Appendix

This is the Appendix to "Verified Erasure Correction in Coq with MathComp and VST", by Joshua M. Cohen, Qinshi Wang, Andrew W. Appel, in *CAV 2022: 34th International Conference on Computer-Aided Verification,* August 2022.

A Decoder Functional Model

We can translate Equation 2, the decoder, into MathComp:

```
Variable l :  list F.
Variable uniq_l:  uniq l. (* l has no duplicates *)
Variable size_l:  size l = max_n.

Definition weights := gaussian_elim (vandermonde max_h max_n l).

Definition decoder (h xh: nat) (input: 'M[F]_(k, c)) (parities: 'M[F]_(h, c))
  (missing: seq 'I_k) (found_parities : list 'I_h) (Hh: h ≤max_h) (x_h : 'I_h)
  : 'M[F]_(k, c) :=
  (* $(W_2')^{-1}$ *)
  let w2' := invmx (submx_rows_cols_rev xh xh weights (widen_ord_seq Hh
    found_parities) (widen_ord_seq k_leq_n missing) x_max_h x_max_n) in
  (* $W_1'$ *)
  let w1' := submx_rows_cols_rev xh (k−xh) weights
    (widen_ord_seq Hh found_parities)
    (widen_ord_seq k_leq_n (ord_comp missing)) x_max_h x_max_n in
  (* $W_1'D_1$ *)
  let s := w1' *m (submx_rows_cols (k−xh) c input (ord_comp missing)
    (ord_enum c) x_k x_c) in
  (* $P - W_1'D_1$ *)
  let s' := minusmx (submx_rows_cols xh c parities (found_parities)
    (ord_enum c) x_h x_c) s in
  (* $(W_2')^{-1}(P - W_1'D_1)$ *)
  let recovered := w2' *m s' in
  fill_rows input recovered missing x_k.
```

The decoder takes in h, xh, the received data matrix, the received parity matrix, the list of missing data packets, and the list of found parities. All of the x_* variables, including x_h, are default arguments that can be ignored. widen_ord_seq has no computational content and can be ignored. submx_rows_cols and submx_rows_cols_rev select rows and columns from a matrix based on user-specified lists. invmx calculates the inverse for a given square matrix and fill_rows puts the recovered data back in its correct positions. Finally, here the weight matrix weights is defined as any row-reduced Vandermonde matrix over n_{max} unique field elements; thus, we define the encoder and decoder in their most generic forms.

B Treating bytes as Field Elements

As with all Reed-Solomon codes, we must "interpret" the input bytes as field elements, which in less-formal settings is done without issue. However, in the Coq versions of the C code generated by CompCert, the input packet elements are represented by the byte type, which is an integer paired with a proof of its bound. On the other hand, the field elements are pairs of MathComp polynomials and proofs of the degree bound. We cannot simply treat these two types as the same in a formal manner without additional effort. One possibility is to insert manual conversion functions everywhere (e.g., the C fec_gf_mult function that multiplies two bytes as field elements could be specified as field_to_byte ((byte_to_field b1)* (byte_to_field b2)), where $*$ denotes field multiplication). But this approach adds significant overhead and requires constant conversion, especially to ensure that all intermediate computations do not overflow. Instead, we define another field structure directly on the byte type, with the field operations defined by converting to and from the associated polynomial. Then, the VST proof only needs to know that the byte type has a field structure; it does not require any information about how the field operations are implemented. The VST proof does not need to reason about polynomials, their bounds, or conversions between bytes and polynomials. Additionally, thanks to Coq's Canonical Structures, used heavily in MathComp, we can use the byte type everywhere and the underlying field structure is automatically inferred. Together, these make the VST proof much simpler, and allow us to simply treat bytes as field elements, as with pen-and-paper proofs.

C $W_2'^{-1}$ Invertibility

A proof sketch of Theorem 1 from §5.1 is as follows:

Proof. Let $r_1, .., r_{m-z}$ be the indices of the rows not included in Y. Let Y' be the submatrix of V that includes all rows, the columns included in V, and columns $r_1, ..., r_{m-z}$ (In other words, we add the columns corresponding to each row missing from Y). Note that all additional columns that we added are part of the first m columns of V, and are thus part of an identity matrix; they each contain only a single nonzero value.

Now we claim that $|\det(Y)| = |\det(Y')|$. Define Y'_i $(0 \leq i \leq m - z)$ inductively as follows:

1. $Y'_0 = Y'$
2. Given Y'_i, let Y'_{i+1} be the submatrix that includes all columns and rows of V included in Y'_i except for row and column r_{i+1}.

With this definition, we have the following two key properties: $Y'_{m-z} = Y$ and for all $0 \leq i \leq m-z$, $|\det(Y'_{i+1})| = |\det(Y'_i)|$. The first property follows from the definitions, while the second follows from cofactor expansion and the fact that column r_i contains a single one with all other entries zero. Therefore, $|\det(Y)| = |\det(Y')|$.

Finally, we know that Y' is invertible, since it was formed via row operations from an $m \times m$ Vandermonde matrix on distinct elements. Thus, by Theorem 2 and the fact that row operations do not affect invertibility, Y' is invertible and therefore so is Y.

D Gaussian Elimination

To prove full Gaussian elimination (Algorithm 1) correct, we need to describe a loop invariant for the while loop. While we did prove a general invariant, it is quite complicated; for this application, we only require the simpler full-rank form:

Definition 1. *Let A be an $m \times n$ matrix with $m \leq n$. A satisfies* $\text{GaussInvar}(A, r)$ *if the following hold:*

1. *The $r \times r$ submatrix formed from the first r rows and the first r columns is a diagonal matrix with nonzero entries along the diagonal.*
2. *All other entries in the first r columns are zero.*

It is straightforward to prove that this invariant is preserved through Gaussian elimination on a full-rank matrix; that is, after the rth step of the first loop in Gaussian elimination, $\text{GaussInvar}(A, r)$ holds. At the end of the first loop, this implies that the left $m \times m$ submatrix is diagonal with nonzero entries along the diagonal, and thus after the second loop, this submatrix is the identity.

E Restricted Gaussian Elimination

We want to determine precisely when standard Gaussian elimination is equivalent to RGE. We first note that, if no ERROR is ever encountered, RGE is equivalent to Gaussian elimination after each iteration of the first loop, and therefore, $\text{GaussInvar}(A, r)$ holds before the rth iteration of the first loop of RGE.

Now suppose that A satisfies $\text{GaussInvar}(A, r)$; we examine the conditions under which the rth step of RGE succeeds. We must determine whether the entries $A_{k,r}$ are nonzero at this point, for all $0 \leq k \leq m$.

It is very difficult to directly reason about the values of elements in the matrix A, since these values change at every step. Instead, we would like to transform this condition ($A_{k,r} \neq 0$) into a statement about the invertibility of particular submatrices of A—since the algorithm consists only of row operations, we can show that invertibility is preserved across iterations of the algorithm.

To do this, we first, consider the entries $A_{k,r}$ for $k < r$. We define the matrix C_k^r as the submatrix of A consisting of the first r rows and the first $r + 1$ columns except for column k. Figure 1 shows this matrix. Then, C_k^r is invertible iff $A_{k,r} \neq 0$ (if $A_{k,r} = 0$, then C_k^r contains a row of zeroes; if $A_{k,r} \neq 0$, the rows of C_k^r are all linearly independent).

Next, consider the entries $A_{k,r}$ for $k \geq r$. We define the matrix R_k^r as the submatrix of A consisting of the first $r + 1$ columns and rows $\{0, \ldots, r - 1, k\}$.

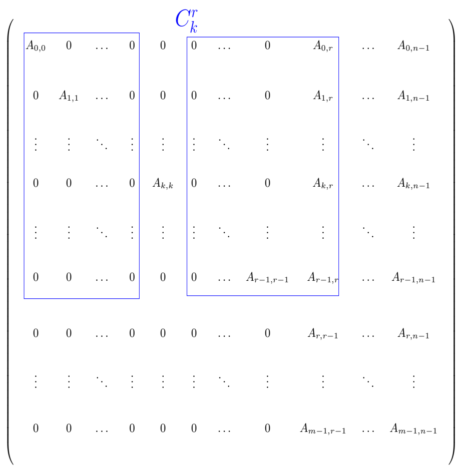

Fig. 1. Definition of C_k^r

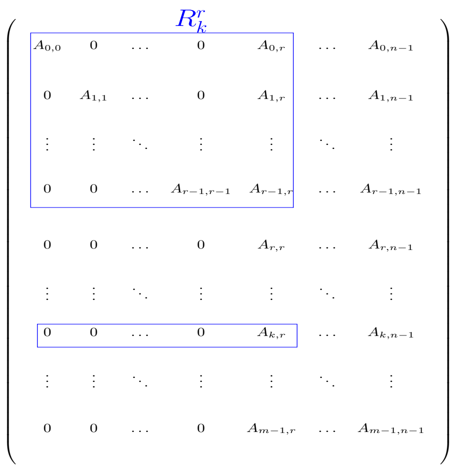

Fig. 2. Definition of R_k^r

Figure 2 shows this matrix. Then, R_k^r is invertible iff $A_{k,r} \neq 0$ (for similar reasons as C_k^r).

Therefore, assuming $\text{GaussInvar}(A, r)$, the rth iteration of RGE is correct (error-free) iff C_k^r is invertible for $0 \leq k < r$ and R_k^r is invertible for $r \leq k < m$. We will call this condition *r-strongly invertible.*

Next, we want to determine the precise conditions under which the entire algorithm succeeds, rather than just the rth step. Note that $r-$strong invertibility is not preserved through all iterations of the algorithm—it certainly does not hold of the output, which may be the identity matrix.

Instead, we can prove the following result:

Lemma 1. *Let $r < r'$ and let A be a $m \times n$ ($m \leq n$) matrix that satisfies* $\text{GaussInvar}(A, r)$*. Suppose that the rth step of Restricted Gaussian Elimination succeeds on A, and let A' be the result. Then A is r'-strongly invertible iff A' is.*

In other words, $r'-$strong invertibility is preserved for all strictly larger r' values. With this result, we can fully classify the set of matrices on which RGE works:

Theorem 1. *Let A be an $m \times n$ ($m \leq n$) matrix. Then, all steps of Restricted Gaussian Elimination succeed on A iff A is $x-$strongly invertible for all* $0 \leq x < m$.

We will say that A is *strongly invertible* (without an index) if it satisfies the right hand condition; this is, it is $x-$strongly invertible for all $0 \leq x < m$.

Recall that we must show that two different matrices are strongly invertible: the matrices W_2' from the decoder and a Vandermonde matrix on $x^{n_{max}-2}, \ldots, x^2, x, 1$. We do this in the following theorems:

Theorem 2. *Let $n < 256$ and V be a Vandermonde matrix on field elements $x^{n-1}, \ldots, x^2, x, 1$ in $GF(2^8)$. Then V is strongly invertible.*

The proof of Theorem 4 is fairly complicated, and relies on properties of row operations and repeated applications of Theorem 2 (§5.1). Note that we only proved strong invertibility for a particular Vandermonde matrix where the distinct field elements are consecutive powers of the primitive element of the field. Thus, even though the general RSE algorithm works on any Vandermonde matrix on distinct field elements (as `decoder_correct` in §5.1 states), we need a stronger condition for this implementation of RSE (using RGE) to work.

Similarly, we can prove that the W_2' are strongly invertible:

Theorem 3. *Let V be an $m \times n$ row-reduced Vandermonde matrix on distinct elements. Let $m \leq n$ and $z \leq min(m, n-m)$. Let Y be the submatrix of V formed by taking z rows of V and z of the last $(n-m)$ columns of V. Then V is strongly invertible.*

This follows directly from Theorem 1.

F Matrix Representations in Memory

1. In fec_matrix_transform, the implementation of RGE, a matrix is represented as a byte pointer, with dimensions given separately. The matrix is laid out in memory, starting at this pointer, row-by-row, except that each row is reversed. Thus, the contents in memory are
$$A_{n,1}, A_{n-1,1}, \dots, A_{1,1}, A_{n,2}, \dots, A_{1,m}.$$
The pointer is needed since the size of the input is not known at compile time (the size of the matrix to be inverted in the decoder depends on the number of lost packets). However, we cannot conjecture a good reason as to why the rows are reversed.
2. The weight matrix, generated in fec_generate_weights, is stored as a global $h_{max} \times n_{max}$ 2D array. Since it is formed by generating the Vandermonde matrix and then calling fec_matrix_transform, this array is cast to a pointer and is implicitly treated as though each of it rows are reversed. At all other points, the weight matrix is used without reversing its rows, meaning that the matrix W' used in the encoder and decoder is actually the reverse of what it appears to be (see §6.1).
3. The input packets for the encoder and the decoder are represented as an array of pointers to each packet, of possibly differing length. Thus, the input "matrix" for these functions is dispersed in memory, and we have to use VST's iter_sepcon construct to reason about the entire list of packets in memory. The elements in these "matrices" are accessed via two separate pointer dereferences, unlike the others.
4. The decoder requires intermediate matrices (for instance, to calculate $(W_2')^{-1}$), which are stored as stack-allocated 2D arrays of the maximum possible size (based on h_{max} and the bound on c). This does not necessarily correspond to the actual size of the matrix, determined by the number of lost packets. Therefore, the 2D array (of size $x \times y$) is treated as a 1D array of size xy, and is filled in from left to right so that it is treated correctly by fec_matrix_transform. This makes the VST proofs much more complicated; the contents are described as a 1D array followed by zeroes, but the memory accesses and VST types are based on 2D arrays.

G VST verifications of remaining functions

G.1 Abbreviations for global variables

We write the following abbreviations in Coq to *encapsulate* the precomputed global tables, and we use these defined names in preconditions and postconditions of our function specs:

```
Definition FIELD_TABLES gv :=
  data_at Ews (tarray tuchar fec_n) (map Vubyte byte_pows) (gv _fec_2_index) ∗
  data_at Ews (tarray tuchar fec_n) (map Vubyte byte_logs) (gv _fec_2_power) ∗
```

```
   data_at Ews (tarray tuchar fec_n) (map Vubyte byte_invs) (gv _fec_invefec).

Definition FEC_TABLES gv :=
  FIELD_TABLES gv *
  data_at Ews tint (Vint (Int.zero)) (gv _trace) *
  data_at Ews (tarray (tarray tuchar (fec_n − 1)) fec_max_h)
        (rev_mx_val weight_mx) (gv _fec_weights).
```

All use the VST primitive data_at, which allows us to describe the contents of a portion of memory, as well as the separating conjunction $*$, which denotes that each predicate refers to a disjoint section of the heap.

FIELD_TABLES gives the contents of the _fec_2_index, _fec_2_power, and _fec_invefec tables, which are the Coq lists byte_pows, byte_logs, and byte_invs, respectively. These lists precisely describe the required field operations, as the following lemmas show (Znth gets the nth element of a list, bx is the element x in the byte field, ^+ is field exponentiation, and ^−1 is field inverse):

```
(* Gives $x^i$ *)
Lemma byte_pows_elts: ∀(i: Z), 0 ≤i < Byte.modulus →
        Znth i byte_pows = bx ^+ (Z.to_nat i).

(* Gives $j$ such that $x^j = i$ for $i \neq 0$ *)
Lemma byte_logs_elts: ∀(i: Z), 0 < i < Byte.modulus →
  bx ^+ Z.to_nat (Byte.unsigned (Znth i byte_logs)) = Byte.repr i.

(* Gives inverse *)
Lemma byte_invs_elts: ∀i, 0 ≤i < Byte.modulus →
  Znth i byte_invs = (Byte.repr i)^−1.
```

FEC_TABLES says that the field tables are initialized, _fec_weights has been filled with the correct weight matrix, and the global variable _trace is set to 0. _trace is a flag for debugging and has no importance in the verification.

G.2 fec_generate_math_tables Spec

We can use the above definitions to give the VST spec for fec_generate_math_tables, the function that generates the power, logarithm, and inverse tables for the field operations:

```
Definition fec_generate_math_tables_spec :=
  DECLARE _fec_generate_math_tables
  WITH gv : globals
  PRE [ ]
    PROP ()
    PARAMS ()
    GLOBALS (gv)
    SEP (data_at Ews (tarray tuchar fec_n)
           (zseq fec_n (Vubyte Byte.zero)) (gv _fec_2_index);
         data_at Ews (tarray tuchar fec_n)
           (zseq fec_n (Vubyte Byte.zero)) (gv _fec_2_power);
         data_at Ews (tarray tuchar fec_n)
           (zseq fec_n (Vubyte Byte.zero)) (gv _fec_invefec))
  POST [ tvoid ]
    PROP ()
    RETURN ()
    SEP (FIELD_TABLES gv).
```

The spec says that global variables fec_2_index, fec_2_power and fec_invefec are initially zero-filled. After the function runs, those field tables (power, log, and inverse) are properly initialized.

G.3 Lemma decoder_list_correct

The lemma decoder_correct proves that the high-level functional model correctly reconstructs the lost packets originally sent to the encoder. This corollary extends that result to the low-level functional model. Composing this lemma with the VST proof that function fec_blk_decode satisfies its specification (fec_blk_decode_spec) proves that the C program correctly reconstructs missing packets.

```
Theorem decoder_list_correct: ∀ k c h xh (data packets : list (list byte))
  (parities : list (option (list byte))) (stats : list byte) (lens : list Z) (parbound: Z),
  0 < k ≤fec_n − 1 − fec_max_h →
  0 < c →
  0 < h ≤fec_max_h →
  xh ≤h →
  xh ≤k →
  0 ≤parbound ≤h →
  Zlength (filter (fun x ⇒ Z.eq_dec (Byte.signed x) 1) stats) = xh →
  Zlength (filter isSome (sublist 0 parbound parities)) = xh →
  Zlength parities = h →
  Zlength stats = k →
  Zlength packets = k →
  Zlength data = k →
  Zlength lens = k →
  (* The lens array contains the lengths of the packets *)
  (∀ i, 0 ≤i < k → Znth i lens = Zlength (Znth i data)) →
  (* All lengths are bounded by c *)
  Forall (fun l ⇒ Zlength l ≤c) data →
  (* The received packets (status code 0) equal the corresponding original packet *)
  (∀ i, 0 ≤i < k → Byte.signed (Znth i stats) <> 1%Z →
    Znth i packets = Znth i data) →
  (* All received parities have length c *)
  (∀ l, In (Some l) parities → Zlength l = c) →
  (* The received parities are equal to those generated by the encoder *)
  parities_valid k c parities data →
  (* Then, decoder_list recovers the original data *)
  decoder_list k c packets parities stats lens parbound = data.
```

G.4 Verifying fec_matrix_transform

The C function signature for fec_matrix_transform, the function that performs Restricted Gaussian Elimination, is:

```
int fec_matrix_transform(unsigned char *p, unsigned char i_max,
  unsigned char j_max)
```

The VST spec is:

```
Definition fec_matrix_transform_spec :=
  DECLARE _fec_matrix_transform
  WITH gv: globals, m : Z, n : Z, mx : list (list byte), s : val, sh: share
  PRE [ tptr tuchar, tuchar, tuchar]
    PROP (0 < m ≤n;
           n ≤Byte.max_unsigned;
           wf_lmatrix mx m n;
           strong_inv_list m n mx;
           writable_share sh)
    PARAMS (s; Vubyte (Byte.repr m); Vubyte (Byte.repr n))
    GLOBALS (gv)
    SEP (FIELD_TABLES gv;
         data_at sh (tarray tuchar (m * n)) (map Vubyte (flatten_mx mx)) s)
  POST [tint]
    PROP()
    RETURN (Vint Int.zero)
    SEP(FIELD_TABLES gv;
        data_at sh (tarray tuchar (m * n))
          (map Vubyte (flatten_mx (gauss_restrict_list m n mx))) s).
```

This spec says that, given a 2D-list of bytes mx that represents a valid and strongly invertible $m \times n$ matrix, if the flattened version of the matrix is stored at s before the function runs, then afterwards, s stores the flattened result of Restricted Gaussian Elimination on mx. Note that the spec shows that with the strong-invertibility precondition, this function never returns an error.

G.5 Verifying fec_generate_weights

The C function signature for fec_generate_weights, which generates the weight matrix by generating a Vandermonde matrix of the appropriate size and then running fec_matrix_transform to row-reduce it, is:

```
void fec_generate_weights(void)
```

The VST spec is:

```
Definition fec_generate_weights_spec :=
  DECLARE _fec_generate_weights
  WITH gv : globals
  PRE [ ]
    PROP ()
    PARAMS ()
    GLOBALS (gv)
    SEP (data_at Ews tint (Vint (Int.zero)) (gv _trace);
         FIELD_TABLES gv;
         data_at Ews (tarray (tarray tuchar (fec_n − 1)) fec_max_h)
           (zseq fec_max_h (zseq (fec_n − 1) (Vubyte Byte.zero)))
             (gv _fec_weights))
  POST [tvoid]
    PROP ()
    RETURN ()
    SEP (FEC_TABLES gv).
```

The spec says that (by the definition of FEC_TABLES) if global variable _fec_weights initially stores all zeroes, then after this function is run, it contains the matrix weight_mx with its rows reversed. The weight matrix is defined as the result of Gaussian elimination on the Vandermonde matrix whose (i, j)th entry is $x^{i(n_{max}-2-j)}$.

G.6 Verifying fec_blk_encode

The C function signature for fec_blk_encode, the encoding function, is:

```
int fec_blk_encode(int k, int h, int c, unsigned char **pdata, int *plen, char *pstat)
```

The VST spec is significantly more involved, since the inputs include the packets (to be appended with parity packets), packet length information, and packet status information, all of which is stored in memory. Below is the (annotated) spec:

```
Definition fec_blk_encode_spec :=
  DECLARE _fec_blk_encode
  WITH gv: globals, k : Z, h : Z, c : Z, pd: val, pl : val, ps: val,
       packets: list (list byte), lengths : list Z, packet_ptrs: list val, parity_ptrs: list val
  PRE [ tint, tint, tint, tptr (tptr tuchar), tptr (tint), tptr (tschar) ]
    PROP ((* bounds for int inputs *)
          0 < k < fec_n − fec_max_h; 0 ≤h ≤fec_max_h; 0 < c ≤fec_max_cols;
          (* lengths for arrays (all others inferred) *) Zlength packet_ptrs = k;
          (* c is the bound on the length of packets *)
          Forall (fun x ⇒ Zlength x ≤c) packets;
          (* the lengths array is correct *) lengths = map Zlength packets)
    PARAMS (Vint (Int.repr k); Vint (Int.repr h); Vint (Int.repr c); pd; pl; ps)
    GLOBALS (gv)
    SEP ((* The parity packets are initialized and contain zeroes *)
         iter_sepcon_arrays parity_ptrs (zseq h (zseq c Byte.zero));
         (* pd points to the packet pointers, followed immediately by the
            parity packet pointers *)
         data_at Ews (tarray (tptr tuchar) (k + h)) (packet_ptrs ++ parity_ptrs) pd;
         (* The packets are each initialized and stored in memory *)
         iter_sepcon_arrays packet_ptrs packets;
         (* The lengths array is stored at pointer pl *)
         data_at Ews (tarray tint k) (map Vint (map Int.repr lengths)) pl;
```

```
          (∗ The statuses, which are all 0, are stored at ps ∗)
          data_at Ews (tarray tschar k) (zseq k (Vbyte Byte.zero)) ps;
          (∗ The field tables and weight matrix are initialized ∗)
          FEC_TABLES gv))
  POST [ tint ]
    PROP ()
    RETURN (Vint Int.zero)
    SEP ((∗ The parity packets store the result of the (functional) encoder ∗)
          iter_sepcon_arrays parity_ptrs (encoder_list h k c packets);
          (∗ No other item in memory has changed ∗)
          data_at Ews (tarray (tptr tuchar) (k + h)) (packet_ptrs ++ parity_ptrs) pd;
          iter_sepcon_arrays packet_ptrs packets;
          data_at Ews (tarray tint k) (map Vint (map Int.repr (lengths))) pl;
          data_at Ews (tarray tschar k) (zseq k (Vbyte Byte.zero)) ps;
          FEC_TABLES gv).
```

In particular, note the use of iter_sepcon_arrays to represent the packets and parities; since we have a variable number of packets stored in memory, we cannot describe each one with a separate data_at predicate. Instead, iter_sepcon_arrays packet_ptrs packets says that each pointer in packet_ptrs points to memory in which the corresponding packet in packets is stored.

G.7 Specification of fec_blk_decode

The C function signature for fec_blk_decode is:

```
int fec_blk_decode (int k, int c, unsigned char ∗∗pdata, int ∗plen, char ∗pstat)
```

The full VST spec is:

```
Definition fec_blk_decode_spec :=
  DECLARE _fec_blk_decode
  WITH gv: globals, k : Z, h : Z, c : Z, parbound: Z, pd: val, pl : val, ps: val,
       packets: list (list byte), parities: list (option (list byte)),
       lengths : list Z, stats: list byte, packet_ptrs: list val, parity_ptrs: list val
  PRE [ tint, tint, tptr (tptr tuchar), tptr (tint), tptr (tschar) ]
    PROP ((∗ bounds for int inputs ∗) 0 < k < fec_n − fec_max_h;
           0 ≤ h ≤ fec_max_h; 0 < c ≤ fec_max_cols; 0 ≤ parbound ≤ h;
           (∗ lengths for arrays ∗) Zlength packet_ptrs = k; Zlength parities = h;
           (∗ c upper bounds the packet lengths ∗)
           Forall (fun x ⇒ Zlength x ≤c) packets;
           (∗ the lengths array is correct ∗) lengths = map Zlength packets;
           (∗ # of lost packets = # of found parities ∗)
           Zlength (filter (fun x ⇒ Z.eq_dec (Byte.signed x) 1) stats) =
           Zlength (filter ssrbool.isSome (sublist 0 parbound parities));
           (∗ The missing parity packets correspond to null pointers ∗)
           ∀ (i: Z), 0 ≤i < h →   Znth i parities = None ↔Znth i parity_ptrs = nullval;
           (∗ All found parity packets have length c ∗)
           ∀ (i: Z) (l: list byte), 0 ≤i < h →   Znth i parities = Some l → Zlength l = c;
           (∗ All stats are FEC_FLAG_KNOWN (0) or FEC_FLAG_WANTED (1) ∗)
           Forall (fun x ⇒ x=Byte.zero ∨ x=Byte.one) stats)
    PARAMS (Vint (Int.repr k); Vint (Int.repr c); pd; pl; ps)
    GLOBALS (gv) (* access to global variables *)
    SEP ((* The packets are stored in memory at their corresponding pointers *)
          iter_sepcon_arrays packet_ptrs packets;
          (* The received parity packets are stored in memory at their
             corresponding pointers *)
          iter_sepcon_options parity_ptrs parities;
          (* pd points to the packet pointers, followed immediately by the
             parity packet pointers *)
          data_at Ews (tarray (tptr tuchar) (k + h)) (packet_ptrs ++ parity_ptrs) pd;
          (* The lengths array is stored at pointer pl *)
          data_at Ews (tarray tint k) (map Vint (map Int.repr lengths)) pl;
          (* The stats array is stored at pointer ps *)
          data_at Ews (tarray tschar k) (map Vbyte stats) ps;
          (* The field tables and weight matrix are initialized *) FEC_TABLES gv)
  POST [ tint ] (* Postcondition; type of return value *)
    PROP ()
    (* The return value is the number of lost (and reconstructed) packets *)
    RETURN (Vint (Int.repr
      (Zlength (filter (fun x ⇒ Z.eq_dec (Byte.signed x) 1) stats))))
    SEP ((* The packets now store the result of the (functional) decoder *)
          iter_sepcon_arrays packet_ptrs
            (decoder_list k c packets parities stats lengths parbound);
          (* Nothing else in memory has changed *)
          iter_sepcon_options parity_ptrs parities;
          data_at Ews (tarray (tptr tuchar) (k + h)) (packet_ptrs ++ parity_ptrs) pd;
          data_at Ews (tarray tint k) (map Vint (map Int.repr lengths)) pl;
          data_at Ews (tarray tschar k) (zseq k (Vbyte Byte.zero)) ps;
          FEC_TABLES gv).
```