Javascript and C++ Reed-Solomon FEC

Perry Kundert

2014-06-25 00:00:00

Now a state-of-the-art, blazing fast implementation of Reed-Solomon error correction is available in C++ and Javascript, for production deployment in both hard-core industrial C++ control and communications software on Linux, Mac or Windows, and in Web-based (browser) and Server (Node.JS) Javascript applications. Thanks to Évariste Galois' group theory inventions, we can solve complex error and loss encoding problems reliably and efficiently.

If you are dealing with recovering data reliably from errors introduced during transmission or storage, then you need Reed-Solomon encoding. Do your users have to enter data such as redemption of coupon-codes, geolocation data or software keys? Stop frustrating them! Reed-Solomon encode, and detect and auto-correct the entered data! EZPWD Reed-Solomon – `https://dominionrnd.com/products/13-reed-solomon` is available now, under GPLv3 and Commercial licenses. (PDF)

Contents

1 Forward Error Correction For The Web Era

Dominion R&D has one of the fastest and most reliable C++ and Javascript Reed-Solomon codecs available; up to twice as fast as Phil Karn's awesome implementation, a trusted industry standard. The full library of Reed-Solomon codec and associated utilities and APIs is available under GPLv3 and Commercial licenses; the core Reed-Solomon codec implementation in C++ is licensed under the terms of the LGPL.

Available immediately on Github at `https://github.com/pjkundert/ezpwd-reed-solomon`.

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

1.1 Why You Need Forward Error Correction (FEC)

1.1.1 If your program deals with human beings entering data...

then you should be helping them enter it reliably with automatic error correction. If you generate redemption codes such as account numbers, gift card codes or product license keys, you client is going to be attempting to enter this data, and is almost certainly going to enter it incorrectly. Each time this happens, they are (perhaps subconsciously) angered – at your software, and your company. And rightly so: it is neither kind nor reasonable to expect a human to perfectly transcribe a sequence of random numbers and letters.

1.1.2 If you program deals in important or safety-critical data...

then you should be protecting its integrity with automatic error detection/correction. Geographical locations are an excellent example; imagine that you are trying to direct someone to a remote accident scene and someone incorrectly notes a single digit of a coordinate, leading the rescue helicopter kilometers in the wrong direction.

Even seemingly simple and obvious things like account and invoice numbers should be encoded! Imagine how many accounting errors occur every day that could be prevented if every account identifying number was automatically error detected and corrected on entry? And there is also the "business intelligence" issues; your competition can accurately estimate the number of clients you develop and your revenues from a sample of just few invoices, if the document sequence numbers are not encrypted. You can both encrypt and error-correct your account numbers in a single step, in C++, Javascript (web browser or Node.JS), and Python applications (with many more application languages such as PHP and Java easily available via Swig).

We've solved this problem, so you don't have to. Over the last decade, we have used Reed-Solomon encoding in industrial communications on a truly massive scale. We have solved some serious performance and reliability problems, and packaged this technology in a usable, reliable form for quick production deployment. We have also made the Commercial licensing simple and extremely economical for any scale of application deployment.

2 APIs Provided

2.1 C++ ezpwd::RS<...> API

Implementing Reed-Solomon error correction in your C++ application is extremely simple:

```
#include <ezpwd/rs>

// Reed Solomon w/ 255 (8-bit) symbols,
// up to 251 capacity ==> 4 parity
ezpwd::RS<255,251> rscodec;

std::vector<uint8_t> data;

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

// Add 4 Reed-Solomon parity symbols (255-251 == 4)
rscodec.encode( data );

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

// Correct errors, discard 4 parity symbols
int fixed = rscodec.decode( data );
```

Advanced APIs are available for dealing with erasures (missing data), corrected error positions, etc. Predefined codecs are available for symbol sizes from 2 to 16 bits (up to 64K blocks are supported). The Reed-Solomon tables are shared efficiently between all R-S codecs with compatible sizes and Galois field parameters, for excellent space efficiency.

Performance testing indicates thruput 25-75% faster than Phil Karn's excellent C implementation, and 25% faster than the renowned Schifra C++ implementation.

2.2 C++ ezpwd::BCH<...> API

We have made Ivan Djelic's wonderful Linux Kernel implementation of BCH codes (or Bose–Chaudhuri–Hocquenghem codes) quickly and easily usable in C/C++ code. It is licensed under GPLv2+.

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>
// By Codeword, Payload and Correction capacities, exactly
ezpwd::BCH<255,239,2> bch_codec;
```

Encoding into a container of uint8_t:

```
std::vector<uint8_t> codeword = {  // 8 data
    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 on bad BCH
codeword.resize(            // discard parity
    codeword.size() - bch_codec.ecc_bytes() );
```

2.3 Javascript RSKEY API: coupon-code

Generating and redeeming binary data codes on websites (Coupons, Product Keys, etc.) is a disaster.

It is inappropriate to expect a user to transcribe a complex coupon code or product key without error. In fact, it is somewhat insulting. When your user (inevitably) gets a couple of characters wrong, they are forced to manually find the error, and re-enter the code.

Instead, just enable Reed-Solomon error correction on the code. A few wrong symbols entered? No problem – they automatically get corrected.

Much happier clients == more revenue for your web store!

Generate a code with built-in Reed-Solomon error correction using the rskey.js APIs. For example, encrypt 64 bits of your Customer ID and card ID and generate an RSKEY like:

```
P5X1-TPW8-5NFP-2M7G
```

and print or email this for your client.

Later, when the client re-enters the code (with error 'W'->'v'), you might get back something like:

```
psx1 tpv8 snfp zm7g
```

No problem. The transcriptions errors are corrected (whitespace and dashes are ignored), and your encrypted card ID data is recovered. A quick request back to your server decrypts the customer and card IDs, and you can present the discount!

Since all of the RSKEY decoding occurs in your client's browser, you can easily display that the coupon code is valid in real-time – even showing the auto-correction, if you wish. All decryption occurs safely and securely on your server, so no encryption keys are ever transmitted to the browser.

2.4 Javascript EZCOD API: 20mm Global Position in 15 Symbols, w/ ~5-Nines Reliability

Communicating important accurate geolocation information (such as where my daughter Amarissa was born) is very difficult. Making a mistake in one digit of a location such as:

```
53.655832,-113.625433
```

can result in anywhere from centimeters to thousands of kilometers of error.

Instead, use EZCOD to encode this information to within +/- 3m accuracy in just 10 symbols, anywhere on the planet – with error detection and erasure correction:

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

If you require more accuracy and reliability, select an EZCOD with up to 12 symbols of position accuracy, and 3 symbols of Reed-Solomon parity. This gives you a position accurate within 20mm, and guarantees reliability with a probability of .99997, in just 15 symbols:

```
R3U 1JU QUY LO2.XJ8
```

Try corrupting a symbol, or replace 1 or 2 symbols with _ to indicate an erasure, or missing symbol. The 10-symbol variant can support up to one erased symbol, and the 15-symbol version w/ 3 parity symbols can recover from one error or up to 3 erased symbols!

Get more details here about EZCOD and how it compares with all the other geolocation encoding schemes. Here is an example of how to use it in your Javascript application (see source code for C++ and Python API details):

```
<script src="//cdn.rawgit.com/pjkundert/ezpwd-reed-solomon/v1.8.0/js/ezpwd/ezcod.js"></script>
<script type="text/javascript">

var ezcod = ezcod_3_12_encode( 53.655832, -113.625433, 12 );
// R3U 1JU QUY LO2.XJ8

// ... later ... 1 error ('Y'->'v'):

var pos = ezcod_3_12_decode( "r3u 1ju quv lo2.xj8" );
// {confidence: 34,
// latitude: 53.65583198145032,
// longitude: -113.62543302588165,
// accuracy: 0.019441640257668036}
</script>
```

Use EZCOD for free. Forever. For whatever application you want.

2.4.1 Comparing EZCOD to Other Geolocation Encodings

EZCOD is superior in several import ways to most geolocation encoding schemes available today.

- Locations nearby each-other almost always have leading symbols in common
- More symbols of position data encode greater precision
  - 3 symbols gets within 100km, almost the level of integer latitude and longitude
  - 6 symbols yields about 1/2 km resolution
  - 9 symbols yields 3m accuracy
  - 12 symbols yields 20mm accuracy
- Error correction; 3 parity symbols yields roughly 5-Nines of certainty, and recovery from 1 error and up to 3 missing symbols
- Readily available and highly performance C++ and Javascript implementations
- GPLv3 and Free Commercial licensing, for use in GPL or Proprietary software and systems
  - Free licensing extends to cover alternative implementations; no vendor lock-in

The 10:10 Code is quite similar, but encodes only to within 10m in the same 10 symbols, due to an inefficient encoding. It has a check character which can help confirm the validity of the code, but (since it doesn't use Reed-Solomon encoding), it cannot be used to recover a missing symbol.

The Natural Area Code can encode in 3 dimensions (something EZCOD cannot), but contains no check characters or parity, so cannot detect or correct errors. It has similar precision to EZCOD, and can scale to encode greater precision. However, its lack of error detection limits its utility for transmitting geolocation information.

A Geohash (`http://geohash.org`) encodes increasing amounts of geolocation precision in increasing numbers of symbols, much like EZCOD. It also encodes bits from both lat/lon interleaved in each symbol, so locations near each-other will almost always have leading symbols in common. However, it does not provide any error detection/correction, making it difficult to use to reliably transmit geolocation data.

The Google Open Location Code attempts to meet many of the goals of EZCOD, but does not provide any error detection or correction, making it undesirable for reliable communication of geolocation.

3 Licensing

Our Javascript and C++ Reed-Solomon libraries are available under the terms of the GPLv3 and Commercial licenses. The bare core c++/ezpwd/rs_base implementation is licensed under the terms of the LGPL.

Commercial licensing is required for usage not covered under the GPLv3. This includes use in websites or other SaaS applications where the complete implementation of the website/service is not available under the terms of the GPLv3 and "Tivo"-like devices where building and replacing the firmware is not possible.

For small projects (<1K unique monthly users, annual avg.), the license fee is $100, plus USD$25/yr support contract (or, a combined license and support fee of USD$225 one-time cost).

3.1 Fees

These fees are denominated in USD$ at the time of request, payable in various cryptocurrencies:

Users	License + Support	Combined
< 1K	$100 + $25/yr	$225
1K - 1M	$1,000 + $250/yr	$2,250
> 1M	$10,000 + $2,500/yr	$22,500