My Project

Generated by Doxygen 1.8.13

Contents

Chapter 1

CS2014 2017 Assignment3 - Moar crypto, of the play-coin variety

our basic cs2094 coin type

Your assignment is to write code that creates bitcoin-like values that meet the spec below and that are verified by the code you're given here. What we'll be doing is actually more like a `hashcash proof-of-work` rather than anything to do with the bitcoin blockchain.

We'll re-use the `mbed TLS` package for the verifier code. You pretty much have to do the same for the coin making code so that the automated assignment marking system works. There are some hints below to which you ought pay attention.

To setup a working environment...

See `assignment2`.

If you clone the course repo and this file is in `$REPO/assignments/assignment3` then that directory contains a Makefile you can use. That Makefile assumes that you already built mbed TLS for assignment 2 and that the mbed TLS header files and library are below `$REPO/assignments/assignment2` - if you used some other directory structure you'll need to figure out what to change.

"CS2014 COIN" Specification

Overview

The basic idea is similar to, but a lot simpler than, the bitcoin idea of mining `"difficulty"`. We require that each "CS2014 coin" includes the inputs to a SHA256 hash whose output value has a selected number of the low order (rightmost) bits with zero values. Since the output of a hash funcion like SHA256 is essentially random, one has to try many times before one sees such an output, and the longer the run of zeros required, the more attempts one needs.

To generate such outputs we include a `cryptographic nonce` in the hash input value. We can vary the nonce value until we find a hash output with the required number of bits zero-valued. (Efficiency in coin mining is clearly important, so please do try to make your code for this part as speedy as you can! Some marks may be available for that - ping me if you think you've written some notably good code.)

In addition (just for the coding fun:-) we require each coin to be digitally signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) with the p256 NIST curve. (That's the default when you generate an Elliptic Curve key pair, so you won't need to understand any details of ECDSA:-) Our coins therefore also include the public key needed to verify the coin signature in the signed data.

There are also some housekeeping fields to help with encoding and decoding and for (pretend:-) futureproofing.

CS2014 coins are binary values. We don't use JSON, XML or any other generic data representation scheme.

Example

Here's a hexdump of an example coin with 20 bits of difficulty:

```
    00,00,00,00,00,00,00,14,00,00,00,9e,30,81,9b,30
    10,06,07,2a,86,48,ce,3d,02,01,06,05,2b,81,04,00
    23,03,81,86,00,04,01,c8,46,55,6b,4e,26,bb,6e,22
    d8,7a,f8,2e,1b,15,0b,18,af,98,33,59,00,66,d9,0c
    08,63,75,4a,ea,50,5e,54,7e,72,8e,3d,57,cb,89,15
    0f,bc,10,0b,5a,1b,3a,84,08,9f,73,0a,e7,38,c7,03
    e4,2e,1a,19,45,08,25,f8,01,bd,89,0f,3a,e1,18,3a
    87,51,74,71,94,a2,4c,8a,1e,3a,7c,52,f3,03,6e,91
    fe,97,42,4f,3e,22,b7,c5,72,8c,f8,da,dd,53,ee,42
    ca,af,8d,78,38,70,10,63,e9,8c,51,a5,02,f2,89,f8
    a0,4d,68,7a,a5,96,d4,67,70,12,00,00,00,20,1e,a9
    86,c4,2b,ff,9f,99,00,2d,be,2e,91,c4,5a,ac,b7,49
    e4,7e,1a,7f,65,ae,29,bf,3f,c7,d0,c5,ce,39,00,00
    00,20,2c,55,ee,bd,2c,f0,ad,c8,77,56,cf,b6,15,e8
    5e,2b,18,ce,3e,5c,fc,56,d2,4f,9a,8b,f5,71,a5,10 <- 20 zero bits start with that last nibble
    00,00,00,00,00,8a,30,81,87,02,41,3b,cb,b3,10,9a
    87,03,89,ec,61,aa,e4,9c,83,1a,7e,27,64,5b,6d,74
    fc,6c,a7,f2,f9,2c,1c,11,c6,56,76,b2,77,aa,92,c8
    cf,de,e8,9d,0f,0f,e3,c0,7a,5b,8f,04,e0,a2,7d,af
    70,27,57,fb,4b,ba,3d,48,c2,fa,e5,ee,02,42,01,86
    ff,a4,93,1e,ba,18,f5,14,65,06,25,86,10,9c,d7,3e
    53,30,c9,39,a3,90,13,b2,7f,a1,ba,10,af,5b,53,c8
    b1,ae,6a,19,ed,2a,a3,3a,ec,8b,01,7c,50,9a,15,8b
    7a,77,7b,28,b4,70,71,1f,77,40,c2,6b,22,0e,6e,fb
```

Protocol data units (PDUs)

Our coin syntax is pretty simple, with all but one field being fixed width (for now), and hence doesn't need us to use any data definition language such as ASN.1, XML schema etc.

The fields in a CS2014 coin are:

Offset	Name	Length	Description
0	Ciphersuite	4	Specifies all cryptographic algorithms - value for now fixed at zero
4	Bits	4	Specifies difficulty
8	Public Key length	4	Specifies length of public key
12	Public Key	158	Public key value (fixed for p256 curve)
170	Nonce len	4	length of nonce
174	Nonce	32	nonce used to generate PoW hash
206	PoW Hash len	4	length of proof-of-work hash
210	PoW Hash	32	proof-of-work hash
242	Signature len	4	length of coin self-signature
246	Signature	Variable (∼138 octets)	coin self-signature

The ciphersuite value of zero means: "use SHA256 for the proof-of-work and use ECDSA with NIST p256 for the public key and signature." The ciphersuite concept is used in TLS and various other cryptographic protocols.

All length fields and the bits field are in network byte order.

As a side-note: the public key and signature fields do actually internally use ASN.1 specified encoding of those values encoded with the Distiguished Encoding Rules (DER). That's done so that those are the same values used in X.509 public key certificates, as used for Transport Layer Security (TLS), e.g. when using HTTPS.

While we don't notice it here, that actually makes those values more complex and bigger for no ostensibly good reason, if one didn't consider the savings in code re-use. Cases like that are common, and are one reason why it can take a very loooong time to migrate away from use of some pretty old format/API to any shiny new format/API. (See issues with `PKCS#1v1.5`.)

Cryptographic inputs/outputs

- Bytes 0..205 inclusive are input to the Proof-of-Work (PoW) hash (which uses SHA256).
- Bytes 206..241 inclusive are the PoW length and hash value
- Bytes 0..241 inclusive are input to the signature
- Bytes 242..end are the signature length and value

Example (again)

Breaking the above sample down into these fields we get...

```
Ciphersuite (0)
00,00,00,00,
Difficulty in terms of bits (20)
            00,00,00,14,
Length of public key (4)
                        00,00,00,9e,
Public key value (138, DER encoded)
                                    30,81,9b,30
10,06,07,2a,86,48,ce,3d,02,01,06,05,2b,81,04,00
23,03,81,86,00,04,01,c8,46,55,6b,4e,26,bb,6e,22
d8,7a,f8,2e,1b,15,0b,18,af,98,33,59,00,66,d9,0c
08,63,75,4a,ea,50,5e,54,7e,72,8e,3d,57,cb,89,15
0f,bc,10,0b,5a,1b,3a,84,08,9f,73,0a,e7,38,c7,03
e4,2e,1a,19,45,08,25,f8,01,bd,89,0f,3a,e1,18,3a
87,51,74,71,94,a2,4c,8a,1e,3a,7c,52,f3,03,6e,91
fe,97,42,4f,3e,22,b7,c5,72,8c,f8,da,dd,53,ee,42
ca,af,8d,78,38,70,10,63,e9,8c,51,a5,02,f2,89,f8
a0,4d,68,7a,a5,96,d4,67,70,12,
Length of Nonce (4)
                              00,00,00,20,
Nonce value (32)
                                          1e,a9
86,c4,2b,ff,9f,99,00,2d,be,2e,91,c4,5a,ac,b7,49
e4,7e,1a,7f,65,ae,29,bf,3f,c7,d0,c5,ce,39,
Length of Proof-of-Work hash (4)
                                          00,00
00,20,
Proof-of-Work hash (32)
      2c,55,ee,bd,2c,f0,ad,c8,77,56,cf,b6,15,e8
5e,2b,18,ce,3e,5c,fc,56,d2,4f,9a,8b,f5,71,a5,10
00,00,
Length of Signature (4)
      00,00,00,8a,
Signature (138, DER encoded)
                  30,81,87,02,41,3b,cb,b3,10,9a
87,03,89,ec,61,aa,e4,9c,83,1a,7e,27,64,5b,6d,74
fc,6c,a7,f2,f9,2c,1c,11,c6,56,76,b2,77,aa,92,c8
cf,de,e8,9d,0f,0f,e3,c0,7a,5b,8f,04,e0,a2,7d,af
70,27,57,fb,4b,ba,3d,48,c2,fa,e5,ee,02,42,01,86
ff,a4,93,1e,ba,18,f5,14,65,06,25,86,10,9c,d7,3e
53,30,c9,39,a3,90,13,b2,7f,a1,ba,10,af,5b,53,c8
b1,ae,6a,19,ed,2a,a3,3a,ec,8b,01,7c,50,9a,15,8b
7a,77,7b,28,b4,70,71,1f,77,40,c2,6b,22,0e,6e,fb
```

Some implementation requirements

The specification above is basically a functional requrements specification that says what your code must do. In addition, and as is common, we also have some implementation requirements that `MUST` also be met:

- Your implementation MUST honour the existing API defined in `cs2014coin.h`
- Your implementation MUST honour the existing command line arguments defined in `cs2014coin-main.c`
- Coins produced by your implementation MUST be verifiable using the verification implementation you've been given. (That is, you cannot just change the verifier to win the game:-)
- Your implementation of a coin miner SHOULD honour the CS2014COIN_MAXITERS value defined in the API - that means your code ought exit with an error if it doesn't find a coin after that number of iterations of nonce values.
- You SHOULD implement your coin miner in one .c file, it'd make sense to keep using the `cs2014coin-make.c` file and just add your code to that.

Some hints...

Here's a few hints to help you with your mining code:

- There are examples in `$BUILD/mbedtls-2.6.0/programs/pkey/` that should help you figure out how to use the mbed TLS APIs. (But you've a bunch of stuff to figure out too!) Don't only stare at that code - build, run and test it too.
- mbed TLS functions you will likely want to use will include:
  - `mbedtls_ctr_drbg_seed`
  - `mbedtls_ecp_gen_key`
  - `mbedtls_md_starts`
  - `mbedtls_md_update`
  - `mbedtls_md_finish`
  - `mbedtls_pk_sign`
- There are also various `mbedtls_*_init` and `mbedtls_*_setup` functions related to the above that you'll need to call to get those to work properly.
- Using network byte order means calls to `htonl` and `ntohl` are needed, in case the miner's and verifier's machines have different endianness.
- The `hexdump` (aka `hd`) tool will help you see what you're putting in files, e.g. to check if your lengths are or are not in network byte order.
- Writing your code to test with a small value for "bits" (say 5) will help
- Until your code seems to be working, limiting the iterations to a small number (say 2) will help you debug your stuff
- You'll likely need to debug with `gdb` for this one. We'll chat about that in class.
- My working implementation of `cs2014coin-make.c` has 250 lines in the file, including comments and debugging code. Yours shouldn't be too much different to that.
- Understanding the verification code will help you write the mining code, so don't ignore that. You can run it and debug it using the sample coin.
- Once you have a working coin miner, then you should be able to use the `check-timing.sh` shell script to see if your performance, (in terms of iterations of attempted proof-of-work), is roughly similar to openssl's sha256 (in hashes per second) on your development box. They should be close - less than an order of magnitude apart.

A typedef for our coins...

The `cs2014coin.h` header file defines the API you have to use, and includes the `typedef struct` below. You can, but don't have to use that in your code. (There are no copies of, or pointers to, instances of that structure passed in the API, so you don't have to use it.)

But we'll look at it now anyway...

```
/*!
 *       *
 * This structure describes a cs2014 coin.
 * Fields are flattened as usual, lengths use network byte order.
 * The hash is over the fields that preceed it in the struct.
 * The rightmost 'bits' bits of the hash value must be zero.
 * The signature is over the fields that proceed it in the struct.
 * All length fields, except 'bits' are in octets
 *
 */
typedef struct cs2014coin_t_defn {
    int ciphersuite; /// specifies all algorithms
    int bits; /// specifies the zero-bit run length needed in the hashval
    int keylen; /// length of the public key
    unsigned char *keyval; /// public key value
    int noncelen; /// length of nonce
    unsigned char *nonceval;
    int hashlen; /// length of hashval
    unsigned char *hashval; /// hash value with 'bits' of the LSBs having a value of zero
    int siglen; /// signature length
    unsigned char *sigval; /// signature value
} cs2014coin_t;
```

If I wanted to use that in some code, then I'd declare a variable

```
{foo```}

        cs2014coin_t mycoin;

And I can access (read/set) the field values like this:

        mycoin.ciphesuite==CS2014COIN_CS_0;

When I declare such a variable on the stack it'll consume a pile
of memory ( 6 ints and 4 pointers, so maybe ~48 bytes total or
more if the compiler aligns things specially).
But your code may be a good bit tidier if you use such a variable
rather than have a load of separate variables (which won't really
be much more memory efficient in many cases). And tidier
code is easier to maintain etc. which is a good thing (tm).

If I wanted to pass a value like mycoin (say after it's been
fully populated) to some function, ```print_coin()``` then I
might declare that function like this:

        void print_coin (cs2014coin_t coin);


To call that with mycoin as an input I just pass the variable to the function
follows:

        print_coin(mycoin);

That's a little inefficient as the full structure is passed
on the stack. So we much more commonly pass a pointer to
the structure and hence would have a function like this
instead:

        void print_coin (cs2014coin_t *coin);

Now there's only one pointer passed on the stack which is
better, for large structs. To call that with mycoin as an
input I need to pass the address of mycoin and not the
value of mycoin, which I do using the &amp; character as
follows:

        print_coin(&mycoin);
```

```
Inside that function, to access (read/set) the values of
the fields we might have code like:


        if (coin->ciphersuite==CS2014COIN_CS_0) {
            printf("Default Cipersuite\n");
        }

There's also an *important* difference in passing
things or pointers to things to functions. Since
C passes everything *by value*, you have to
pass a pointer to that thing if the function
you're calling
needs to modify the value of an input.
(To make a parameter an in/out parameter in
e.g. doxygen terms.)

I like [this](https://denniskubes.com/2012/08/20/is-c-pass-by-value-or-reference/)
description of how C does pass by value.

But again, you might or might not want to use
this particular struct in doing assignment 3, I mainly
included it to have this discussion of *pass
by value* and *pass by reference*.

## What's here?

The files in this assignment directory you should see now are:

- [cs2014.coin](cs2014.coin) - a sample coin
- [cs2014coin-main.c](cs2014coin-main.c) - the main line code
- [cs2014coin.h](cs201coin.h) - the API definition
- [cs2014coin-int.h](cs201coin.h) - macros and function prototypes used internally by the API
      implementation
- [cs2014coin-util.c](cs2014coin-check.c) - some API utilities
- [cs2014coin-check.c](cs2014coin-check.c) - the API implementation code for coin checking
- [cs2014coin-make.c](cs2014coin-make.c) - a stub of the API implementation code for making coins - you'll
      write the code for this
- [check-timing.sh](./check-timing.sh) - a script to see how fast/slow we are vs. openssl
- [Makefile](Makefile)  - the Makefile to builld the above and link in the mbed TLS library
- [refman.pdf](./refman.pdf) - the Doxygen PDF documentation for this project
- [../assignment2/mbedtls-2.6.0](../assignment2/mbedtls-2.6.0/) - the directory with the mbed TLS stuff
- [../assignment2/mbedtls-2.6.0-apache.tgz](https://tls.mbed.org/download/start/mbedtls-2.6.0-apache.tgz)-
      the tarball you downloaded
- [README.html](README.html) - this HTML file
- [README.md](README.md) - the markdown source for this HTML file

## Noteworthy

There are a number of notewothy things in the code given to you for this assignment:

- We've seen our first use of an [RFC2119](https://tools.ietf.org/html/rfc2119) MUST
- We've used ```extern``` in the internal API stuff for error strings.
- We use ```getopt()``` - for details see ```man 3 getop
```

Do also look at this `example`

- The verification `code` attempts to be `constant time`. That code is nearly, but not quite, done - anyone interested in improving that is welcome to try and I'll be interested in what you find. (I might do a bit more on that myself as I need to learn how to do sometime:-)
- `dumpbuf()` etc is handy and fairly typical
- Things like `cs2014coin-int.h` and `cs2014coin-util.c` are typical too.
- Error string handling like this is sorta-but-not-that common
- The doxygen docs for this could be interesting to look at, see if you can make them?
- I'd have preferred to use `EdDSA`, and in particular `Ed25519`, for this, or at least `deterministic signatures`, but that'd have made this too hard an assignment. The reason is that ECDSA signature generation requires a random number, and if that's badly chosen, then the private signing key can leak (invalidating the entire point of signing). Such details are often not easily visible to you as a programmer, but can be critically important for the overall system produced. (The point here is about security, but similar issues arise for any emergent property of the system, such as performance or usability.)

Deadline

The deadline for submission of this assignment is 2017-10-30

Submission

For this assignment you should only submit your single file of source code, which can be called `cs2014coin-make.c`

To submit your assignment use `https://cs2014.scss.tcd.ie/` as usual.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 cs2014coin_t_defn Struct Reference

our basic cs2094 coin type

```
#include <cs2014coin.h>
```

Public Attributes

- int ciphersuite

    specifies all algorithms
- int bits

    specifies the zero-bit run length needed in the hashval
- int keylen

    length of the public key
- unsigned char $*$ keyval

    public key value
- int noncelen

    length of nonce
- unsigned char $*$ **nonceval**
- int hashlen

    length of hashval
- unsigned char $*$ hashval

    hash value with 'bits' of the LSBs having a value of zero
- int siglen

    signature length
- unsigned char $*$ sigval

    signature value

4.1.1 Detailed Description

our basic cs2094 coin type

This structure describes a cs2014 coin. Fields are flattened as usual, lengths use network byte order. The hash is over the fields that preceed it in the struct. The rightmost 'bits' bits of the hash value must be zero. The signature is over the fields that proceed it in the struct. All length fields, except 'bits' are in octets

The documentation for this struct was generated from the following file:

- cs2014coin.h

Chapter 5

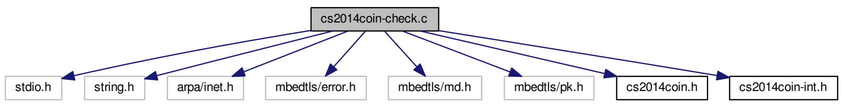
File Documentation

5.1 cs2014coin-check.c File Reference

This is the implemetation of the cs2014 coin checker.

```
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <mbedtls/error.h>
#include <mbedtls/md.h>
#include <mbedtls/pk.h>
#include "cs2014coin.h"
#include "cs2014coin-int.h"
```

Include dependency graph for cs2014coin-check.c:



Macros

- #define SB(___var___, ___shift___) (___var___|=(1<<___shift___))

    set a bit in a mask to indicate some error
- #define CB(___var___, ___shift___) (___var___&=∼(1<<___shift___))

    clear a bit in a mask to indicate that error wasn't seen

Functions

- int lc_memcmp (const void ∗a, const void ∗b, size_t size)

    a local constant time memcmp
- int cs2014coin_check (int bits, unsigned char ∗buf, int buflen, int ∗res)

    check a coin

5.1.1 Detailed Description

This is the implemetation of the cs2014 coin checker.

It should go without saying that these coins are for play:-)

This is part of CS2014 https://down.dsg.cs.tcd.ie/cs2014/examples/c-progs-2/README.html

5.1.2 Function Documentation

5.1.2.1 cs2014coin_check()

```
int cs2014coin_check (
            int bits,
            unsigned char * buf,
            int buflen,
            int * res )
```

check a coin

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	specifies the input coin size
res	contains the result of checking the coin

Returns

zero for success, non-zero for fail (note: success != good coin!)

Check a coin of the required quality/strength Note we attempt to be roughly constant time here, just for fun But, if this were called on recovered plaintext, that might be significant! See https://cryptocoding.net/index.php/Coding_rules TODO: have a dummy public key ready for this so we do sig check of some sort

5.1.2.2 lc_memcmp()

```
int lc_memcmp (
            const void * a,
            const void * b,
            size_t size )
```

a local constant time memcmp

Parameters

a	is buffer 1
b	is buffer 2
size	is the length of both

Returns

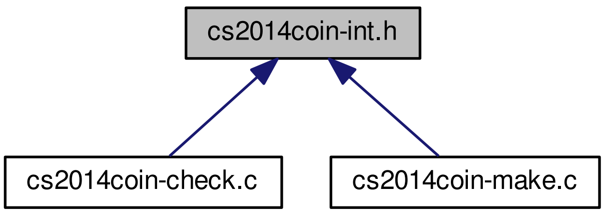
0 for same, 1 for not same

A constant time memcmp, found at: `https://github.com/OpenVPN/openvpn/commit/11d21349a4e7e38a025849`
Note - this was done due to a real attck on OpenVPN `https://www.rapid7.com/db/vulnerabilities/alpine-linu`

5.2 cs2014coin-int.h File Reference

Definitions/macros used internally.

This graph shows which files directly or indirectly include this file:



Macros

- #define CC_BUFSIZ 16000

  turns on some debug printing
- #define CC_DEFERR "Bummer, no idea what went wrong there"

  a generic error
- #define CC_GENERR "Some kind of fairly generic error happened"

  another generic error
- #define **CC_TOOLONG** 1
- #define **CC_DRBGCRAP** 2
- #define **CC_KEYGENFAIL** 3
- #define **CC_ITERS** 4
- #define **CC_BADCIPHERSUITE** 5

Functions

- void dumpbuf (char ∗msg, unsigned char ∗buffer, int buflen)

  debug printer for buffers
- int zero_bits (int bits, unsigned char ∗buf, int buflen)

  check if rightmost N bits of a buffer are zero

Variables

- const char ∗ errstrs [ ]

  an array of speciic strings - CC_foo #define'd values are index the strings in this array

5.2.1 Detailed Description

Definitions/macros used internally.

It should go without saying that these coins are for play:-)

This is part of CS2014 `https://down.dsg.cs.tcd.ie/cs2014/examples/c-progs-2/READM←E.html`

5.2.2 Macro Definition Documentation

5.2.2.1 CC_BUFSIZ

```
#define CC_BUFSIZ 16000
```

turns on some debug printing

turns on loadsa debugging we'll use this on stack to save coding mallocs etc

5.2.3 Function Documentation

5.2.3.1 dumpbuf()

```
void dumpbuf (
             char * msg,
             unsigned char * buffer,
             int buflen )
```

debug printer for buffers

Parameters

buffer	is where stuff's at buflen is how many octets

Returns

void

debug printer for buffers

Parameters

buffer	is where stuff's at buflen is how many octets

Returns

void

5.2.3.2 zero_bits()

```
int zero_bits (
            int bits,
            unsigned char * buf,
            int buflen )
```

check if rightmost N bits of a buffer are zero

Parameters

bits	is the value of N (in bits)
buf	is the buffer
buflen	is the length

Returns

1 if those N bits are all zero, 0 otherwise

be better if this were more efficient

5.2.4 Variable Documentation

5.2.4.1 errstrs

```
const char* errstrs[]
```

an array of speciic strings - CC_foo #define'd values are index the strings in this array

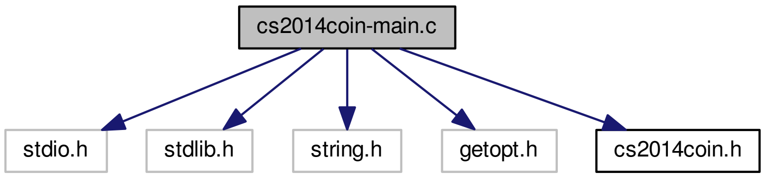
an array of speciic strings - CC_foo #define'd values are index the strings in this array

5.3 cs2014coin-main.c File Reference

This is the main function for cs2014 coin handling.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include "cs2014coin.h"
```

Include dependency graph for cs2014coin-main.c:



Macros

- #define MAXZEROBITS 20
  the API we want...
- #define DEFZEROBITS 17
  the default length of zero-bit run
- #define MODE_MAKE 0
  mode where we make a coin
- #define MODE_CHECK 1
  mode where we check a coin looks feasible
- #define DEFFNAME "cs2014.coin"
  default file name

Functions

- void **usage** (char ∗progname)
- int main (int argc, char ∗argv[ ])

5.3.1 Detailed Description

This is the main function for cs2014 coin handling.

This is part of CS2014 `https://down.dsg.cs.tcd.ie/cs2014/examples/c-progs-2/READM↩ E.html`

5.3.2 Macro Definition Documentation

5.3.2.1 MAXZEROBITS

```
#define MAXZEROBITS 20
```

the API we want...

the max length of zero-bit run we can support

5.3.3 Function Documentation

5.3.3.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

buffer for a coin

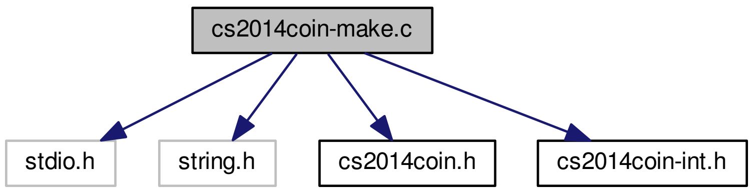
string for file name

5.4 cs2014coin-make.c File Reference

This is the implementation of the cs2014 coin maker.

```
#include <stdio.h>
#include <string.h>
#include "cs2014coin.h"
#include "cs2014coin-int.h"
```

Include dependency graph for cs2014coin-make.c:



Functions

- int cs2014coin_make (int bits, unsigned char ∗buf, int ∗buflen)

  make a coin

5.4.1 Detailed Description

This is the implementation of the cs2014 coin maker.

It should go without saying that these coins are for play:-)

This is part of CS2014 https://down.dsg.cs.tcd.ie/cs2014/examples/c-progs-2/READM↩ E.html

5.4.2 Function Documentation

5.4.2.1 cs2014coin_make()

```
int cs2014coin_make (
             int bits,
             unsigned char * buf,
             int * buflen )
```

make a coin

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	is an in/out parameter reflecting the buffer-size/actual-coin-size

Returns

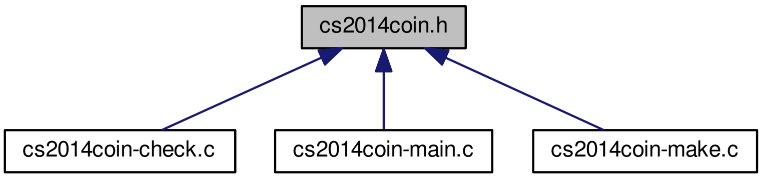
the random byte

Make me a coin of the required quality/strength

5.5 cs2014coin.h File Reference

This is the external i/f for the cs2014 coin example.

This graph shows which files directly or indirectly include this file:



Classes

- struct cs2014coin_t_defn

  our basic cs2094 coin type

Macros

- #define **CS2014COIN_BUFSIZE** 1024
- #define **CS2014COIN_PROGRESS**
- #define **CS2014COIN_GOOD** 0
- #define **CS2014COIN_BAD** 1
- #define **CS2014COIN_GENERR** -1
- #define **CS2014COIN_MAXITER** 1024∗1024
- #define **CS2014COIN_CS_0** 0

Typedefs

- typedef struct cs2014coin_t_defn cs2014coin_t

  our basic cs2094 coin type

Functions

- const char ∗ cs2014coin_err (int errno)

  return out best guess error string
- int cs2014coin_make (int bits, unsigned char ∗buf, int ∗buflen)

  make a coin
- int cs2014coin_check (int bits, unsigned char ∗buf, int buflen, int ∗res)

  check a coin

5.5.1 Detailed Description

This is the external i/f for the cs2014 coin example.

It should go without saying that these coins are for play:-)

This is part of CS2014 `https://down.dsg.cs.tcd.ie/cs2014/examples/c-progs-2/READM↩E.html`

5.5.2 Typedef Documentation

5.5.2.1 cs2014coin_t

```
typedef struct cs2014coin_t_defn cs2014coin_t
```

our basic cs2094 coin type

This structure describes a cs2014 coin. Fields are flattened as usual, lengths use network byte order. The hash is over the fields that preceed it in the struct. The rightmost 'bits' bits of the hash value must be zero. The signature is over the fields that proceed it in the struct. All length fields, except 'bits' are in octets

5.5.3 Function Documentation

5.5.3.1 cs2014coin_check()

```
int cs2014coin_check (
             int bits,
             unsigned char * buf,
             int buflen,
             int * res )
```

check a coin

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	specifies the input coin size
res	contains the result of checking the coin

Returns

zero for success, non-zero for fail (note: success != good coin!)

Make me a coin of the required quality/strength

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	specifies the input coin size
res	contains the result of checking the coin

Returns

zero for success, non-zero for fail (note: success != good coin!)

Check a coin of the required quality/strength Note we attempt to be roughly constant time here, just for fun But, if this were called on recovered plaintext, that might be significant! See `https://cryptocoding.net/index.php/Coding_rules` TODO: have a dummy public key ready for this so we do sig check of some sort

5.5.3.2 cs2014coin_err()

```
const char* cs2014coin_err (
             int errno )
```

return out best guess error string

Parameters

errno	is an errno, presumably returned from somewhere else in this API

Returns

error string (a const)

Error string handler. Good to use unique error numbers for all different conditions

5.5.3.3 cs2014coin_make()

```
int cs2014coin_make (
             int bits,
             unsigned char * buf,
             int * buflen )
```

make a coin

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	is an in/out parameter reflecting the buffer-size/actual-coin-size

Returns

zero for success, non-zero for fail (note: success != good coin!)

Make me a coin of the required quality/strength

Parameters

bits	specifies how many bits need to be zero in the hash-output
buf	is an allocated buffer for the coid
buflen	is an in/out parameter reflecting the buffer-size/actual-coin-size

Returns

the random byte

Make me a coin of the required quality/strength

Index