Verified Forward Erasure Correction with Coq and VST

Josh Cohen
1/13/21
Advisor: Andrew Appel

Plan

- Introduction
- Reed-Solomon Overview
- Reed-Solomon Erasure (RSE) Algorithm
- Verification Structure
- Verifying the Functional Model
  - Verifying Gaussian elimination
- Verifying the Implementation
- Results
  - Encoder VST Spec and Correctness Theorem
  - Bug found
- Related Work
- Conclusion and Future Work

Reliable Networking

- How can we ensure network defenses will protect against attackers?
- Attackers may have access to source code, ability to disrupt certain links
- Testing and static analysis not sufficient - need to know that defenses work for all possible inputs
- Larger project (Princeton, Cornell, Peraton Labs) - formally verify network components written in C and P4
- One particular defense - Forward Erasure Correction

Error-Correcting Codes

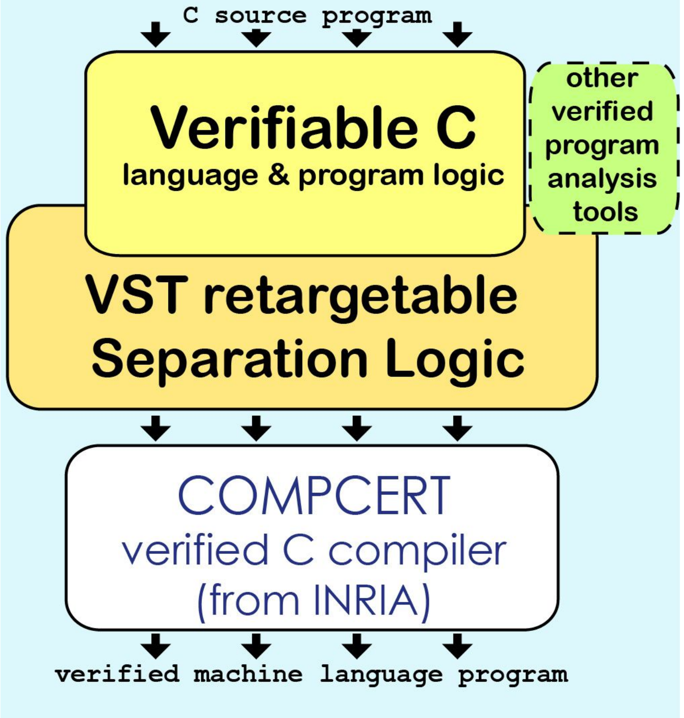
- Transporting data across network can result in lost packets
- Goal of Forward Erasure Correction - add extra parity packets to allow lost packets to be recovered
- Do so with use of an *error-correcting code (ECC)*
- ECCs used in cases when retransmission is expensive or impossible (eg: networks, satellites, etc) and in data storage
- Lots of ECCs exist (Hamming, Reed-Solomon, Convolutional, BCH, etc), most based on fairly sophisticated math
- Correctness is difficult to formally prove
- *Erasure code* - locations of missing packets are known

Project Goals

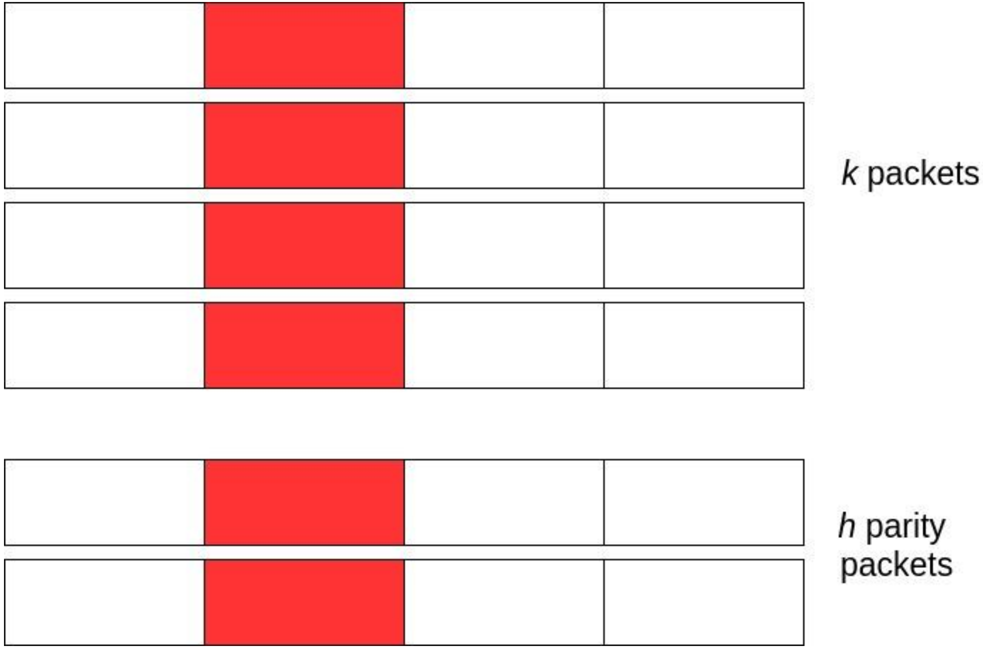
- Formally verify real-world C implementation of FEC with Coq and the Verified Software Toolchain (VST)
- C code was originally written by Anthony McAuley of Bellcore in early ‘90s based on algorithm developed by Rabin [Journal of the ACM 1989], McAuley [SIGCOMM 90], and others
- Code has been in active use since
- Verification consists of two very different tasks:
    1. Prove that the underlying algorithm is correct (using Coq and Mathematical Components)
    2. Prove that the C code implements this algorithm correctly (using Coq and VST)

Coq and VST

- Coq is an interactive theorem prover using a higher-order dependently typed logic
- CompCert (Leroy) - optimizing C compiler verified in Coq
- Verified Software Toolchain (Appel) - a program logic for C programs (higher-order separation logic) and proof automation tools for verifying C code
- Proved sound wrt CompCert C
- Formal proof that any theorems proved with VST hold of the assembly code generated by CompCert



FEC Setup



- Append *h* extra packets to recover up to *h* lost packets
- Parity packets computed columnwise - we can think of each column as a vector
- Algorithm is based on Reed-Solomon coding (first invented in 1960s)

Reed-Solomon Overview

- Interpret data as a polynomial over a finite field
  - ie: $(a_0, a_1, \ldots, a_{k-1}) \to a_0 + a_1 x + \ldots + a_{k-1} x^{k-1}$
- Evaluate polynomial at *k+h* distinct points in the field
- Equivalently, multiply by Vandermonde matrix
- To make systematic, multiply by row-reduced Vandermonde matrix

$$\begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ x_0^2 & x_1^2 & x_2^2 \end{bmatrix}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & x_1 & x_2 & x_3 \\ 0 & 1 & 0 & x_4 & x_5 & x_6 \\ 0 & 0 & 1 & x_7 & x_8 & x_9 \end{pmatrix}$$

Correcting Erasures

- Simpler than full Reed-Solomon because we don’t need to find error locations
- Encoder - multiply input by some *weight matrix W:*

$$P_{h \times c} = W_{h \times k} \cdot D_{k \times c}$$

- What properties does *W* need to allow us to decode?

Correcting Erasures

- First, suppose last *h* data packets are lost, all parities received
- Let D_1 be received data, D_2 be missing data, we have:

$$W = h\left\{ \left[\overbrace{\quad W_1 \quad}^{k-h} \overbrace{\quad W_2 \quad}^{h} \right] \right.$$

$$D = \begin{array}{r} k-h\{ \\ h\{ \end{array} \left[\begin{array}{c} \overbrace{\qquad D_1 \qquad}^{c} \\ D_2 \end{array} \right]$$

$$P = WD = W_1D_1 + W_2D_2$$

$$D_2 = (W_2)^{-1}(P - W_1D_1)$$

So, we need $(W_2)^{-1}$ (arbitrary *h* x *h* submatrix of *W*) to be invertible!

Correcting Erasures

- What if some parity packets are lost?
- Let *xh* be number of lost data packets; we must have received at least *xh* parities
- Suppose only the first *xh* parities were received (P_1):

$$\begin{array}{r} xh \left\{ \right. \\ h - xh \left\{ \right. \end{array} \begin{pmatrix} \overbrace{W_1}^{k-xh} & \overbrace{W_2}^{xh} \\ & \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \end{pmatrix} = \begin{pmatrix} P_1 \\ \end{pmatrix} \left. \right\} xh$$

$$P_1 = W_1 D_1 + W_2 D_2$$

$$D_2 = (W_2)^{-1}(P_1 - W_1 D_1)$$

Correcting Erasures

- General case is similar, but W_1, W_2, and P_1 may not be contiguous
- We need W_2 to be invertible, where W_2 is any *xh* x *xh* submatrix of W
- In other words, we need every submatrix up to size *h* x *h* to be invertible
- Row-reduced Vandermonde matrix has this (strong) property!

$$V = \begin{pmatrix} 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \dots & \alpha_{h+k} \\ \alpha_1^2 & \alpha_2^2 & \dots & \alpha_{h+k}^2 \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_1^{h-1} & \alpha_2^{h-1} & \dots & \alpha_{h+k}^{h-1} \end{pmatrix}$$

$$V_{h\times(h+k)} \xrightarrow{\text{Gaussian elim}} \left[\ I_{h\times h} \mid W_{h\times k}\ \right]$$

Reed-Solomon Erasure (RSE) Algorithm

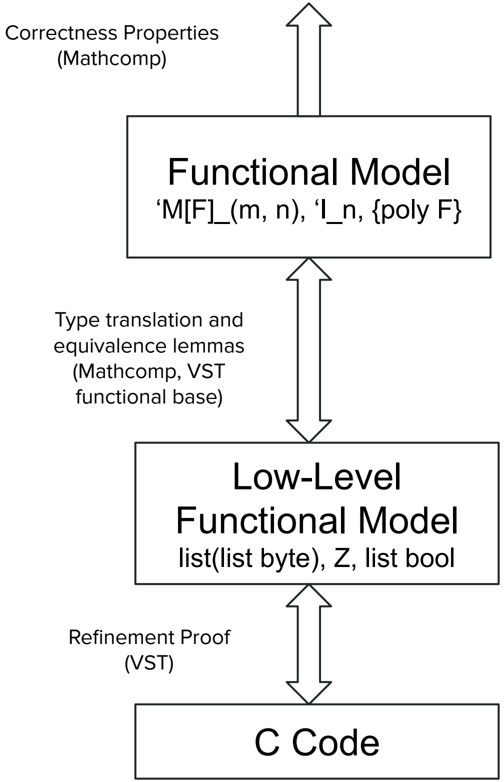
- Same as above, but using static weight matrix

- Initialization: create row-reduced Vandermonde matrix (weight matrix)
- Encoder - matrix multiplication (P=WD)
- Decoder - 2 matrix multiplications and a matrix inversion $((W_2)^{-1}(P_1-W_1D_1))$

Verification Structure

- 2 very different tasks: prove algorithm is correct and prove C code implements algorithm
- Define *functional model* - purely functional version of algorithm written in Gallina using Mathematical Components library (collection of formalized mathematics)
- Can prove properties of functional model completely independent of C program - can be used for other implementations and serves as independent spec
- Prove only that C program implements functional model with VST
- Makes proofs shorter, more modular

Verification Structure



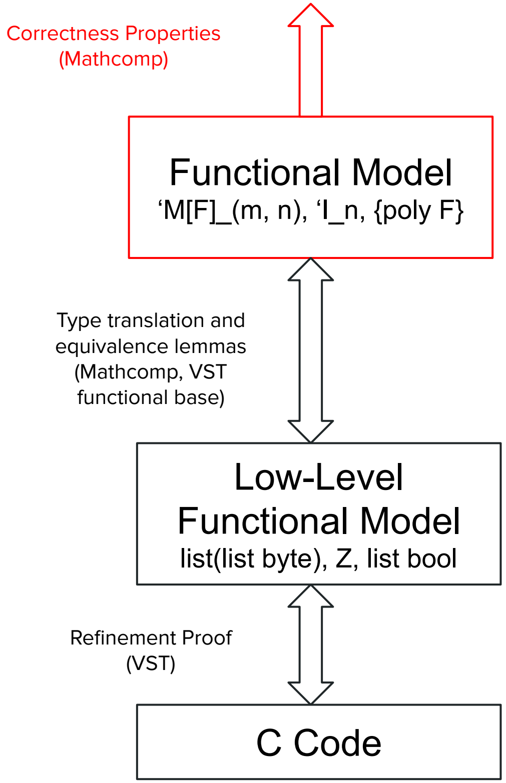
- Actually need 2 functional models
- High-level uses uses Mathcomp matrix, ordinal, and polynomial types - abstract and dependently typed
- Low level uses concrete types - list(list byte) and similar
  - nontrivial translation
- Correctness properties only use Mathcomp, refinement proof only uses VST

Verification Example - Gaussian elimination

- Standard algorithm in linear algebra to row reduce a matrix over a field
  - transform using row swaps, scalar multiplication, and adding multiples of rows
- Can be used to calculate inverses, determinants, solve systems of linear equations
- In this application - used to create weight matrix and invert matrix in decoder

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & x_1 & x_2 & x_3 \\ 0 & 1 & 0 & x_4 & x_5 & x_6 \\ 0 & 0 & 1 & x_7 & x_8 & x_9 \end{pmatrix}$$

Verification Example - Gaussian elimination



1. Define functional model and prove correctness properties

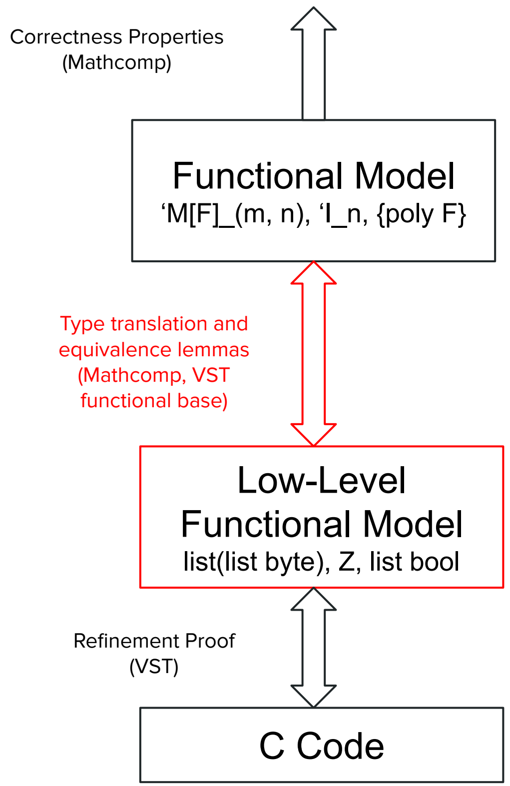
```
Definition gaussian_elim {m n} (A: 'M[F]_(m, n)) :=
  all_lc_1 (gauss_all_steps A (insub 0%N) (insub 0%N)).
```

$$\left[\;A\;\middle|\;I\;\right] \xrightarrow[\text{elim}]{\text{Gaussian}} \left[\;I\;\middle|\;A^{-1}\;\right]$$

```
Definition find_invmx {n} (A: 'M[F]_n) :=
  rsubmx (gaussian_elim (row_mx A 1%:M)).


Lemma gaussian_finds_invmx: forall {n} (A: 'M[F]_(n, n)),
  A \in unitmx ->
  find_invmx A = invmx A.
```

Verification Example - Gaussian elimination



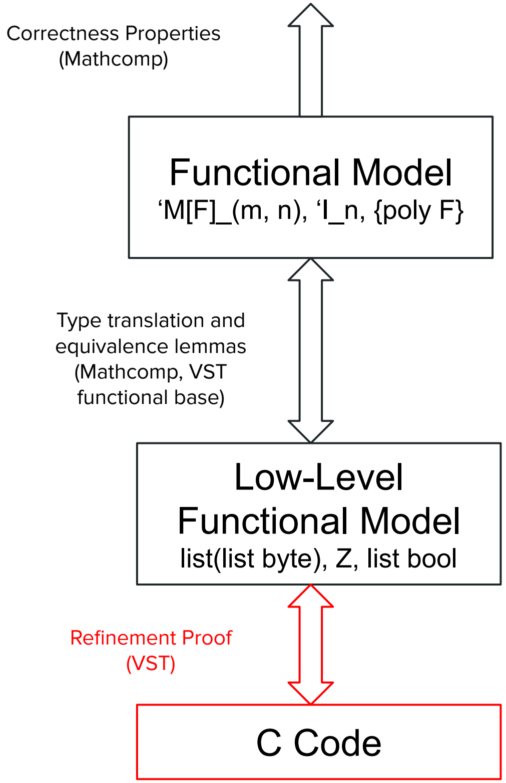
2. Define low-level functional model and prove equivalence

```
Definition lmatrix := list (list byte).

Definition gauss_restrict_list m n (mx: lmatrix) :=
all_lc_one_partial m n (gauss_all_steps_list_partial m n mx m) (m-1).

Lemma gauss_restrict_list_equiv: forall {m n} (mx: lmatrix) (Hmn: m <= n),
  wf_lmatrix mx m n ->
  lmatrix_to_mx m n (gauss_restrict_list m n mx) =
  gaussian_elim_restrict_noop (lmatrix_to_mx m n mx) (le_Z_N Hmn).
```

Verification Example - Gaussian elimination



3. Define and prove VST spec using low-level functional model

```
int fec_matrix_transform (fec_sym * p, fec_sym i_max, fec_sym j_max)
```

```
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).
```

```
Lemma body_fec_matrix_transform : semax_body Vprog Gprog
  f_fec_matrix_transform fec_matrix_transform_spec.
```

Verifying the Functional Model

- Mathcomp includes thousands of theorems about matrices, polynomials, rings and fields, etc
- We needed to prove results about constructing finite fields, computable polynomial division, Gaussian elimination, and properties of Vandermonde matrices
- 2 main challenges
    1. proving that W_2 in decoder is invertible (need sophisticated properties of Vandermonde matrices)
    2. proving that modified Gaussian elimination is correct

Restricted Gaussian Elimination

- C Code does the following: on column *r*
  - Multiply each row by inverse of r^{th} element
  - Subtract r^{th} row from all other rows
  - At end, scalar multiply to make all leading coefficients 1
- This is "restricted" Gaussian Elimination - only works if all elements in r^{th} column are nonzero!
- C code returns errors if this condition is violated
  - "FEC: swap rows (not done yet!)"
- Suggests that authors were unclear why this was sufficient

$$\begin{pmatrix} a & 0 & b & c \\ 0 & d & e & f \\ 0 & 0 & g & h \\ 0 & 0 & i & j \end{pmatrix} \rightarrow \begin{pmatrix} \frac{a}{b} & 0 & 1 & \frac{c}{b} \\ 0 & \frac{d}{e} & 1 & \frac{f}{e} \\ 0 & 0 & 1 & \frac{h}{g} \\ 0 & 0 & 1 & \frac{j}{i} \end{pmatrix} \rightarrow \begin{pmatrix} \frac{a}{b} & 0 & 0 & \frac{c}{b} - \frac{h}{g} \\ 0 & \frac{d}{e} & 0 & \frac{f}{e} - \frac{h}{g} \\ 0 & 0 & 1 & \frac{h}{g} \\ 0 & 0 & 0 & \frac{j}{i} - \frac{h}{g} \end{pmatrix}$$

```
while (*(q - k) == 0) {          /* if zero */
    if (++w == i_max) {
        return (FEC_ERR_TRANS_FAILED);  /* failed */
    }
    if (*(p + (w * i_max) + i_max - 1 - k) != 0){        /* swap rows */
        printf ("FEC: swap rows (not done yet!)\n");
        return (FEC_ERR_TRANS_SWAP_NOT_DONE);   /* Not done yet! */
    }
}
```

Restricted Gaussian Elimination

When does the r^{th} step succeed?

- Assume that first r-1 steps succeeded
- Upper left submatrix is diagonal with nonzeroes along diagonal
- All other entries in first r columns zero
- Want all $A_{k,r}$!= 0

$$C_k^r$$

$$\begin{pmatrix}
A_{0,0} & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & A_{0,r} & \cdots & A_{0,n-1} \\
0 & A_{1,1} & \cdots & 0 & 0 & 0 & \cdots & 0 & A_{1,r} & \cdots & A_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & A_{k,k} & 0 & \cdots & 0 & A_{k,r} & \cdots & A_{k,n-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & A_{r-1,r-1} & A_{r-1,r} & \cdots & A_{r-1,n-1} \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & A_{r,r} & \cdots & A_{r,n-1} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & A_{m-1,r} & \cdots & A_{m-1,n-1}
\end{pmatrix}$$

For $0 \leq k < r$,

$C_k^r =$ submatrix including rows $0 \ldots r-1$ and columns $0 \ldots r$ except k

C_k^r is invertible iff $A_{k,r} \neq 0$

Restricted Gaussian Elimination

When does the r^th^ step succeed?

- Assume that first r-1 steps succeeded
- Upper left submatrix is diagonal with nonzeroes along diagonal
- All other entries in first r columns zero
- Want all $A_{k,r}$!= 0

$$R_k^r$$

$$\begin{pmatrix} A_{0,0} & 0 & \dots & 0 & A_{0,r} & \dots & A_{0,n-1} \\ 0 & A_{1,1} & \dots & 0 & A_{1,r} & \dots & A_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & A_{r-1,r-1} & A_{r-1,r} & \dots & A_{r-1,n-1} \\ 0 & 0 & \dots & 0 & A_{r,r} & \dots & A_{r,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & A_{k,r} & \dots & A_{k,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & A_{m-1,r} & \dots & A_{m-1,n-1} \end{pmatrix}$$

For $k \geq r$,

$R_k^r =$ submatrix including columns $0 \dots r$ and rows $0 \dots r-1$ and k

R_k^r is invertible iff $A_{k,r} \neq 0$

Restricted Gaussian Elimination

r^{th} step succeeds iff

C_k^r is invertible for $0 \le k < r$ and

R_k^r is invertible for $r \le k < m$

We call this condition $r -$ strongly invertible.

```
Lemma gauss_one_step_restrict_equiv_iff:
  forall {m n} (A: 'M[F]_(m, n)) (r: 'I_m) (Hmn: m <= n),
  gauss_invar A r r ->
  gauss_one_step_restrict A r Hmn =
    Some((gauss_one_step A r (widen_ord Hmn r)).1) <->
  r_strong_inv A Hmn r.
```

The whole algorithm succeeds iff A is $x -$ strongly invertible for $0 \le x < m$.

```
Theorem gaussian_elim_equiv:
  forall {m n} (A: 'M[F]_(m, n)) (Hmn: m <= n) (Hm: 0 < m),
  strong_inv A Hm Hmn <->
  gaussian_elim_restrict A Hmn = Some(gaussian_elim A).
```

- Strong invertibility is difficult to satisfy in general (requires m^2 specific submatrices to be invertible)
- But the matrices in this application are strongly invertible (this is not trivial to show)
- Result - formal proof that simpler algorithm suffices in this instance
- Shows why this optimization/mistake is correct

Verifying the Implementation

- C code relatively challenging to verify
  - Originally written over 20 years ago
  - Code does clever and not-so-clever things
  - Not written in ways particularly conducive to verification
  - Documentation is very sparse
- Code is verified exactly as written, except that 1 macro was turned into function
- Found 1 bug - used undefined behavior

Verifying the Implementation - Challenges

- Matrices are represented many different ways in memory
  - Pointer to elements, 2D global array, 2D local array (partially filled), unsigned char**
  - Sometimes rows are reversed (for unknown reasons), sometimes not
  - Need lemmas to convert between 1D arrays, 2D arrays, pointers, etc
- Decoder is long and complex - uses about 30 local variables, many nested loops
  - VST becomes very slow and requires significant proof engineering to make verification feasible
- Uses (inconsistent) mix of array indexing and pointer arithmetic
  - Requires lemmas and tactics to relate these memory addresses

```
Lemma data_at_2darray_concat : forall sh t n m (al : list (list (reptype t))) p,
  Zlength al = n ->
  Forall (fun l => Zlength l = m) al ->
  complete_legal_cosu_type t = true ->
  data_at sh (tarray (tarray t m) n) al p
    = data_at sh (tarray t (n * m)) (concat al) p.
```

Encoder VST Spec

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

```coq
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 (0 < k < fec_n - fec_max_h; 0 <= h <= fec_max_h; 0 < c <= fec_max_cols;
          Zlength packet_ptrs = k;
          Forall (fun x => Zlength x <= c) packets;
          lengths = map (@Zlength byte) packets)
    PARAMS (Vint (Int.repr k); Vint (Int.repr h); Vint (Int.repr c); pd; pl; ps)
    GLOBALS (gv)
    SEP (iter_sepcon_arrays parity_ptrs (zseq h (zseq c Byte.zero));
         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)
  POST [ tint ]
    PROP ()
    RETURN (Vint Int.zero)
    SEP (iter_sepcon_arrays parity_ptrs (encoder_list h k c packets);
         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).
```

Logical variables

C function name

Function param types

Coq propositions

Function input values

Postcondition

Function return type

Precondition

Separation logic predicates (heap)

Function return value

Correctness Theorem (Low Level)

```
Theorem decoder_list_correct: forall 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 ->
  (forall i, 0 <= i < k -> Znth i lens = Zlength (Znth i data)) ->
  Forall (fun l => Zlength l <= c) data ->
  (forall i, 0 <= i < k -> Byte.signed (Znth i stats) <> 1%Z -> Znth i packets = Znth i data) ->
  (forall l, In (Some l) parities -> Zlength l = c) ->
  parities_valid k c parities data ->
  decoder_list k c packets parities stats lens parbound = data.
```

Bounds and length info

The lens array is correct

All lengths are bounded by c

The received packets are correct

Parity packets have length c

Received parities were produced by the encoder

Decoder recovers original data

Bug in Implementation

- In Gaussian elimination, have the code shown in 2 separate places
- i ranges from 0 to i_max, p is pointer to input matrix
- When i=0, m points to p-1
- The comparison n > m is undefined behavior (in C11, even the definition of m is undefined behavior)
- VST will not let us prove this program correct without modifying it

```
q = (p + (i * j_max) + j_max - 1);
m = q - j_max;
for (n = q; n > m; n--) {
        //loop body
}
```

Related Work - Network Function Verification

VigNAT [Zaostrovnykh et al., SIGCOMM 2017]: formally verified NAT using symbolic execution and Verifast (semi-automated separation logic tool)

Vigor [Zaostrovnykh et al., SOSP 2019]: extend VigNAT to handle other network functions (load balancer, firewall, etc) and make verification fully automatic

Gravel [Zhang et al., NSDI 2020]: use symbolic execution and SMT solvers to verify middlebox-specific properties of Click elements in C++

All of these efforts are more automatic, but cannot handle FEC - allow only restricted uses of state, do not allow unbounded loops, cannot handle sophisticated mathematical reasoning

Related Work - Formalization of Coding Theory

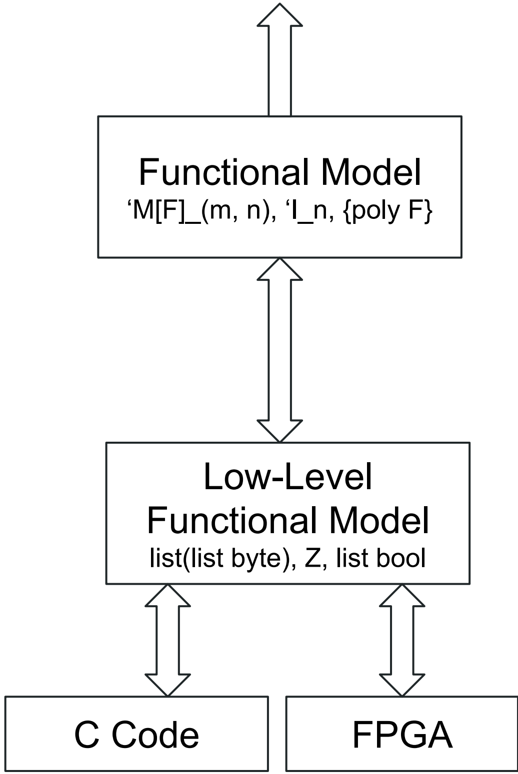
In **Coq** [Affeldt et al., Journal of Automated Reasoning 2020 and others]: library of formalized coding theory, including Hamming, Reed-Solomon, BCH, and (acyclic) LDPC codes

In **Lean** [Hagiwara et al., ISITA 2015 and Kong et al., ISITA 2018]: formalized Hamming and Insertion-Deletion codes and results about Levenstein distance

In **ACL2** [Nasser et al., Journal of Electronic Testing 2020]: verified Hamming and convolutional codes against a particular memory model

Conclusion and Future Work

- Core FEC code is fully verified
- Code is at https://github.com/verified-network-toolchain/Verified-FEC
- Remaining - code that handles buffer and packet management (calls core FEC code)
- Possible future work - implement incremental FEC encoding and decoding at line rate on an FPGA, verify correctness according to same functional model



Questions?

Thanks for listening!