EZPWD Reed-Solomon Version 1.7.0

Perry Kundert

May 5, 2017

Contents

1 Reed-Solomon Loss/Error Correction

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

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 Performance

The Reed-Solomon codec's performance is often a critical determinant in product performance, and often influences expensive decisions such as CPU selection for embedded applications.

The `ezpwd::RS<...>` codec corrects data in-place to avoid unnecessary copying, using shared data tables between RS instances with compatible size, parity and Galois field parameters. The use of C++ template parameters for size-related Reed Solomon parameters enables the compiler to emit optimal code for all internal table lookups, sometimes dramatically improving performance.

1.1.1 25-75% Faster Than Competing R-S Codecs

EZPWD Reed-Solomon performs R-S correction operations (using g++ on an Intel i7) about 75% faster faster than Phil Karn's general case code, and about 25% faster than Phil's optimized code for 8-bit/CCSDS symbols. It also tests at about 25% faster than the Shifra C++ Reed-Solomon library. It appears that, for certain applications at least, EZPWD Reed-Solomon may be the fastest Open Source general-purpose C++ Reed-Solomon library presently available.

1.2 Availability

Source code is hosted at `https://github.com/pjkundert/ezpwd-reed-solomon`. It is heavily tested under both g++ (version 4.8 to 5.1) and clang++, with full optimization enabled. Get details at Hard Consulting Corporation (hard-consulting.com/products), and learn about competing geolocation encoding and why EZCOD Geolocation Encoding (hardconsulting.com/research)]] is necessary.

1.2.1 C++ Build Requires Header Source Only

Carefully implemented as C++11 standard inline code, to require no C++ object-code or library build procedure. As a result, integration into existing C++ code-bases and build systems is extremely simple.

1.2.2 Javascript Hosted by MaxCDN

Production minified Javascript is served by MaxCDN, using a URL like: `https://cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/ezcod.js`.

For example, to use the EZCOD position encoding API in your website, add this at the end of your website template, before your application.js:

```
<script
  src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/ezcod.js">
</script>
```

Note the current <VERSION> number encoded in the URI as =. . . /v<VERSION>/. . . ".

Your application may permanently access any current or historical version of the EZPWD Reed-Solomon Javascript which has a Tag in the official Git repo, by encoding the <VERSION> number in the URL:

```
//cdn.rawgit.com/.../v<VERSION>/...
```

1.2.3 Python Interface using Swig

A high-performance Python 2/3 interface is provided, using Swig (requires at least Swig Version 3.0.5 for C++11 support). Since the Python module is generated from the C++ interface, it must be generated and compiled using the appropriate target OS, Python and C++ compiler implementation.

Therefore, there is no `pip install ezpwd_reed_solomon` available for the Python bindings; they must be built and installed from the ezpwd-reed-solomon source, on the target platform, using the platform's C++ compiler with C++11 support.

To build and install the `ezpwd_reed_solomon` package, obtain the source from `https://github.com/pjkundert/ezpwd-reed-solomon`, and build with the version of Python (2 or 3) you wish to support:

```
$ cd ezpwd-reed-solomon/swig/python
$ python setup.py install
```

or:

```
$ cd ezpwd-reed-solomon
$ make swig-python
```

Presently, only `ezpwd_reed_solomon.ezcod` is available; see the 4 section for Python API details.

1. Swig 3.0.5+

   To install the Python API, you'll need a modern Swig.

   On Mac, use homebrew:

   ```
   $ brew install swig
   ```

   On Linux Debian or Ubuntu Linux systems, you should be able to use something like this (other Linux variants should have similar package installation facilities):

   ```
   $ apt-get -u install autoconf autogen libpcre3-dev bison yodl
   $ git clone git@github.com:swig/swig.git
   $ cd swig
   $ autoconf
   $ ./autogen.sh
   $ ./configure && make && make install
   ```

1.3 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.

1.3.1 GPLv3 Licensing

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

1.3.2 Commercial Licensing and Pricing

If you use any of the EZPWD Reed-Solomon based APIs in your product but you don't wish to make your product's or website's source code available, then a Commercial license from Hard Consulting Corporation (hardconsulting.com) is available. The pricing breakdown is as follows (in USD$):

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

Use of the EZCOD robust geolocation encoding module of EZPWD Reed-Solomon is free, forever, for any application. It is available under both GPLv3 and free Commercial licenses, and may even be re-implemented freely in any language, so long as it remains compatible (includes the Reed-Solomon error correction, and equivalent encoding and decoding of Latitude and Longitude coordinates).

Call us at +1-780-970-8148 or email us at info@hardconsulting.com to discuss your application.

1.4 Enhancements

Several enhancements have been made to Phil's implementation.

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/ no locking required

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 intrinsically thread-safe.

1.5 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>

// Reed Solomon codec w/ 255 symbols, up to 251 data, 4 parity symbols
ezpwd::RS<255,251> rs;

std::vector<uint8_t> data;

// ... fill data with up to 251 bytes ...

rs.encode( data ); // Adds 4 Reed-Solomon parity symbols (255-251 == 4)
```

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

int fixed = rs.decode( data );                // Correct errors, and
data.resize( data.size() - rs.nroots() );     // Discard the 4 R-S parity symbols
```

See `rssimple.C` for some basic examples.

1.5.1 Constructing an RS(SIZE,PAYLOAD) Instance

When you decide on an N-bit symbol, how do you decide on and create an instance of a Reed-Solomon codec (coder/decoder) appropriate for your data payload?

Chose your R-S Codeword symbol bit size and hence your R-S Codeword `SIZE`. Then decide how many erroneous/missing symbols you need to be able to correct for and hence your number of `PARITY` symbols required.

Now you have `SIZE`, and `PAYLOAD` is `SIZE-PARITY`.

Finally, break your data into chunks of at most `LOAD` (chunks of size < `LOAD` will be internally considered to be padded with NUL/0 symbols; you don't need to provide exactly `LOAD`-sized chunks).

For example, for 8-bit symbols, `SIZE` = 2^8-1 = 255, and for 4 symbols of `PARITY`, `LOAD` = 255-4 = 251. Therefore, the notation for the Reed-Solomon codec is RS(255,251), and the C++ declaration for such a R-S codec is:

```
ezpwd::RS<255,251> rs; // Up to 251 symbols data load; adds 4 symbols parity
```

1. Codeword `SIZE` is always 2^N-1

   For example, 8-bit symbols always use an RS(255,255-PARITY) codec. For 5-bit symbols (or, to correct only the bottom 5 bits of a larger symbol), you would use an RS(31,31-PARITY) codec.

2. Codeword `PARITY` may be from 1 to `SIZE`-1

   You may specify an R-S codec specifying a codeword with as little as 1 symbol of data payload and the remainder R-S parity, to as little as 1 symbol of parity and the remainder data payload.

1.5.2 Encoding Your Data w/ `PARITY` R-S Parity Symbols

The encode method can add symbols to a `std::string` or `std::vector<T>` (where `T` is `uint8_t` or `uint16_t`) container:

```
std::string data( "Hello, world!" )
rs.encode( data ); // Add the 4 R-S parity symbols to data
```

Alternatively, you can keep the parity separate, and not interfere with the original data (the container is not resized):

```
std::string data( "Hello, world!" );
std::string parity;
rs.encode( data, parity ); // resize and place rs.nroots() parity symbols
```

Or, if you provide a fixed-size `std::array<size_t,T>`, it will presume that the space for parity must already there at the end:

```
std::array<17,uint8_t> data(
    'H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!', // 13
    'x', 'x', 'x', 'x' );                                           // 4
rs.encode( data ); // Place the 4 R-S parity symbols at end of data
```

Or, pass pairs of `uint8_t` or `uint16_t` iterators into any container or buffer you desire:

```
std::vector<uint8_t> data( 255 );
// Fill data with 251 bytes of payload, eg:
for ( uint8_t i = 0; i < 251; ++i )
    data[i] = i;
// Append 4 symbols of R-S parity, using pairs of iterators
rs.encode( std::make_pair( data.cbegin(),       data.cbegin() + 251 ),
   std::make_pair( data.begin()  + 251, data.begin()  + 255 ))
```

1.5.3 Decoding Data w/ Corrupt/Missing Symbols

Once your data payload+parity is received, it may contain unknown erroneous symbols (called "errors"), or known missing symbols (called "erasures"). Erasures are easier to correct (because we know their location), to they only consume one R-S parity symbol to correct. Unknown errors, however, are lost in both position and value, so they each consume 2 R-S parity symbols to correct.

If the R-S algorithm can correct any errors and erasures present and recover a valid R-S "codeword", it will report a positive value:

```
int correct = rs.decode( data );
if ( correct >= 0 )
    std::cout << "Recovered data w/ " << correct << " errors" << std::endl
```

```
else
    std::cout << "Failed to recover data; " << rs << " overwhelmed" << std::endl;
```

If desired, you can pass erasure positions, and get back recovered error positions (remember that `erasures` symbols reported missing might not actually be incorrect, so might not be reported back in `position`!):

```
std::vector<int> erasures = { 1 }; // Report second symbol missing
std::vector<int> position; // And get back corrected symbols here
int correct = rs.decode( data, erasures, &position );
```

1.5.4 Discard The PARITY R-S Parity Symbols

In all cases where `rs.encode()` has added symbols to a resizable `std::string` or `std::vector<T>` container, it is your responsibility to remove them after `rs.decode()` finishes. The `rs.nroots()` method reports the number of parity symbols.

2 BCH Error Correction

Implements the Linux Kernel API for BCH error correction encoding and decoding. Thanks to Ivan Djelic for making this available under GPLv2+!

2.1 ezpwd::...bch: C++ BCH API

A C++ implementation in many ways similar to the ezpwd::RS<...> is provided. There are 3 classes (`ezpwd::bch_base`, `ezpwd::bch<...>` and `ezpwd::BCH<...>`), but the recommended one is `ezpwd::BCH<...>`.

Creating a BCH codec w/ precisely the desired codeword size, payload and bit-error correction capacity (constructor throws exception if no match BCH codec is available):

```
#include <ezpwd/bch>
ezpwd::BCH<255,239,2> bch_codec;        // By Codeword, Payload and Correction capaciti
```

Encoding into a container of uint8_t:

```
std::vector<uint8_t> codeword = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF }
bch_codec.encode( codeword ) // + 2 parity added
```

Decoding (and correcting errors)

```
int corrections = bch_codec.decode( codeword );
assert( corrections > 0 ); // fail if BCH decode failed
codeword.resize( codeword.size() - bch_codec.ecc_bytes() ); // discard parity
```

2.1.1 Classic Djelic Linux Kernel API

The stock Linux Kernel C API is retained as-is, and is made available in the `ezpwd::` C++ namespace. Initializing a BCH codec:

```
#include <ezpwd/bch_base>
// Allocate a BCH codec w/ 2^8-1 == 255 bit codeword size, and 2 bits of correction ca
// This results in a BCH( 255, 239, 2) codec: 255-bit codeword, 239-bit data payload c
// hence 255-239 == 16 bits of parity.
ezpwd::bch_control *bch = ezpwd::init_bch( 8, 2, 0 );
```

Run `bch_test` to see all available BCH codec.

Encoding parity bits on the end of an existing message is performed something like this:

```
std::array<uint8_t,10>              codeword= {
    0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF,         // data
    0, 0 };                                                 // parity, initialized to
unsigned int len = 8;
uint8_t *data = &codeword[0];
uint8_t *parity = &codeword[len];
ezpwd::encode_bch( bch, data, len, parity );
```

Decoding and correcting using the convenience API that receives error locations and applies correction(s) to supplied data:

```
int corrections = correct_bch( bch, data, len, parity );
```

Of course, the stock Linux Kernel API is available; it does not correct in-place, and the caller must perform the bit-error corrections at the error locations detected by the API:

```
unsigned int errloc[2]; // must be at least bch->t in size
int corrections = decode_bch( bch, data, len, parity, 0, 0, errloc );
for ( int n = 0; n < corrections; ++n )
    if ( errloc[n] < 8*len )
data[errloc[n]/8] ^= 1 << (errloc[n] % 8);
```

See `bchsimple.C` and `bch_test.C` for some basic examples, and `bch_itron.C` for a more advanced implementation in a real protocol.

3 RSKEY: Data Key API

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` to use RSKEY error-corrected binary data input tokens in your application. Raw binary data (in Javascript 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).

3.1 Javascript Library: js/ezpwd/rskey.js

```
rskey_<PARITY>_encode( <bytes>, data, [ sep ] ) -- encode data to RSKEY
rskey_<PARITY>_decode( <bytes>, key )           -- decode RSKEY
```

PARITY of 2-5 is supported, with a maximum capacity of 31-PARITY bytes of base-32 encoded data (raw data expands by the factor (<bytes> * 8 + 4) / 5 when base-32 encoded). With PARITY 2, the maximum capacity is 18 bytes; with PARITY 5, 16 bytes.

The `data` may be an ArrayBuffer of byte-length <= `<bytes>`. If a string is supplied, it may be a hex string beginning with '0x...' (all subsequent pairs of hex digits are used; any data beyond that is ignored). Otherwise, the string is decoded as utf-8 (of course, this means that you can't supply a utf-8 string that starts with '0x'...).

The optional `sep` parameter (default 5) is the cluster size to separate the RSKEY into; 0 specifies no separators.

Load the rskey.js Javascript into your project:

```
<script
  src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/rskey.js">
</script>
```

Use rskey.js's API to encode your data into an easily human readable key. Call the `rskey_<PARITY>_encode` API (with PARITY 2-5), specify the number of bytes of data to encode in the RSKEY's payload, and provide some data to encode (as a hex string "0x3344...", or as a utf-8 string):

```
> 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, as indicated by an '\', or the last few are simply not yet entered). Each error consumes 2 parity symbols, each erasure or missing symbol uses 1, therefore 1 error + 2 erasures results in 20% of parity remaining for validation:

```
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00000-MVRFN" )
{confidence: 100, data: ArrayBuffer, utf8: "Mag.1ck", hex: "0x4D61672E31636BCF80000000"
> rskey_5_decode( 12, "9MGNE-BHHCD-MVY00-00X00-MVR" ) // 1 error, 2 not yet entered
{confidence: 20, data: ArrayBuffer, utf8: "Mag.1ck", hex: "0x4D61672E31636BCF80000000"}
> rskey_5_decode( 12, "9_GNE-BHH_D-MVY00-00X00-MVRFN" ) // 1 error, 2 unreadable w/ '_'
{confidence: 20, data: ArrayBuffer, utf8: "Mag.1ck", hex: "0x4D61672E31636BCF80000000"}
```

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
{confidence: 20, data: ArrayBuffer, utf8: "4321DCBATSRQ", hex: "0x34333231444342415453
> new Int32Array( dec.data ) // recover original data
[825373492, 1094861636, 1364349780]
```

3.2 RSKEY Demo: `http://rskey.hardconsulting.com`

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.

3.3 Example Node.JS: Encrypted Gift Card Codes

Lets say you have an online Widget business, and generate gift cards. You average about 5000 unique visitors/month over the year, with a peak of 25000 around Christmas. You want to make your gift card redemption more reliable and secure, and less painful for your clients.

Your RSKEY license 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.

See `rskey_node.js` for sample code (the communication of the JSON request and reply between the client Website and the Node.JS server is left as an excercise to the reader.)

3.3.1 Client Website RSKEY Implementation

On the client website, you would use something like:

```
<script
  src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/rskey.js">
</script>
<script>
var client = {
    //
    // card_key_encode( card ) -- encrypt card's IDs on the server, return RSKEY
    // card_key_decode( key )  -- recover RSKEY, decrypt IDs on server, return card
    //
    //     These are run in the browser, and expect to call server methods that
    // run under Node.js back on the server.  For this demo, we'll all just run
    // here in Node.js...
    //
```

```
    card_key_encode: function( card ) {
// Get the server to encrypt the card IDs
server.card_keydata_encode( card );
// Produce the RSKEY from the card's keydata w/ Uint8Array's ArrayBuffer
card.key = rskey_3_encode( 8, new Uint8Array( card.keydata ).buffer, 4 );
return card.key;
    },

    card_key_decode: function( key ) {
// Decode the ASCII key; will raise an Exception if decode fails
var keyinfo = rskey_3_decode( 8, key );

// Convert ArrayBuffer (as Uint8Array) to plain javascript Array
var keyuint8 = new Uint8Array( keyinfo.data );
var keydata = Array( 8 );
for ( var i = 0; i < 8; ++i )
    keydata[i]  = keyuint8[i];

// Get the server to decrypt the card.keydata, return the card IDs
return server.card_keydata_decode({ keydata: keydata });
    }
}

// Your first customer ever, buys his first gift card!
card = {
    id: 0,
    customer: { id: 0 },
}

// Encode the card IDs to RSKEY
card_key = client.card_key_encode( card );
// ==> {
//   customer: { id: 0 },
//   id: 0,
//   keydata: [ 185, 124, 29, 95, 168, 45, 159, 113 ],
//   key: 'P5X1-TPW8-5NFP-2M7G'
// }
//
// "P5X1-TPW8-5NFP-2M7G" 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 (the base-32 encoding handles the Z/z/2, S/s/5, I/i/1 and O/o/0 substitutions (these symbols are equivalent in EZPWD base-32); the W/v substitution is an error):

```
// Decode the customer-entered data using the same RSKEY parameters:
//                            error:         v
//                     equivalents:   v v      v    v
//                        original: "P5X1-TPW8-5NFP-2M7G"
card_dec = client.card_key_decode( "psxi tpv8 snfp zm7g" );
// ==> {
//   keydata: [ 185, 124, 29, 95, 168, 45, 159, 113 ],
//   customer: { id: 0 },
//   id: 0
// }
//
// This is gift card ID 0, purchased by our very first customer ID 0! Find out
// what that gift card is still worth, and apply it to the order...
//
```

3.3.2 Server Node.js Encryption Implementation

All encryption should take place on the server, with a secret symmetric encryption key (which should not be stored in the repo! Use other secure key storage, or existing key material already on the server). Encrypt on the server using an appropriate cipher that encrypts all 64 bits as a block (such as `blowfish`).

```
/*
 * rskey_node.js -- Demonstrate use of rskey in Node.js application
 *
 *     Node.js "crypto" uses the Buffer type to manipulate binary data.  The
 * rskey library uses ArrayBuffer, because it is intended to be used in both
 * Node.js and Browser Javascript applications.
 *
 *    The server will expect an Object containing (at least) card.id and
 * card.customer.id, and produce/consume card.keydata.
 *
 */
```

```
var crypto              = require( "crypto" );
var crypto_algo         = 'blowfish'; // 64-bit block cipher
var crypto_secret       = 'not.here'; // Super secret master key; don't keep in Git...

var server = {
    //
    // card_keydata_encode -- Encipher card IDs into card.keydata Array
    // card_keydata_decode -- Decipher card IDs from card.keydata Array
    //
    //     Run these on your server (of course, keeping crypto_secret... secret.)
    //
    card_keydata_encode: function( card ) {
// Create Buffer containing raw card ID data
var buf         = new Buffer( 8 );
buf.writeUInt32LE( card.customer.id,    0 );
buf.writeUInt32LE( card.id,             4 );

// Encrypt the Buffer of keydata
var encipher    = crypto.createCipher( crypto_algo, crypto_secret );
encipher.setAutoPadding( false ); // must use exact 64-bit blocks
var enc         = Buffer.concat([
    encipher.update( buf ),
    encipher.final()
]);

// Return card w/ encrypted IDs as plain Javascript Array in .keydata
card.keydata    = enc.toJSON().data; // {type: 'Buffer', data: [1,2,...]}
return card;
    },

    card_keydata_decode: function( card ) {
if ( card.keydata.length != 8 )
    throw "Expected 8 bytes of card.keydata, got: " + card.keydata.length;

// Decrypt the Buffer of keydata
var decipher    = crypto.createDecipher( crypto_algo, crypto_secret );
decipher.setAutoPadding( false ); // must use exact 64-bit blocks
var dec         = Buffer.concat([
    decipher.update( new Buffer( card.keydata )),
    decipher.final()
```

```
]);

// Recover raw card IDs from Buffer
if ( card.customer == undefined )
    card.customer = {};
card.customer.id= dec.readUInt32LE( 0 );
card.id         = dec.readUInt32LE( 4 );
return card;
    }
};
```

Assuming that an attacker does not have access to the encryption key used by the server 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 (remember: all data must be encrypted with a block cipher)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.

4 EZCOD: 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.

4.1 Javascript Library: js/ezpwd/ezcod.js:

```
ezcod_3_10_encode( lat, lon, [ symbols ] ) -- encode location to EZCOD
ezcod_3_10_decode( ezcod )                 -- decode EZCOD to position
```

There are three variants provided:

- `ezcod_3_10_...` – 1 parity symbol
- `ezcod_3_11_...` – 2 parity symbols
- `ezcod_3_12_...` – 3 parity symbols

Load the ezcod.js Javascript into your project:

```
<script
 src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/ezcod.js">
</script>
```

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 "EZCOD 3:12", and enter "mmf-bbf-Xc1-a16.1vd" as the EZCOD. 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.)

4.1.1 Asynchronous Loading

Emscripten-generated code must have its run-time initialized before it can be called. If you get Javascript resources normally, they will load asynchronously, but be run in the order you load them so the Emscripten run-time will be safely initialized before your applivation's Javascript runs.

If you load other Javascript libraries like jQuery and your application.js, and you load ezcod.js asynchronously, you must ensure that they do not use any Emscripten libraries (such as ezcod.js) until their run-time initialization is complete. Our Emscripten-based libraries are completely self-contained,

so you can use the `<script onload...>` to signal jQuery to trigger its `on( 'ready', ... )` event. Regardless of whether `jquery.min.js` or `ezcod.js` loads first, this code will ensure that your `app.js on( 'ready', ... )` event will not fire until `ezcod.js` has its Emscripten run-time initialized:

```
<script type="text/javascript">
   // Bindings for Emscripten initialization detection.
   var jquery_release = function() {
       console.log( "Emscripten run-time initialized before jQuery loaded" );
       jquery_loaded = function() {}; // nothing left to do after jquery loads
   };
   var jquery_loaded = function() {
       console.log( "Emscripten run-time initialize blocking jQuery..." );
       $.holdReady(true);            // force delay of jQuery.on( 'ready', ...
       jquery_release = function() {
   console.log( "Emscripten run-time initialized; jQuery released" );
   $.holdReady(false);       // ... 'til Emscripten runtime initialized
       };
   };
</script>
<script async onload="jquery_release()"
    src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.7.0/js/ezpwd/ezcod.js">
</script>
<script defer onload="jquery_loaded()"
    src="//ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
</script>
<script defer
    src="js/app.js">
</script>
```

4.2 Python Library: ezpwd$\backslash_{\text{reed}\backslash\text{solomon}}$

The Python `ezpwd_reed_solomon` package contains an `ezcod` sub-module. While fully functional, it is designed to be simple to augment, should your geolocation encoding needs be unique.

It is extremely simple to add new EZCOD APIs to the Python binding. Simply edit the `swig/python/ezcod/ezcod.i` file, and re-install the Python binding. For example, to add a new binding class called `ezcod.ezcod_20mm_15` (with 20mm accuracy in 12 location encoding symbols + .99997 certainty in 3 parity symbols), add the following to the bottom

of `ezcod.i`:

```
%template(ezcod_20mm_15) ezpwd::ezcod<3,12>;
```

4.2.1 `ezpwd_reed_solomon.ezcod`

Classes are provided to produce three variants of EZCOD by default: 3m (9 symbols) of location accuracy, plus 1, 2 or 3 Reed-Solomon parity symbols. They are named `ezcod_3_10`, `ezcod_3_11` and `ezcod_3_12`, respectively, indicating the default 3m accuracy, and the total number of symbols.

```
$ python
>>> from ezpwd_reed_solomon import ezcod
```

The API supports the following classes, methods and attributes:

1. `ezcod_3_{10,11,12}( "<EZCOD>"|[lat,[lon,[seper,[chunk]]]] )`

   Creates an <ezcod> instance containing the specified geolocation (defaults to latitude 0.0, longitude 0.0, '.' separator and chunk 3). If a string is supplied, it is decoded (if possible; an Exception is raised if the provided EZCOD is invalid).

   ```
   >>> EZCOD = ezcod.ezcod_3_12( 53.5, -113.8 )
   >>> print repr( EZCOD )
   <R3U 06B MJ3.JXR (100%)  ==  +53.5000000000, -113.8000000000 +/-   0.00mm>
   ```

2. `ezcod_3_{10,11,12}.encode( [precision] )`

   Encodes the current `ezcod_3_{10,11,12}`'s `.latitude` and `.longitude` to the given number of symbols of precision (default: 9, or 3m). The accuracy may be anywhere from 1 to 12 (20mm accuracy) symbols.

   ```
   >>> print EZCOD.encode( 12 )
   R3U 06B MJ3 EDD.K56
   ```

3. `ezcod_3_{10,11,12}.decode( "<EZCOD>" )`

   Any variant of `ezcod_3_{10,11,12}` can decode a valid EZCOD with the expected amount (or more) parity, so long as it contains a '.' or '!' to separate the position and R-S parity symbols.

   The percentage certainty is returned – the proportion of expected R-S parity symbols that remain unused after error detection and correction. A value of 0 indicates that the EZCOD's R-S decoding did not fail, but no parity symbols remain in excess to verify its validity.

```
>>> print EZCOD.decode( "r3u 06b mj3 edd.k56" )
100
>>> EZCOD.latitude
53.49999999627471
>>> print EZCOD.decode( "r3u 06b m_3 edd.k56" )
67
>>> print EZCOD.decode( "r3u 06b mX3 edd.k56" )
34
>>> print repr( EZCOD )
<R3U 06B MJ3.JXR ( 34%)  ==  +53.4999999963, -113.8000000734 +/-   19.4mm>
>>>
```

If any symbols are unknown, replace them with either `_` or `?` to indicate that they are erasures (and consume only a single symbol of R-S parity to correct). Any undetected erroneous symbol corrected by the R-S codec consumes 2 parity symbols. A failure to decode (too many errors or erasures) will raise a `RuntimeError` exception:

```
>>> EZCOD.decode( "r3u 06b mj3 __d.__6" )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
RuntimeError: ezpwd::ezcod::decode: Error correction failed; too many erasures
>>> EZCOD.decode( "r3u 06b mj3 eXd.__6" )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
RuntimeError: ezpwd::ezcod::decode: Error correction failed; R-S decode failed
```

If an EZCOD codec expecting fewer R-S parity symbols (eg. an EZCOD 3:10 codec) is used to decode an EZCOD with more parity (eg. an EZCOD 3:12 code w/ 3 parity), it will only decode with the "strength" of the shorter codec.

For example, even though an EZCOD 3:12 offers almost 5-nines probability of correctness (1-1/32^3 == P(.99997)), if you use an EZCOD 3:10 codec to decode it, it will only use one of the R-S parity symbols, and thus only be able to correct 1 erasure (instead of 1 error and 1 erasure). Furthermore, it will only be able to provide 1-nines probability of correctness (1-1/32 == P(.96875))

```
>>> ezcod.ezcod_3_12().decode("r3u 06b mj3 edd.k56")
100
>>> ezcod.ezcod_3_10().decode("r3u 06b mj3 edd.k56")
100
>>> ezcod.ezcod_3_12().decode("r3u 06b mj3 ed_.k56") # even though 3 parity availa
67
>>> ezcod.ezcod_3_10().decode("r3u 06b mj3 ed_.k56") # all codec parity capacity u
0
>>> ezcod.ezcod_3_12().decode("r3u 06b mj3 e__.k56")
34
>>> ezcod.ezcod_3_10().decode("r3u 06b mj3 e__.k56")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
RuntimeError: reed-solomon: number of erasures exceeds capacity (number of roots)
```

4. `ezcod_3_{10,11,12}` Instance Attributes

   The following attributes are available in each `ezcod_3_{10,11,12}` instance:

Attribute	Type	Range	Description
latitude	float	[-90,90]	Geographical position in degrees
longitude	float	[-180,180]	”
latitude\error	float	[0,inf]	Axis error in meters
longitude\error	float	[0,inf]	”
accuracy	float	[0,inf]	Average of error ellipse axes in meters
precision	int	[1,12]	Desired number of location symbols
confidence	int	[0,100]	Percentage of parity symbols remaining
certainty	float	[0.0,1.0]	Certainty that EZCOD decoded was correct
chunk	int	[0,6]	Spaces every ’chunk’ position symbols
separator	char	’.’, ’!’, ’ ’	SEP\NONE/DEFAULT/DOT/BANG/SPACE
space	char	’ ’, ’-’	CHK\NONE/DEFAULT/SPACE/DASH
SEP\NONE	char	”	Output no position/parity separator
SEP\DEFAULT	char	’00’	Output no position/parity separator
SEP\DOT	char	’.’ (default)	Output a ’.’ position/parity separator
SEP\BANG	char	’!’	Output a ’!’ position/parity separator
SEP\SPACE	char	’ ’	Output a ’ ’ position/parity separator
CHK\NONE	char	”	Output no space between chunks
CHK\DEFAULT	char	’00’	Output the default between chunks
CHK\SPACE	char	’ ’ (default)	Output a ’ ’ space between chunks
CHK\DASH	char	’-’	Output a ’-’ space between chunks

It is recommended to use either `SEP_DOT` (default) or `SEP_BANG` (avoid `SEP_NONE`) for `separator`, so that the EZCOD parser can unambiguously determine the total EZCOD size, and the number of parity symbols to expect.

4.3 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 certainty 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.

4.4 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.

4.5 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.

4.5.1 EZCOD REST API Demo

A self-hosted website like `ezcod.com` with an EZCOD converstion REST API can be made available on `http://localhost:8000` by installing the Python `ezpwd_reed_solomon` module and running `examples/ezcod_api/server.py`. On a Mac, the complete process for this is:

```
$ git clone https://github.com/pjkundert/ezpwd-reed-solomon.git
$ brew install swig
$ make -C ezpwd-reed-solomon/swig/python install
$ pip install web.py
$ cd ezpwd-reed-solomon/examples/ezcod_api
$ ./server.py --prefix api --bind localhost:8000
```

Argument	Description
–bind <i'face>:<port>	Bind the web server to the given interface and port
–analytics <id>	Issue Google Analytics code using the given ID
–prefix <path>	Host the REST API at the URL: <path>/<version>
–log <file>	Put logs into the given file

The REST API URL always includes the version `v#.#.#`; for the above command the API is hosted at: `http://localhost:8000/api/v1.7.0`. To get the details for an EZCOD, encode a request with the EZCOD as a query

option. For example, visit this with a web browser: `http://localhost:8000/api/v1.7.0?ezcod=r3u08mpvt.d`. This will return the decoded data as HTML. To get it in JSON form, append `.json` to the API requests path: `http://localhost:8000/api/v1.7.0.json?ezcod=r3u08mpvt.d`.

This demo application supports GET query options and POST form variables (or body JSON of the form `{...}` or `[{...},...]` with object properties) matching:

Keyword	Description
ezcod	An EZCOD 3:10/11/12
latlon	A "Lat,Lon" pair as a string
latitude	A geographic Latitude in degrees
longitude	A geographic Longitude in degrees
precision	The number of symbols of geolocation data
parity	The number of desired EZCOD parity symbols

For example, to get the details of an EZCOD using `wget`:

```
$ wget -S --header='Content-Type: application/json'         \
    -qO - --post-data '{"ezcod":"r3u08mpvt.d", "parity":3}' \
    http://localhost:8000/api/v1.7.0
HTTP/1.1 200 OK
 Cache-Control: no-cache
 Content-Type: application/json
 Transfer-Encoding: chunked
 Date: Wed, 03 May 2017 12:31:38 GMT
 Server: localhost
{
   "accuracy": 0.0,
   "certainty": 1.0,
   "confidence": 100,
   "ezcod": "R3U 08M PVT.JKG",
   "latitude": 53.55553865432739,
   "latitude_error": 0.0,
   "longitude": -113.87387037277222,
   "longitude_error": 0.0,
   "precision": 9
}
```

You can supply single objects, or a list:

```
... --post-data '[{"ezcod":"r3u08mpvt.d"},{"latlon:" "53.5,-113.8"}'
```

5 RSPWD: Password Correction API

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

5.1 Javascript Library: js/ezpwd/rspwd.js