Encryption:
The ChaCha20 Symmetric Stream Cipher

Michael Brockway

March 5, 2018

Overview

Transport-layer security employs an asymmetric public cryptosystem to allow two paries (usually a client application and a server) to authenticate each other and negotiate, among other things, a shared key for use in a symmetric cryptosystem for fast data encryption.

An example public-key system will be outlined in another lecture.

The present lecture gives an example of a fast symmetric system.

- *AES* (advanced encryption standard, *Rijndael* has been the symmetric system of choice in TLS some years now. Its design and implmentation are somewhat technical (mathematically and computationally) but a link to some notes about it are given in the FutherRead section at the end.
- *ChaCha20* is a recently devloped *stream cipher* approved for use in TLS and is straightforward to describe in a single lecture.

Block versus Stream Ciphers

These are two design philosophies for ciphers. A block cipher

- divides the plain- or cipher-text into *blocks* of a fixed number of bytes.
- The algorithm defines a function (dependent on a *round key*) to encrypt a block, usually iterating this over several *rounds*.
- The round keys are derived from the master key by a *key scheduling* algorithm.
- The cipher operates on the text as a sequence of blocks. The algorithm for encrypting a block generally depends in some way on the output of encryption of the previous block.
- Encryption and decryption both work like this.
- AES (Rijndael) is built like this. The first item in 'further reading' gives an example. One round of processing of a block involves applying four helper functions one after the other. They are invertible, and decryption consists in applying the inverse functions in reverse order.

Block versus Stream Ciphers

A stream cipher

- does not block the input, but just works on the text as a *stream* of bytes.
- A common approach is, given a *key* and an *initial vector* (a short sequence of bytes) to generate a long stream bytes which look 'random' but are not, being entirely determmined by the key and initial vector.
- In this kind of approach, text is simply XORed bit by bit with the generated byte stream.
- XORing twice cancels itself out: $(b \; XOR \; b') \; XOR \; b'$ is just b back again;
- so a text encrypted in this way can be decrypted be repeating the process using the same key and initial vector.

XOR (exclusive OR)

You can check this from the *truth table* of XOR:

XOR	0	1
0	0	1
1	1	0

Alternatively, for any bit b, (b *XOR* 0) is just b while (b *XOR* 1) is $\sim b$.

For efficiency the data streams are generally handled a byte or a *word* at a time: a word is, with modern hardware, 32 or 64 bits. Bitwise XOR of a word is a single machine instruction.

ChaCha20 Overview

- Developed by David Bernstein from an earlier cipher, Salsa20.
- Both ciphers are 'ARX': the operations are combinations of addition, rotation of bits, and exclusive-OR.
- It is claimed that this approach can produce sufficient 'mangling' of the input without the need for S-boxes or similar, and as the result, encryption is fast and efficient.
- The design and implementation of Salsa20 are described by Bernstain in design [click] and specification [click]. The latter is clear enough for anyone to write their own implmentation or Salsa20 - a good programming exercise!
- ChaCha20 is introduced by Berstein as a development of Salsa here [click].

The URLs of these papers are listed in 'further reading' below where there are also links to local copies.

ARX

The X stands for XOR as above, perfomed bitwise on 32-bit words.

- Thus 0x48a5d72e XOR 0x234a79bb = 0x6befae95.

A is for *addition*. 32-bit words are added as unsigned integers. If the result does not fit into 32 bits, the extra 'carry' is discarded: so this is effectively addition modulo 2^{32}.

- 0x78a5d72e + 0xf34a79bb = 0x6bf050e9. (Carried 1 discarded.)

ARX

R is for *rotation*. a 32-bit word can be *rotated* left or right some number of bits. This is like a *shift* but the bit that are shift-out of the register 'run around the back' and shift in.

Thus, in binary, the byte `10110100` rotated left 3 bits ends up as `10100101`. The blue bits ran around the back.

Rotations work similarly with (unsigned) shorts, ints (words), longs. They can be defined in terms of shifts:

- left-rotation of x by n bits is `(x << n) | (x >> (w-n))`
- right-rotation of x by n bits is `(x >> n) | (x << (w-n))`
- w is the width: 32 for 32-bit words.
- These are *logical* shifts. In java you would use >>> rather than >>.

0x78a5d72e ...

- rotated left 7 bits is 0x52eb973c;
- rotated right 13 bits is b973c52e.

ChaCha20

The cipher, the form used in transport-layer security, is described in detail in RfC 7539[click] see sections 2.1 – 2.4.

Although it is a *stream* cipher, the algorithm actually generates the stream 64 bytes at a time, and encrypts a stream of bytes 64 at a time by XORing with these, generating another lot of 64 bytes as it needs to. Do not be distracted with the term 'block' in the literature!

The data is handled internally in 32-bit words (unsigned integers), 16 words (64 bytes) at a time.

- A *state* of the cipher is an array of 16 words

The '20' signifies that the cipher runs in 20 rounds.

- The basic 'unit' the the algorithm is the *quarter-round*,
- Four of these are combined to make a *column-round*
- and four are combined in another way to make a *diagonal-round*.
- Ten column-rounds alternate with ten diagonal-rounds to make 20 *rounds*.

ChaCha20 Quarter-round

The basic function is

```
void quarterround(word * a, word * b, word * c, word * d) {
  *a += *b; *d ^= *a; *d = lRot(*d,16);
  *c += *d; *b ^= *c; *b = lRot(*b,12);
  *a += *b; *d ^= *a; *d = lRot(*d, 8);
  *c += *d; *b ^= *c; *b = lRot(*b, 7);
}
```

Four words pointed at by a, b, c, d are updated by an ARX combination of additions, XORs and left-rotations.

This function is applied to a state (array of 16 words) by picking four indexes into the array to serve as quarter-round parameters:

```
void qrOnState(word * state, int h, int i, int j, int k) {
  quarterround(state+h, state+i, state+j, state+k);
}
```

Column-Quarter-round

ChaCha applies this function to a state in two different ways. First, the column-round:

```
void columnRound(word * state) {
  qrOnState(state, 0, 4,  8, 12);
  qrOnState(state, 1, 5,  9, 13);
  qrOnState(state, 2, 6, 10, 14);
  qrOnState(state, 3, 7, 11, 15);
}
```

Think of the state as

s[0]	s[1]	s[2]	s[3]
s[4]	s[5]	s[6]	s[7]
s[8]	s[9]	s[10]	s[11]
s[12]	s[12]	s[14]	s[15]

and you may see why this is called ‘column-round’

Diagonal-Quarter-round; Double-Round

The diagonal-round is, similarly,

```
void diagRound(word * state) {
  qrOnState(state, 0, 5, 10, 15);
  qrOnState(state, 1, 6, 11, 12);
  qrOnState(state, 2, 7,  8, 13);
  qrOnState(state, 3, 4,  9, 14);
}
```

(Can you relate this to the 4x4 picture of a state?)

Now a *double-round* is

```
void doubleRound(word * state) {
  columnRound(state);
  diagRound(state);
}
```

8 quarter-rounds = a double-round.
ChaCha20 runs 10 iterations of this on a state (hence 20 rounds).

ChaCha20 state initialisation

- A 32-byte *key* and a 12-byte *initial vector* (*nonce*) are provided as arrays of bytes.
- These are converted to arrays of 8 (32-bit, 4-byte) words and 3 words resectively: each 4 bytes makes a word in *little-endian fashion*.

The *initial state* is the built thus

- state[0..3] are word-size constants `0x61707865, 0x3320646e, 0x79622d32, 0x6b206574`;
- state[4..11] are are the key derived from the little-endian byte array;
- state[12] is a word-sized *block counter*. Typically it is set to 1 at the beginning and incremented every time the algorithm is required to generate another output block.
- state[13..15] are the initial vector derived from the little-endian byte array.

ChaCha20 block function

```
void chaCha20Block(byte* key, byte* nonce, word blkCt, byte* output) {
  word state[16], state0[16];
  initState(key, nonce, blkCt, state); //initialise the state

  int i;
  for (i=0; i<16; i++)
    state0[i] = state[i];  //... make a copy of the state
  for (i=0; i<10; i++)
    doubleRound(state);    //... do 10 double-rounds

  for (i=0; i<16; i++)     //... add the origianal state word-by-word
    state[i] += state0[i];

  //convert result to a sequence of 64 bytes for output:
  for (i=0; i<16; i++)
    invLittleEndian(state[i], output + i*4);
}
```

NB Initialisation of the state is as on previous slide;
Addition is word-wise, mod 2^{32}.

ChaCha20 block function

We now have an algorithm which, given a key and an initial vector and a block-count value, will produce an array of 64 bytes: namely the byte-format version of the state after initialisastion and 10 double-rounds.

This will produce a long stream of bytes, if we initialize the block-counter, say, to 1 and iterate, incrementing the block-counter.

- We get 64 bytes each iteration
- The block counter can count up to 2^{32}.
- 2^{38} bytes: 256 GiB

ChaCha20 encryption

To encrypt a sequence of up to 2^{38} bytes, (given a key and initial vector as above),

- Initialise block count and generate 64 bytes with the block function;
- while (there are ≥ 64 bytes of input text left)
  - XOR the next 64 text bytes with these;
  - increment block count and generate another 64 bytes with the block function.
- Now there are < 64 text bytes left to do; truncate the generated stream to this number and XOR with the text bytes.

Further reading

- A paper On AES/Rijndael, for your interest: http://computing.northumbria.ac.uk/staff/cgmb3/teaching/cryptography/RijndaelAES.pdf
- Bernstein's paper on design of Salsa20: https://cr.yp.to/snuffle/design.pdf
- Bernstein's paper on implmentation of Salsa20: https://cr.yp.to/snuffle/spec.pdf
- Bernstein's paper on ChaCha20: https://cr.yp.to/chacha/chacha-20080128.pdf
- RfC 7539 on ChaCha as implemented for TLS: https://tools.ietf.org/html/rfc7539

There are local copies of the Salsa and ChaCha papers at http://computing.northumbria.ac.uk/staff/cgmb3/teaching/cryptography/index_crypto.html

ZIPs of the ChaCha implementation discussed in the lecture (and the Salsa implementation) are availble here also.