EZPWD Reed-Solomon

Perry Kundert

February 8, 2015

Contents

1 Reed-Solomon Loss/Error Correction Coding

Error and erasure detection and correction for C++ and Javascript programs. Based on Phil Karn's excellent implementation (as used by the Linux kernel), converted to C++.

Performs about 40% faster than Phil's general case code, about 20% faster than his optimized code for 8-bit/CCSDS symbols.

Available both under GPLv3 and Commercial licenses (Phil's original code)

Several Javascript implementations using Reed-Solomon are provided. They aer produced from the C++ implementation using `emscripten`.

1.1 c++/ezpwd/rs

C++ implementation of Reed-Solomon codec. Fully implemented as inline code, in C++ header files.

```
#include <ezpwd/rs>

ezpwd::RS<255,251> rs;       // Reed Solomon w/ 255 8-bit symbols, up to 251 data
std::vector<uint8_t> data;   // fill data with  up to 251 bytes ...
rs.encode( data );           // Add 4 Reed-Solomon parity symbols (255-251 == 4)

// ... later, after data is possibly corrupted ...

int fix = rs.decode( data ); // Correct errors, discard 4 R-S parity symbols
```

1.2 js/ezpwd/rskey.js

A Javascript error-corrected user data input utility.

Asking a user to reliably enter even a few bytes of data (eg. a product key or a redemption code) is, well, basically impossible.

Use rskey.js to encode your data into an easily human readable key:

```
> rskey_5_encode( 12, "Mag.1ck" );
"9MGNE-BHHCD-MVY00-00000-MVRFN"
```

Later, you can decode it – even if the user adds an error or two (the 'X', below), or skips a few symbols (the last few, or by indicating they are missing with an _):

```
    Mag.1ck\pi
```

```
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00000-MVRFN" )
Object {confidence: 100, data: ArrayBuffer, string: "Mag.1ck\pi"}
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00X00-MVR" )
Object {confidence: 20, data: ArrayBuffer, string: "Mag.1ck\pi"}
> rskey_5_decode( 12, "9_GNE-BHH_D-MVY00-00X00-MVRFN" )
Object {confidence: 20, data: ArrayBuffer, string: "Mag.1ck\pi"}
```

1.3 js/ezpwd/rspwd.js

Javascript implementation of Reed-Solomon codec based password error detection and correction.

1.4 Enhancements

Several enhancements have been made.

1.4.1 Rejects impossible error position

Phil's version allows the R-S decode to compute and return error positions with the unused portion of the Reed-Solomon codeword. We reject these solutions, as they provide indication of a failure.

The supplied data and parity may not employ the full potential codeword size for a given Reed-Solomon codec. For example, and RS(31,29) codec is able to decode a codeword of 5-bit symbols containing up to 31 data and parity symbols; in this case, 2 parity symbols (31-29 == 2).

If we supply (say) 9 data symbols and 2 parity symbols, the remaining 20 symbols of unused capacity are effectively filled with zeros for the Reed-Solomon encode and decode operations.

If we decode such a codeword, and the R-S Galois field solution indicates an error positioned in the first 20 symbols of the codeword (an impossible situation), we reject the codeword and return an error.

1.4.2 Shared data tables w/ optional locking

Instead of re-computing all of the required data tables used by the Reed-Solomon computations, every instance of RS(SIZE,*) with compatible Galois polynomial parameters shares a common set of tables. Furthermore, every instance of RS(SIZE,LOAD) w/ compatible Galias polynomiam parameters shares the tables specific to the computed number of parity symbols.

The initialization of these tables is protected by a Mutex primitive and Guard object. These default to 'int' (NO-OP), but if a threading mutex and guard are provided, the shared initialization is thread-safe.