for networks. *POPL*. ACM, 113–126.

[3] Richard Barrie, Ming Yang, and Anthony Chan Carusone. 2023. Statistical BER Analysis of Concatenated FEC in Multi-Part Links.

[4] Will Bliss, Maged F. Barsoum, and German Feyh. 2022. Proposal for a Specific (128,120) Extended Inner Hamming Code with Lower Power and Lower Latency Soft Chase Decoding than Textbook Codes. https://www.ieee802.org/3/df/public/22_10/22_1005/bliss_3df_01_220929.pdf.

[5] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: managing data plane diversity with control plane synthesis. *NSDI*. USENIX Association, 133–153.

[6] Frank Y. Chang, Kiyoshi Onohara, and Takashi Mizuochi. 2010. Forward Error Correction for 100G Transport Networks. *IEEE Commun. Mag.*, 48, 3.

[7] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. *TACAS* (Lecture Notes in Computer Science). Vol. 4963. Springer, 337–340.

[8] Keerthikumara Devarajegowda, Valentin Hiltl, Thomas Rabenalt, Dominik Stoffel, Wolfgang Kunz, and Wolfgang Ecker. 2020. Formal Verification by the Book: Error Detection and Correction Codes. *Design and Verification Conference and Exhibition*.

[9] H Ferreira. 1985. The Synthesis of Magnetic Recording Trellis Codes with Good Hamming Distance Properties. *IEEE Transactions on Magnetics*, 21, 5, 1356–1358.

[10] Robert Gallager. 1962. Low-Density Parity-Check Codes. *IRE Transactions on information theory*, 8, 1, 21–28.

[11] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. *CAV* (Lecture Notes in Computer Science). Vol. 3114. Springer, 175–188.

[12] Aarti Gupta, Roope Kaivola, Mihir Parang Mehta, and Vaibhav Singh. 2022. Error Correction Code Algorithm and Implementation Verification Using Symbolic Representations. *FMCAD*.

[13] Richard W Hamming. 1950. Error Detecting and Error Correcting Codes. *The Bell system technical journal*, 29, 2, 147–160.

[14] Jun Huang, Guoliang Xing, Jianwei Niu, and Shan Lin. 2015. CodeRepair: PHY-Layer Partial Packet Recovery without the Pain. *INFOCOM*. IEEE, 1463–1471.

[15] Mohammad Karzand, Douglas J. Leith, Jason Cloud, and Muriel Médard. 2017. Design of FEC for Low Delay in 5G. *IEEE J. Sel. Areas Commun.*, 35, 8, 1783–1793.

[16] Philip Koopman and Tridib Chakravarty. 2004. Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks. *DSN*. IEEE Computer Society, 145.

[17] Kim G. Larsen, Anders Mariegaard, Stefan Schmid, and Jirí Srba. 2023. Allsynth: A bdd-based approach for network update synthesis. *Sci. Comput. Program.*, 230, 102992.

[18] Adam Li. 2007. RTP Payload Format for Generic Forward Error Correction. *RFC*, 5109, 1–44.

[19] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications using a Binary Instrumentation Tool. *SC*. IEEE/ACM, 57.

[20] Alexey Lvov, Luis Alfonso Lastras-Montaño, Viresh Paruthi, Robert Shadowen, and Ali El-Zein. 2012. Formal verification of error correcting circuits using computational algebraic geometry. *FMCAD*. IEEE, 141–148.

[21] D Mandelbaum. 1979. Construction of Error Correcting Codes by Interpolation. *IEEE Transactions on Information Theory*, 25, 1, 27–35.

[22] Jedidiah McClurg. 2021. Correct-by-Construction Network Programming for Stateful Data-Planes. *SOSR*. ACM, 66–79.

[23] Jedidiah McClurg. 2018. *Program Synthesis for Software-Defined Networking*. PhD thesis. University of Colorado Boulder, (Jan. 2018).

[24] Jedidiah McClurg, Miles Claver, Jackson Garner, Jake Vossen, Jordan Schmerge, and Mehmet E. Belviranli. 2022. Optimizing Regular Expressions via Rewrite-Guided Synthesis. *PACT*. ACM, 426–438.

[25] Jedidiah McClurg, Hossein Hojjat, Pavol Cerný, and Nate Foster. 2015. Efficient Synthesis of Network Updates. *PLDI*. ACM, 196–207.

[26] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerný. 2016. Event-Driven Network Programming. *PLDI*. ACM, 369–385.

[27] Jiayuan Meng, Anand Raghunathan, Srimat T. Chakradhar, and Surendra Byna. 2010. Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution. *IPDPS*. IEEE, 1–12.

[28] Mahum Naseer, Waqar Ahmad, and Osman Hasan. 2020. Formal Verification of ECCs for Memories Using ACL2. *J. Electron. Test.*, 36, 5, 643–663.

[29] Andrei Nedelcu, Stefano Calabrò, Youxi Lin, and Nebojša Stojanović. 2022. Concatenated SD-Hamming and KP4 Codes in DCN PAM4 4x200 Gbps/lane. *2022 European Conference on Optical Communication (ECOC)*. IEEE, 1–4.

[30] Minesh Patel, Jeremie S. Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. 2020. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics. *MICRO*. IEEE, 282–297.

[31] Lenin Patra, Arash Farhood, Rajesh Radhamohan, Will Bliss, Sridhar Ramesh, and Dave Cassan. 2023. FEC Baseline Proposal for 200Gb/s per Lane IM-DD Optical PMDs. https://www.ieee802.org/3/dj/public/23_03/patra_3dj_01b_2303.pdf.

[32] Jonathan Protzenko, Sebastian Burckhardt, Michal Moskal, and Jedidiah McClurg. 2015. Implementing Real-Time Collaboration in TouchDevelop using AST merges. *MobileDeLi*. ACM, 25–27.

[33] Irving S Reed and Gustave Solomon. 1960. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8, 2, 300–304.

[34] Pedro Reviriego, Salvatore Pontarelli, Juan Antonio Maestro, and Marco Ottavi. 2013. A Method to Construct Low Delay Single Error Correction Codes for Protecting Data Bits Only. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 32, 3, 479–483.

[35] Armando Solar-Lezama, Christopher Jones, and Rastislav Bodík. 2008. Sketching Concurrent Data Structures. *PLDI*. ACM, 136–148.

[36] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial Sketching for Finite Programs. *ASPLOS*. ACM, 404–415.

[37] Georgios Tzimpragos, Christoforos Kachris, Ivan B. Djordjevic, Milorad Cvijetic, Dimitrios Soudris, and Ioannis Tomkos. 2016. A Survey on FEC Codes for 100G and Beyond Optical Networks. *IEEE Commun. Surv. Tutorials*, 18, 1, 209–221.

[38] Sébastien Vaucher, Niloofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. 2020. ZipLine: In-Network Compression at Line Speed. *CoNEXT*. ACM, 399–405.

[39] Xinyuan Wang and Xiang He. 2022. FEC Code and Scheme Observation in 800G/1.6TbE. https://www.ieee802.org/3/df/public/22_02/wang_3df_01_220215.pdf.

[40] Can Zhang, Søren Forchhammer, Jakob Dahl Andersen, Tayyab Mehmood, Metodi P. Yankov, and Knud J. Larsen. 2020. Fast SD-Hamming Decoding in FPGA for High-Speed Concatenated FEC for Optical Communication. *GLOBECOM*. IEEE, 1–6.