EZPWD Reed-Solomon

Perry Kundert

February 9, 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 R-S correction operations at about 135% (clang) - 200% (gcc) of the rate of Phil's general case code, about 120% (clang) - 150% (gcc) of the rate of his optimized code for 8-bit/CCSDS symbols.

Several C++ and Javascript utilities implemented using Reed-Solomon are provided.

The Javascript APIs are produced from the C++ implementation using `emscripten`, and are very performant. All Javascript is available for inclusion in websites via MaxCDN.

1.1 Licensing

All ezpwd-reed-solomon code is available under both GPLv3 and Commercial licenses. Phil's original Reed-Solomon code is LGPL, so my Reed-Solomon implementation in ezpwd-reed-solomon/c++/ezpwd/rs (which uses Phil's, with some improvements and conversion to C++) is available under the terms of the LGPL.

1.1.1 GPLv3 Licensing

If your application complies with the terms of the GPLv3, then you can use EZPWD Reed-Solomon without cost. All users of the software (eg. an installed application) or "software as a service" (eg. a website) must have access to the software source code so they can freely modify and run the software. Any modifications to underlying GPLv3 software (ie. EZPWD) must also be made available.

1.1.2 Commercial Licensing

If you use EZPWD Reed-Solomon APIs in your product but you don't wish to make your product's source code available, then a Commercial license is appropriate. The pricing breakdown is as follows (in CAD$):

Users avg. (monthly)	Price CAD$	Support CAD$/yr	Included application assistance
<10	0	25	Interesting project? ask... :)
10-1K	100	25	1 hour
1K-1M	1000	250	4 hours
1M-	10000	2500	16 hours

- Example: website gift card redemption codes
  Lets say you have an online Widget business, and generate gift cards. You average about 500 unique visitors/month, with a peak of 1500 around Christmas. You want to make your gift card redemption more reliable and secure.

Your cost would be $100, plus a $25/yr support subscription, and you would have access to an hour of time with a support developer to help you apply the js/ezpwd/rskey.js API to your website's gift card generation and redemption pages.

You decide to associate each gift card with the buyer's account (so you and the gift-card giver can know when the card is redeemed). So, each gift card RSKEY needs to contain:

- a 32-bit customer ID
- a 32-bit gift card ID

Using an RSKEY encoding 8 bytes of data, with 3 parity symbols, we get protection against 1 error or 2 erased symbols, with 1 parity symbol left over for validation.

```
// Your first customer ever, buys his first gift card!
card = {
    customer: {
        id:         0,
    },
    id:             0,
}
card_arr = new Int32Array( [card.customer.id, card.id] );

// Encrypt the gift card contents w/ an appropriate library (I don't
// recommend using a fixed one-time pad with poor randomness
// embedded in your source code...)
card_arr[0] ^= 0x21222324; // ==  555885348
card_arr[1] ^= 0x51525354; // == 1364349780

// Encode the data using the ArrayBuffer interface to RSKEY
card_key = rskey_3_encode( 8, card_arr.buffer, 4 );

// RSKEY: "4GHJ-48AL-AD95-2X8V" is printed/emailed on gift card
```

Later on, the gift card recipient comes back to the website and enters the gift-card key during checkout, mistyping some symbols, and using lower-case and alternative whitespace:

```
// Decode the customer-entered data using the same RSKEY parameters:
```

```
// EZPWD base-32 equivalents:               v v
//                    errors:    v
//                  original: "4GHJ-48AL-AD95-2X8V"
card_dec = rskey_3_decode( 8, "4g8j 48al ad9S zx8v" )
// Object {confidence: 34, data: ArrayBuffer, string: "$#"!TSRQ"}

// Recover the IDs and decrypt (once again -- don't ship one-time pad!
// The encrypted IDs can be recovered in the client, but decryption
// should, of course, be done back on the server...)
card_arr = new Int32Array( card_dec.data )// [555885348, 1364349780]
card_arr[0] ^= 0x21222324; // == 0
card_arr[1] ^= 0x51525354; // == 0

card = {
    customer: {
        id:         card_arr[0],
    },
    id:             card_arr[1],
}
// This is gift card ID 0, purchased by our first customer ID 0!
```

Assuming that an attacker does not have access to the encryption key used to encrypt the customer and card IDs in a single 64-bit block, then the probability of a fake key being produced and accepted is vanishingly small.

Lets assume that they **do** know that you are using EZPWD Reed-Solomon, and therefore always present RSKEYs that are valid R-S codewords. Furthermore, lets assume that you have alot of customers (> 2 billion), so your 32-bit customer ID is likely to accidentally match a valid customer with a probability >50%.

The decrypted customer and card IDs must be correct – match a valid customer and card ID. Since it is unlikely for each customer to generate more than a handful of gift cards, the probability that the 32-bit card ID will accidentally decrypt to any given value is $1/2^{32}$ (1 in ~4 billion). The combined 64-bit RSKEY indexes a sparsely populated array of valid values; given a number in the range (0,2^{64}], only every 4-billionth value will turn out to be valid (much less than that, in realistic scenarios).

Therefore, an attacker must generate and try more than 2 billion valid

RSKEYs before they have a 50% chance of stumbling upon one that matches a valid gift card, given the above (generous) assumptions. Even if you don't rate-limit your card redemption API, you might notice that your server is saturated with gift-card redemption requests. Assuming that your server can process 1000 redemptions per second, it would take the attacker 23 days (2,000,000 seconds) to have a 50% chance of finding his first valid fake key. So, I recommend rate-limiting your gift-card redemption API to 10 request per second, increasing the time to 6 years.

Therefore, using RSKEY and a simple encoding scheme presents an effective, robust and secure means of generating and redeeming gift-card codes.

Customer aggravation due to mis-typed codes is reduced, increasing the likelihood of return visits and positive reviews.

1.2 Enhancements

Several enhancements have been made to Phil's implementation.

1.2.1 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.

1.2.2 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<CAPACITY,*> with compatible Galois polynomial parameters shares a common set of tables. Furthermore, every instance of RS<CAPACITY,PAYLOAD> w/ compatible Galias polynomial 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.

1.3 c++/ezpwd/rs: C++ Reed-Solomon API

C++ implementation of Reed-Solomon codec. Fully implemented as inline code, in C++ header files. Highly performant, in both C++ and Javascript.

```
#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
```

1.4 Javascript Data Key API: js/ezpwd/rskey.js

Asking a user to reliably enter even a few bytes of data (eg. a product key or a redemption code) is, well, basically impossible. It is not reasonable to expect that someone will be able to perfectly read and enter a blob of random letters and numbers with 100% reliability.

Import `js/ezpwd/rskey.js` Javascript to use RSKEY error-corrected binary data input tokens in your application. Raw binary data (in Javascript or string or ArrayBuffer) can be encoded into an RSKEY for later entry by a user. Using built-in parity (extra validation) symbols, any errors or missing symbols can be detected and possibly recovered. An RSKEY that validates as correct can be trusted with a high degree of certainty, proportional to the number of excess parity symbols remaining (beyond those consumed by error detection and correction).

Use rskey.js's API 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 (if some were unreadable, indicated with an _, or the last few are not yet entered):

```
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00000-MVRFN" )
Object {confidence: 100, data: ArrayBuffer, string: "Mag.1ck"}
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00X00-MVR" ) // not yet entered
Object {confidence: 20, data: ArrayBuffer, string: "Mag.1ck"}
> rskey_5_decode( 12, "9_GNE-BHH_D-MVY00-00X00-MVRFN" ) // or unreable w/ _
Object {confidence: 20, data: ArrayBuffer, string: "Mag.1ck"}
```

If you have raw numeric data (eg. record IDs, data HMACs, etc), use the ArrayBuffer interface. You can supply any type of raw data, up to the capacity of the RSKEY (12 bytes, in this case). Then, even if errors are introduced on entry, they will be recovered if the parity is sufficient, and the returned Object's .data property will be an ArrayBuffer containing the original binary data, which you can used a TypedArray to access:

```
> ia = new Int32Array([0x31323334, 0x41424344, 0x51525354])
[825373492, 1094861636, 1364349780]
> rskey_5_encode( 12, ia.buffer ) // raw capacity is 12 bytes, w/ 5 parity
"6GRK4-CA48D-142M2-KA98G-V2MYP"
> dec=rskey_5_decode( 12, "6GRK4-CA48D-142M2-KA98G-V2XXP" ) // XX are errors
Object {confidence: 20, data: ArrayBuffer, string: "4321DCBATSRQ"}
> new Int32Array( dec.data ) // recover original data
[825373492, 1094861636, 1364349780]
```

1.4.1 RSKEY Demo examples/rskey.html

Clone RSKEY Reed-Solomon into ~/src/ezcod-reed-solomon:

```
cd ~/src
git clone git@github.com:pjkundert/ezpwd-reed-solomon.git
```

In your web browser, visit (replace <username> with your user name):

```
file:///Users/<username>/src/ezpwd-reed-solomon/examples/rskey.html
```

Try changing the Parity, Data Size and Data. Try changing the Key by entering some _ (indicating a missing/invalid symbol). These are called Erasures in Reed-Solomon terms, and we can recover one Erasure with each Parity symbol. Try changing some Key values to incorrect values. These Reed-Solomon Errors each require 2 Parity symbols to detect and correct.

You can also access the Console (right click, select Inspect Element, click on "Console"), and enter the above `rskey_...` API example code.

1.5 js/ezpwd/ezcod.js: Javascript Location Code API

To specify the location of something on the surface of the earth, a Latitude, Longitude pair is typically used. To get within +/-3m, a Latitude, Longitude pair with at least 5 digits of precision after the decimal point is required.

So, to specify where my daughter Amarissa was born, I can write down the coordinate:

```
53.655832,-113.625433
```

This is both longer and more error prone than writing the equivalent EZCOD:

```
R3U 1JU QUY.0
```

If a digit is wrong in the Latitude or Longitude coordinate, the amount of error introduced is anywhere from a few centimeters to many kilometers:

```
53.655832,-113.62543X == centimeters error
53.655832,-1X3.625433 == many kilometers error
```

EZCOD uses error/erasure correction to correct for up to 1 known missing (erased) symbol by default, with greater erasure/error detection and correction optionally available.

To encode a position of center of the Taj Mahal dome to 3m accuracy (9 position symbols, the default) and 20mm accuracy (12 symbols), and with 3 parity symbols (5-nines confidence):

```
> ezcod_3_12_encode( 27.175036985,  78.042124565 ) // default: 3m (9 symbols)
"MMF BBF GC1.2U2"
> ezcod_3_12_encode( 27.175036985,  78.042124565, 12 ) // 20mm (12 symbols)
"MMF BBF GC1 A16.1VD"
```

Later, if the EZCOD is entered, errors and erasures are transparently corrected, up to the capacity of the Reed-Solomon encoded parity:

```
> ezcod_3_12_decode( "MMF BBF GC1 A16.1VD" )
Object {confidence: 100, latitude: 27.17503683641553, longitude: 78.04212455637753,
    accuracy: 0.020401379521588606}
> ezcod_3_12_decode( "MMF BBF GC1 A16.1" ) // missing some parity
```

```
Object {confidence: 34, latitude: 27.17503683641553, longitude: 78.04212455637753,
    accuracy: 0.020401379521588606}
> ezcod_3_12_decode( "mmf-bbf-Xc1-a16.1vd" ) // An error
Object {confidence: 34, latitude: 27.17503683641553, longitude: 78.04212455637753,
    accuracy: 0.020401379521588606}
```

Try it at ezcod.com. Switch to “EXCOD 3:12”, and enter “mmf-bbf-Xc1-a16.1vd” as the EXCOD. You will see a computed accuracy of 20.4mm, and observe that the ‘X’ (error) is corrected to “G”. (The website defaults to 9 digits of precision, so it will re-encode the position, discarding the extra precision.)

1.5.1 Robustness

All symbols after the initial 9 are Reed-Solomon code symbols. Each R-S symbol can recover one known erasure; every two R-S symbols can detect and correct one other erroneous symbol. If any R-S symbols remains unused in excess of all erasures and errors, then the entire sequence can be confirmed as an R-S “codeword”, and its validity is assured, to a probability of:

```
P(1-1/2^(5*excess))
```

For example, with one R-S symbol remaining, the probability that the EZCOD is correct is:

```
P(1-1/2^5) == .969
```

If two excess R-S symbols exist, then the probability rises to:

```
P(1-1/2^10) == P(1-1/1024) == 0.999
```

With 3, it’s:

```
P(1-1/2^15) == P(1-1/32768) == 0.99997
```

Therefore, if extremely robust positions are required, select an EZCOD with 3 parity symbols, yielding almost 5-nines reliability in transmitting accurate position information – even if it must be written down, recited or entered by a human.

1.5.2 Precision

To identify the location of something within +/- 10 feet (3m) is simple: you must specify the Latitude (-90,90) to within 1 part in 4,194,304 (2^{22}) and Longitude (-180,180) to within 1 part in 8,388,608 (2^{23}).

The default 10-symbol EZCOD transmits 22 bits of Latitude and 23 bits of Longitude in 9 symbols of position data (the 10th is a parity symbol). The EZCOD API can encode up to 12 symbols of position data (29 bits of Latitude, and 31 bits of Longitude), yielding a maximum precision capability of +/- 20 millimeters.

Since the earth's circumference at the equator is ˜40,075,000m, each part in both vertical and horizontal directions is 40,075,000 / 8,388,608 == 4.777m. If you can specify a rectangle having sides of length equal to one part in the vertical and horizontal direction, then at the equator, you have a square that is 4.777m on a side. So, if we know which square some geographical coordinate lies within, it is at most sqrt(2 * $(4.777/2)^2$) == 3.378m distant from the center of the square.

As you travel north or south, the circumference of the Longitude lines decreases, as absolute Latitude increases. The average radius of the earth is ˜6,371,000m. At 53 degrees North, the circumference of the earth along a line of fixed Latitude is:

```
2 * pi * radius * cos( Latitude )
2 * 3.1415926534 * 6,371,000m * 0.60181502315
24,090,760m
```

Thus, each part along the vertical axis is still 4.777m, but each horizontal part is:

```
24,090,760 / 8,388,608 == 2.872m.
```

Now the point within each rectangle is at most:

```
sqrt( (4.777/2)^2 + (2.872/2)^2 ) == 2.787m
```

distant from the center of the rectangle.

Thus, with 9 symbols of position data, the precision of such a Latitude/Longitude encoding is at worst +/- 3.378m at the equator, at best +/-2.389m at the poles, and has an average error of less than +/-3m.

1.5.3 EZCOD Demo: http://ezcod.com

To see EZCOD in action, visit ezcod.com. Try entering:

```
    R3U 1JU QUY.0
```

to see where my daughter Amarissa was born.

You can also access the Console (right click, select Inspect Element, click on "Console"), and enter the above `rskey_`... API example code.

1.6 js/ezpwd/rspwd.js: Javascript Password Correction API

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