Cryptography Tutorial

Joe Armstrong

Contents

1 Introduction

This is "work in progress" - if you wish to contribute please refer to the archive at http://github.com/joearms/crypto_tutorial. All the code for this tutorial is in the archive.

Warning: Do not use this code in production – Code like this is fine for *understanding* cryptographic algorithms but it is not intended for large-scale production. For production system more in-depth knowledge of cryptography is required. I'm fully aware that AES256 and 1024 bit RSA keys are not considered "state of the art"[1] but for the purposes of *understanding the algorithms* a 32 bit RSA key suffices!

Always remember that the security of a system depends upon the weakest link, which in many cases boils down to key-management[2].

1.1 Understanding not code

In this tutorial, I'm going to introduce a number of simple cryptographic techniques that can be used to build secure applications. The main goal of the tutorial is to understand a small number of well-known cryptographic techniques and to be able to apply that knowledge.

Wherever possible I have tried to implement the algorithms in pure Erlang in as few lines of code as possible. This is possible for algorithms such as RSA, where the core RSA algorithm can be coded in a handful of lines of code. This fits in with my philosophy of programming. I think it is better to completely understand RSA implemented using Erlang bignums, rather than

[1] So don't bother to tell me, I know!

[2] So don't loose your passwords, or give them away!

not understand a fast RSA algorithm written in thousands of lines of assembler and using hardware acceleration which I cannot myself validate.

Algorithms such as AES256 are not written in Erlang (since the algorithm itself is not very exciting) and I assume that the implementation in the Erlang Crypto libraries are correct.[3]

Once you have understood the algorithms you will be able to make informed decisions over which algorithm to use and how to combine them. If efficiency is your goal over personal understanding, then you might like to trust some third party library, so I'll wind up with a introduction to NACL.

In a tutorial like this the problem is "what to leave out" rather than "what to include" – so I've left out a lot of detail.

In many cases, distributed algorithms which involve the interaction between a client and server have been programmed as interactions between pairs of Erlang processes. This is so we can concentrate on the algorithms themselves and not be concerned with details of how data is transported between clients and servers. For our purposes it suffices to assume that data can be sent between clients and servers and not *how* that data is transmitted. Turning distributed algorithms, involving message passing between processes, can easily be transformed to algorithms that uses TCP sockets or some other communication mechanism to transmit the data.

1.2 The Basic Methods

I'll go into these in detail:

- RSA.
- AES256.
- SHA1.
- Padding.
- Salting.

[3] Actually this claim would be difficult to verify, since the Erlang crypto libraries make use of OpenSSL and this library is many thousands of lines of (to me) incomprehensible C code, and has also had buggy code in it in the past.

- Secret sharing.

1.3 Applications

Applications are made by combining the basic methods. I'll deal with:

- Challenge response algorithms
- Authentication
- Secret message streams
- Transport Layer Security
- Self-certifying File Systems

As you'll see, no one technique dominates. To make an application, we will have to make several design choices. We'll choose a cryptographic strength[4] and a set of algorithms that work together.

1.4 Warning

I guess, I should give a word of warning before starting. I'm not a cryptography expert and the code I present has not been subject to review - on the other hand the algorithms are so simple and they closely follow the math that hopefully, they are correct.

I had a choice - I could *trust* other people's code that I cannot review and understand myself, or I can write my own code that I do understand. For algorithms like RSA, where the math is simple and the code borders on the trivial, I choose the latter.

For algorithms like AES and SHA1, I'll trust that the reference implementations are correct and secure.

In defense of my position, I'd point out that many well known crypto algorithms in the public domain have been subject to large scale peer-review

[4] How many bits in the keys, how paranoid are we?

despite this, they have been found to contain security holes. One of the responses to this (not the only) is that they are written in imperative language like C *which are extremely difficult to understand*.[5]

As a final note, I'd say that no cryptographic system is perfect – you just have to decide how much effort an adversary is prepared to put into breaking your system. If you were designing a system to manage financial transactions you should put a lot of effort into making it secure. If you want to build a flower-arranging website you'd be less bothered.

2 Techniques

Certain common techniques crop up over and over again in different crypto applications. Real applications are made by combining small fragments of code which do some rather specific things. The whole can appear complicated, but this is mainly because combining many small things can create the illusion of complexity, when in fact the systems are in principle rather simple. We'll see how this works as we progress throughout this tutorial.

2.1 Salting

If we encode the same thing many times we would like to get different crypto text for each encoding. This is to make the attacker's life more difficult. A common way to do this is to add "salt" to the password that is used to encrypt the data. The salt (which is a random number) is transmitted together with the crypto text; this ensures that if the same text is encrypted many times each encryption will result in different crypto text.

Note: Encrypting known fragments of text in the input text helped crack the enigma machine in WWII and lead to the development of computers.

[5] In my opinion virtually all C is extremely difficult to understand, FPLs with no mutable state that closely follow the crypto math are far easier to understand. By the time you get to the end of this tutorial I hope you'll agree with me.

```
Encrypt with: Key ++ Salt
Transmit:     Salt ++ Encrypted text
```

2.2 Envelopes

Often we apply several transformations to the input data. We salt the data, then pad it and encrypt it and so on. To recover the data we perform the steps in the opposite order to which the transformations were applied. This is very simple if each step is itself invertible. So, if we encode data by performing a set of transformations:

$Out = F(G(H(I(In))))$

Then all we have to do is invert each step:

$In = I^{-1}(H^{-1}(G^{-1}(F^{-1}(Out))))$

In Erlang to encode some data we might do something like:

```
Bin1 = encrypt(Bin, SymKey),
Sha = sha1(Bin1),
Bin2 = term_to_binary({packet, Sha, Bin1}),
```

To decoding this we'd do the steps in the opposite order:

```
{packet, Sha, Bin1} = binary_to_term(Bin2),
Bin = decode(Bin1, SymKey),
case sha1(Bin1) of
    Sha -> ...;
    _   -> exit(bad_packet)
end
```

3 Symmetric Algorithms

We'll start with the simplest of algorithms. These use the same key for both encryption and decryption. These are called "symmetric" algorithms. We'll look at a number of different symmetric algorithms, the first few (one-time pads and LCGs) are "toy" implementations and just here for illustrative purposes. For production applications, some AES variant or RC4 would be a better choice.

3.1 One time pad

A one time pad is a pre-computed sequence of random bytes that both the sender and receiver have agreed upon. It is used once xoring the bits in the message with the bytes in the one-time pad.

Here's an example of a one time pad:

symmetric.erl

```
pad() ->
    "12ishiyr72TY873TGKY8HHAE7YT8YsadaHGFIYLIasdjasdgjasgd".

encrypt_0([H|T], [H1|T1]) ->
    [H bxor H1 | encrypt_0(T, T1)];
encrypt_0(_, []) ->
    [].
```

The one time pad is the return value of the function `pad/0`. The function `encrypt_0(Pad, Str)` encrypts the `Str` using the characters in `Pad`:

```
1> Pad = symmetric:pad().
"12ishiyr72TY873TGKY8HHAE7YT8YsadaHGFIYLIasdjasdgjasgd"


2> C = symmetric:encrypt_0(Pad, "hello joe").
[89,87,5,31,7,73,19,29,82]
```

And we decrypt using the same pad:

```
3> symmetric:encrypt_0(Pad, C).
"hello joe"
```

One time pads are extremely secure provided we can securely distribute the pad to both parties in advance. There is no algorithm to crack.

3.2 Xor Text with a stream of random bytes

Our next method generates a stream of random bytes using a linear congruent generator (LCG) and then XORS the byte stream with the data to be encrypted.

To decrypt the data we just XOR the encrypted data with the same byte stream to recover the original data. This works because:

```
(M xor R) xor R = M
```

A LCG generates random numbers with a recurrence relation of the form `X[k+1] = (aX[k] + c) mod n` the WikiPedia page https://en.wikipedia.org/wiki/Linear_congruential_generator gives a number of values for `a c` and `n`

`mod` is called `rem` in Erlang.

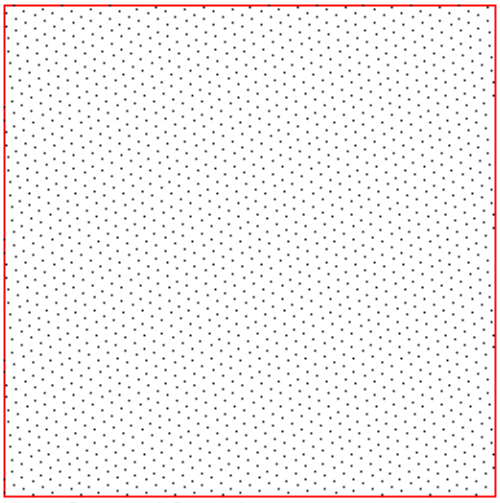
We can easily turn a LCG into an encryption routine like this:

symmetric.erl

```
encrypt_1(Password, Str) ->
    X0 = password_to_int(Password),
    encrypt_1(X0, Str, []).

password_to_int(Str) ->
    erlang:phash(Str, 4294967296).

encrypt_1(X0, [H|T], L) ->
    X1  = lcg(X0),
    Ran = next_byte(X0),
    encrypt_1(X1, T, [H bxor Ran|L]);
encrypt_1(_, [], L) ->
    lists:reverse(L).

next_byte(N) ->
    (N bsr 4) band 255.

lcg(X) ->
    A = 1664525,
    C = 1013904223,
    M = 4294967296,   %% 2^32
    (X*A + C) rem M.
```

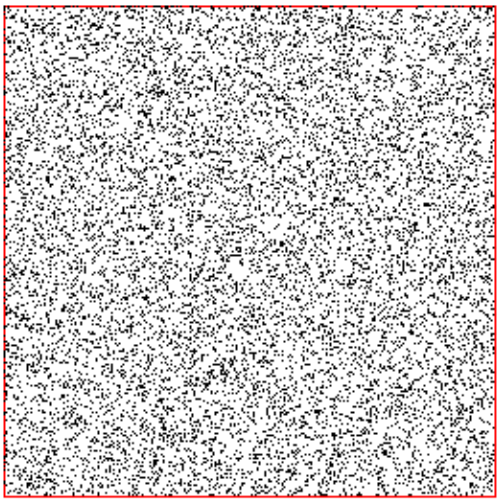
Note this code is for illustration only, this would be very easy to break so don't use it in practice.

To get an idea of how good the LCG we have used, we can generate pairs of random bytes, and use them as the `X` and `Y` coordinates of points in a 2-D 256 x 256 scatter plot. The resulting plot is as follows:



Linear Congruent Generator

As you can see, the result is not very random. Using `crypto:rand_bytes(K)` we obtain:



Erlang crypto:rand_bytes()

Which looks much better. The code for this can be found in `plot_random.erl` in the project archive.

3.3 Adding Salt

The problem with the previous algorithm is that if we encrypt the same text many times with the same password the encrypted text is always the same.

To remedy this, we generate a random string each time and append it to the password:

symmetric.erl

```
encrypt_2(Password, Txt) ->
    Salt = random_session_key(),
    X0   = password_to_int(Salt ++ Password),
    encrypt_1(X0, Txt, lists:reverse(Salt)).
```

To decrypt the data we need to extract the salt before decryption:

symmetric.erl

```
decrypt_2(Password, Str) ->
    {Salt, Str1} = lists:split(10, Str),
    X0 = password_to_int(Salt ++ Password),
    encrypt_1(X0, Str1, []).
```

Note that in both encryption and decryption we reused the code that encrypted the original data – salting is done with a small wrapper around the original code.

3.4 AES256

LCGs are pretty poor sources of random numbers, I've just used them here for illustration. A better symmetric algorithm is AES256 which is part of the Advanced Encryption Standard. AES assumes the data to be encrypted is a multiple of 16 bytes log and requires salting. A simple interface the the Erlang crypto application is in the module `ez_crypt_aes.erl`

ez_crypt_test.erl

```
aes_test() ->
    Password = <<"secret">>,
    Data = make_binary(2000), %% random binary
    C1   = ez_crypt:aes_encrypt(Password, Data),
    Data = ez_crypt:aes_decrypt(Password, C1),
    C2   = ez_crypt:aes_encrypt(Password, Data),
    false = (C1 == C2),
    ok.
```

Encrypting the same data twice gives a different crypto text, so the AES library is "self salting".

3.5 Stream Encryption

Encryption can operate in two modes:

- Batch encryption – all the data to be encrypted is available at the same time and the data size is relatively small.
- Stream encryption – the data is encrypted in chunks and we use a synchronized pair of senders and receivers.

Stream encryption is typically used when the data to be encrypted is huge or for things like streaming media – where the stream can be considered "infinite."

This kind of code has an initialization step:

```
S0 = crypto:stream_init(Type, Password)
```

`S0` is an initial state. When new data `Bin` is to be encrypted we call:

```
{S1, C1} = crypto:stream_encrypt(S0, Bin)
```

`C1` is the crypto text and `S1` is the new state of the encrypter which must be used in the next encryption call.[6]

To illustrate stream encryption we can set up a pair of processes and set up a stream encryption channel between them:

Typical client code looks like this:

stream.erl

```
client(Password, Server) ->
    S0 = crypto:stream_init(rc4, Password),
    client_loop(S0, Server).

client_loop(S0, Server) ->
    receive
        {send_server, Bin} ->
            {S1, B1}= crypto:stream_encrypt(S0, Bin),
            Server ! {data, B1},
            client_loop(S1, Server);
        stop->
            Server ! stop
    end.
```

The server code follows the same pattern, only now we use `stream_decrypt` instead of `stream_encrypt`

stream.erl

```
server(Password, Parent) ->
    S0 = crypto:stream_init(rc4, Password),
    Parent ! server_loop(S0, <<>>).

server_loop(S0, B) ->
    receive
        {data, Bin} ->
            {S1, B1} = crypto:stream_decrypt(S0, Bin),
            server_loop(S1, <<B/binary, B1/binary>>);
        stop ->
```

[6] Yes it's a Monad!

```
            B
    end.
```

I've include a small test hardness so we can run the code.

stream.erl

```
test() ->
    Password = <<"secret">>,
    Server = spawn(?MODULE, server, [Password, self()]),
    Client = spawn(?MODULE, client, [Password, Server]),
    Client ! {send_server, <<"hello">>},
    Client ! {send_server, <<" world">>},
    Client ! stop,
    Got = receive X -> X end,
    true = (Got == <<"hello world">>),
    hooray.
```

This code is very simple. To run this in a real application we'd use a socket TCP interface and a "middle man" pattern.[7]

4 Hashing and Padding

Before we move to public key algorithms, we'll have a quick look at hashing and padding, since we'll need these in the next chapter.

[7] Read my Erlang Book to see how :-)

4.1 Hashing

> A cryptographic hash function is a hash function which is considered practically impossible to invert, that is, to recreate the input data from its hash value alone. These one-way hash functions have been called "the workhorses of modern cryptography".[1] The input data is often called the message, and the hash value is often called the message digest or simply the digest.
> The ideal cryptographic hash function has four main properties:
>
> - it is easy to compute the hash value for any given message.
> - it is infeasible to generate a message that has a given hash.
> - it is infeasible to modify a message without changing the hash.
> - it is infeasible to find two different messages with the same hash.
>
> Quote From: `http://en.wikipedia.org/wiki/Cryptographic_hash_function`

SHA1 is one of the most commonly used cryptographic hash algorithms. It produces a 120 bit hash of a data set. SHA1 is part of the Erlang standard libraries.

There are two ways of calling it:

```
digest1() ->
    crypto:hash(sha, "hello world").

digest2() ->
    S0 = crypto:hash_init(sha),
    S1 = crypto:hash_update(S0, "hello "),
    S2 = crypto:hash_update(S1, "world"),
    crypto:hash_final(S2).
```

The first example can be used when the data involved is small. The second where the data concerned is large. For example, if we wanted to compare digital images of a few MBytes we could use the first method, but to compute the SHA1 checksum of a GByte movie we would use the second method with code like the following:

ez_crypt.erl

```
file2sha(File) ->
    hash_file(File, sha).

hash_file(File, Method) ->
    % the file can be huge so read it in chunks
    case file:open(File, [binary,raw,read]) of
        {ok, P} -> hash_loop(P, crypto:hash_init(Method));
        Error   -> Error
    end.

hash_loop(P, C) ->
    case file:read(P, 4096) of
        {ok, Bin} ->
            hash_loop(P, crypto:hash_update(C, Bin));
        eof ->
            file:close(P),
            {ok, crypto:hash_final(C)}
    end.
```

4.2 Applications of hashing

The single most important application of cryptographic hashing is in *validation*. Two data sets can be considered identical if they have the same checksum.

Note: This is not a mathematical certainty. If we have more than 2^{120} *different files then two will have the same SHA1 checksum.*[8]

[8] Since an SHA1 checksum has 120 bits.

4.3 Padding

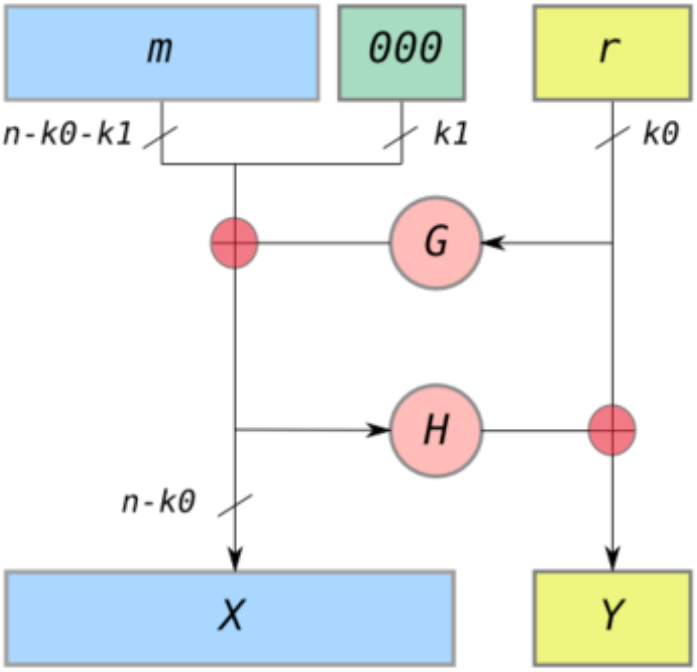
Suppose we have a *fixed length buffer*, containing salt and encrypted data. Something like this:

```
<---------------- fixed length ----------------->
+-------+---------------+----------------------+
| Salt  | Encrypted text | unused area          |
+-------+---------------+----------------------+
```

There is a problem with the unused area. If it contains some constant (like padding with zeros) we will leak information about the encrypted text, like, for example, the length of the text. A "padding scheme" fills the unused area with random bits. Something like:

```
<---------------- fixed length ----------------->
+-------+---------------+----------------------+
| Salt  | Encrypted text | random bits          |
+-------+---------------+----------------------+
```

The padding scheme I use in this tutorial is called OAEP Padding An explanation and the following diagram can be found at https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding



The Erlang implementation is straightforward and follows the diagram:

oaep_byte_padding.erl

```
pad(Msg, N, K0) when is_binary(Msg), is_integer(N),
                     is_integer(K0) ->
    NPad = N - K0 - size(Msg),
    Pad  = make_padding(NPad),
    R    = crypto:strong_rand_bytes(K0),
    G    = g(N-K0, R),
    M0   = <<Msg/binary, Pad/binary>>,
    X    = xor1(G, M0),
    H    = h(X, K0),
    Y    = xor1(R, H),
    <<X/binary, Y/binary>>.
```

5 Public Key Systems

In a public key system two different keys are used. One key is used to encrypt the data and a different key is used to decrypt the data. Use of different keys is called *Asymmetric Encryption*.

In this tutorial I'll take a detailed look at The RSA[9] algorithm. RSA makes use of two keys `{E,N}` and `{D,N}`.

5.1 RSA in a nutshell

Here's a simple test that illustrates how to use RSA:

ez_crypt_test.erl

```
rsa_test() ->
    {E,D,N} = ez_crypt:make_rsa_key(800),
    Secret = 1234567890,
    Crypt  = ez_crypt:mod_pow(Secret,E,N),
    Secret = ez_crypt:mod_pow(Crypt,D,N),
    %% swap E and D
    Crypt1 = ez_crypt:mod_pow(Secret,D,N),
    Secret = ez_crypt:mod_pow(Crypt1,E,N),
    %% It's determanistic
    Crypt2 = ez_crypt:mod_pow(Secret,E,N),
    true = (Crypt2 == Crypt),
    yes.
```

`{E,D,N}` is a triplet of three integers and `mod_pow(X, P, N)` computes $X^P mod\ N$:

[9] Named after Don Rivest, Adi Shamir and Leonard Adleman.

ez_crypt_math.erl

```
mod_pow(A, 1, M) ->
    A rem M;
mod_pow(A, 2, M) ->
    A*A rem M;
mod_pow(A, B, M) ->
    B1 = B div 2,
    B2 = B - B1,
    %% B2 = B1 or B1+1
    P = mod_pow(A, B1, M),
    case B2 of
        B1 -> (P*P) rem M;
        _  -> (P*P*A) rem M
    end.
```

Note1: the algorithm is defined over *integers* not strings or binaries.

Note2: either `E` or `D` can be used for encryption, provided we use the other value for decryption. So we can use `D` to encrypt and `E` to decrypt:

This works because $(X^E)^D mod\ N$ is the same as $(X^D)^E mod\ N$ The $mod\ N$ bit is irrelevant, and obviously $(X^E)^D = (X^D)^E = X^{D*E}$

Using a `N` bit key we can encrypt any integer whose binary representation is less than or equal to `N` bits.[10] In practice I usually encrypt something like an SHA1 checksum (160 bits) with a 800 bit key. 800 bits is fast enough for me and sufficiently difficult to crack that for most purposes can be considered secure.

Creating a key pair is very easy:

ez_crypt_math.erl

```
make_rsa_keypair(K) ->
    %% K is the bitsize of the modulo N
    K1 = K div 2,
```

[10]Note: It is not a good idea to encrypt either very small values or values whose size approaches the bit size of the key. This is since we need to have enough free space in the key for some "salt" and some "padding".

```
    E = 65537,
    P = gen_prime(K1, E),
    Q = gen_prime(K-K1, E),
    N = P*Q,
    Phi = (P-1)*(Q-1),
    D = inv(E, Phi),
    {E,D,N}.
```

`gen_prime(K, N)` makes a prime of `K` bits that is relatively prime to `N`.

`inv(A, B)` computes `C` if it exists such that `A*C mod B = 1` (ie $A^{-1} mod\ B$).[11]

Note: If we know that `P` and `Q` are prime we can can compute `N = P * Q` but given `N` we cannot easily recover `P` and `Q`.

For example:

```
1> P = 3760483207475282540887.
3760483207475282540887
2> Q = 3760483207475282540887.
3760483207475282540887
3> N = P*Q.
14141233953703588876397602262890002826746769
4> ez_crypt:is_probably_prime(N).
false
5> ez_crypt:is_probably_prime(P).
true.
```

5.2 Text-book RSA

RSA encrypts and decrypts integers but not strings. To encrypt a string we first convert it to a integer.[12]

The functions `str2int` and the inverse `int2str` convert between strings and integers.

[11] This is called the modular inverse, and is computed using the extended Euclidean algorithm.

[12] A string can be considered a base 256 integer.

ez_crypt_math.erl

```
str2int(Str) -> str2int([$z|Str], 0).

str2int([H|T], N) -> str2int(T, N*256+H);
str2int([], N)    -> N.

int2str(N) ->
    "z" ++ S = int2str(N, []),
    S.

int2str(N, L) when N =< 0 -> L;
int2str(N, L) ->
    N1 = N div 256,
    H  = N - N1 * 256,
    int2str(N1, [H|L]).
```

Note: I have appended a `z` character so that leading zeros in the string get correctly converted.

Now we can defined the simplest version of RSA encode:

ez_crypt_rsa.erl

```
encrypt_1({E,N}, Bin) ->
    Str = binary_to_list(Bin),
    I   = ez_crypt_math:str2int(Str),
    if I =< N ->
            ez_crypt_math:mod_pow(I, E, N);
       true ->
            io:format("** size error in ez_crypt_rsa:encrypt_1 ~p ~p~n",
                      [I,N]),
            exit({encrypt,size,I,N})
    end.
```

And the inverse:

ez_crypt_rsa.erl

```
decrypt_1({D,N}, I) ->
    J = ez_crypt_math:mod_pow(I,D,N),
    list_to_binary(ez_crypt_math:int2str(J)).
```

We convert the binary `Bin` to an integer `I` then compute `I^E mod N`. `I` has to be less than `N`. The number of bits in `I` is approximately `8*(size(Bin) + 1)` which means the maximum size of `Bin` is about 127 bytes. Provided we use this algorithm for small binaries we won't have any problems.[13]

5.3 RSA for small data with padding

Our second algorithm pads the binary with random numbers using OEAP padding which we explained earlier:

ez_crypt_rsa.erl

```
encrypt_2(Key, Bin) when size(Bin) =< 120 ->
    Bin1 = oaep_byte_padding:pad(Bin, 120, 20),
    encrypt_1(Key, Bin1).
```

The padding extends the size of the data to be encrypted (which is a good thing) and adds salt (which is also good) – double goodness!

The `120` and `20` specify the size of the buffers in the OAEP algorithm. `120` is the total buffer size in bytes. When converted to an integer this must be less than the modulus used in the RSA algorithm.[14]

And the inverse:

ez_crypt_rsa.erl

[13] We'll exit if we can't encode the data.

[14] Note that there is a slight mismatch here. RSA is conventionally described in terms of a fix bit size modulus - this fits nicely with languages like C, but is a conceptual mismatch with Erlang which happily uses bignums. Given the size of a binary we don't know the exact size in bits of the integer returned by str2int. This could be fixed - but us an irrelevant detail as far as this tutorial is concerned.

```
decrypt_2(Key, Bin) ->
    Bin1 = decrypt_1(Key, Bin),
    oaep_byte_padding:unpad(Bin1, 20).
```

As you can see all this does is add a small wrapper round "text book RSA."[15]

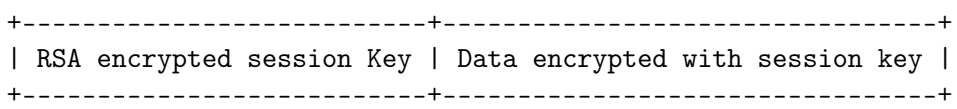
5.4 RSA with large data volumes

RSA is *slow* and *can only encrypt a small amount of data* (ie some value less than the modulus `N` in the key). This is not a problem since we typically use it to encrypt an SHA1 checksum (120 bits) or a short password.

To speed up modulo arithmetic we might use "Montgomery reduction" (ie computations module N are time consuming, so we do this modulo 2^K which is easier, then do some transformation to compute modulo N).

2048 bit modulus are considered secure.[16]

RSA should only be used to encrypt small integers (ie less than the modulus) If we want to encode a large value, we use two steps. First we generate a session key and use a fast symmetric encryption algorithm such as AES256 to encrypt the data, then we encrypt the session key with RSA. So we transmit:



I've chosen random 160 bit session keys (The same bit length as SHA1), with this design choice the code is very simple:

ez_crypt_rsa.erl

```
encrypt_3(Key, Bin) ->
    %% make a session key
```

[15] We saw this phenomena earlier, crypto software gets lay-on-layer of abstractions, so we have to keep a clear head when writing it.

[16] The world record is RSA-768 (2000 years on single code 2.2GHz AMD Opteron.

```
    Session = ez_crypt:random_session_key(),
    %% encrypt the data with the session key
    EncData = ez_crypt_aes:encrypt(Session, Bin),
    %% encrypt the session key
    EncSession = encrypt_2(Key, Session),
    %% add a wrapper
    Bundle = term_to_binary({EncSession, EncData}),
    Bundle.
```

Calling `encrypt_2` make the code *very* simple since `encrypt_2` adds padding (and indirectly salting).

and decrypting is easy:

ez_crypt_rsa.erl

```
decrypt_3(Key, Bundle) ->
    %% remove the wrapper
    {EncSession, EncData} = binary_to_term(Bundle),
    %% decrypt the session key
    Session = decrypt_2(Key, EncSession),
    %% decrypt the data
    Data    = ez_crypt_aes:decrypt(Session, EncData),
    Data.
```

Again note how this code just uses a small wrapper round `encrypt_2` and `decrypt_2`. Also observe how the encryption and decryption code mirror each other. `term_to_binary` and `binary_to_term` are used to pack and unpack the data avoiding the use of complex envelopes (like ASN.1).[17]

This is the most robust version of the RSA encryption routines so I've aliased these from `ez_crypt.erl`:

ez_crypt.erl

```
rsa_encrypt({E,N}=Key, Bin) when is_integer(E),
                                 is_integer(N),
```

[17] Isn't this nice. Pity all crypto code isn't this easy to understand.

```
                                    is_binary(Bin) ->
    ez_crypt_rsa:encrypt_3(Key, Bin).
```

and

ez_crypt.erl

```
rsa_decrypt({E,N}=Key, B) when is_integer(E),
                               is_integer(N),
                               is_binary(B) ->
    ez_crypt_rsa:decrypt_3(Key, B).
```

Now we're done with RSA. But what about the keys? How should we manage these?

5.5 Storing keys in files

The next problem we'll look at is storing and distributing keys. We can create a key pair with

```
1> ez_crypt:make_rsa_key(128).
{65537,91830260069961236696707256003409304625,
 271081471744743460660673776656782478911}
```

This makes a key pair. But what we want to do is create two files from this. A plain text file with the public key which anybody can read and an encrypted file with the private key that is password protected:

ez_crypt.erl

```
make_rsa_keyfiles(Name, Email, Len, Password) ->
    {E,D,N} = make_rsa_key(Len),
    {ok, S} = file:open(Name ++ ".pub", [write]),
    io:format(S,"~p.~n",
              [#{type=>public_key,
                 email => Email,
                 e => E,
```

```
                n => N}]),
    file:close(S),
    Term = #{type => private_key,
             name => Name,
             email => Email,
             e => E,
             d => D,
             n => N},
    Bin1 = term_to_binary(Term),
    Bin2 = aes_encrypt(Password, Bin1),
    {ok, S1} = file:open(Name ++ ".pri", [write]),
    io:format(S1,"~p.~n",
              [#{type=>encrypted_private_key,
                 value => Bin2}]),
    file:close(S1).
```

We can run this:

```
> ez_crypt:make_rsa_keyfiles("joe", "erlang@gmail.com",
                             1024, <<"verysecret">>).
ok
```

This creates two files `joe.pub` which contains something like this:

```
#{e => 65537,
  email => "erlang@gmail.com",
  n => 10806934495662030846296771 49 ... 751495357,
  type => public_key}.
```

We can recover the key with:

ez_crypt.erl

```
read_public_key(File) ->
    {ok, [#{type := public_key,
```

```
            e := E, n := N}]} = file:consult(File),
    {E, N}.
```

What do we do with this file? We can either distribute this file together with our application, or we can cut-and paste the contents into some Erlang code which returns the public key. This way the key will be loaded without touching the file system.

The public key contains something like this:

```
#{type => encrypted_public_key,
  value => <<81,15,195,174,78,46,109,191,197,49,53,174,17,
             ... 103,199,16,138,86,27,184,52>>}
```

The value has been encrypted with the password we supplied when we created the key. We can read the key with the following:

ez_crypt.erl

```
read_private_key(File, Password) ->
    {ok, [#{type := encrypted_private_key,
            value := Bin}]} = file:consult(File),
    Bin1 = aes_decrypt(Password, Bin),
    Term = binary_to_term(Bin1),
    #{d := D, n:= N} = Term,
    {D, N}.
```

5.6 RSA Open SSL key pairs

RSA is essentially a pair of keys `{E,N}` and `{D,N}` and some modular arithmetic $M^E mod\ N$ so how come Open SSL is so complex?

It turns out to be rather simple if we dig a little.

We start by generating an RSA key pair:

```
ssh-keygen -t rsa -b 1024 -C "joe@somewehere.com"
```

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/joearmstrong/.ssh/id_rsa): joe_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in joe_rsa.
Your public key has been saved in joe_rsa.pub.
```

`joe_rsa` contains the `{E,D,N}` tuple that is the source of all goodness, and some other stuff that is less exciting. We can pull out this data as follows:

decode_rsa_keys.erl

```
test() ->
    {E,D,N} = Key = read_pri_certificate(),
    {E,N}   = read_pub(),
    io:format("Key:~p~n",[Key]),
    false  = ez_crypt_primes:is_prime(N),
    check(12345, {E,D,N}),
    check(54321, {D,E,N}),
    1024 = ez_crypt_math:bsize(N),  %% modulus size
    wow.

check(Msg, {E,D,N}) ->
    Crypt  = ez_crypt_math:mod_pow(Msg, E, N),
    Decode = ez_crypt_math:mod_pow(Crypt, D, N),
    Msg = Decode,
    ok.

read_pri_certificate() ->
    {ok, B} = file:read_file("joe_rsa"),
    [DSA] = public_key:pem_decode(B),
    X = public_key:pem_entry_decode(DSA),
    #'RSAPrivateKey'{modulus = N,
                     publicExponent = E,
                     privateExponent=D} = X,
    {E,D,N}.

read_pub() ->
    {ok, B} = file:read_file("joe_rsa.pub"),
    [{Pub,_}] = public_key:ssh_decode(B,public_key),
```

```
    #'RSAPublicKey'{modulus = N, publicExponent = E} = Pub,
    {E, N}.
```

And have some fun!

```
> decode_rsa_keys:test().
Key:{65537,
    8465878345925402733279....971822495488001,
    1071945495772 ... 1194367526413419199174309}
wow
```

So it was easy after all.

6 Secret sharing

https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing algorithm.

Shami'r secret sharing algorithm is a `K`, `N` algorithm. The key is split into `N` fragments. Any `K` of them can used to reconstruct the key.

The implementation here is due to Robert Newson and was published at https://github.com/rnewson/shamir/.

As an example, suppose we want to share the secret `hello` using seven shares, so that any three shares unlock the secret. We can generate the shares like this:

```
> L=shamir:share(<<"hello">>, 3,7).
[{share,3,1,<<206,145,84,97,217>>},
 {share,3,2,<<229,208,230,155,102>>},
```

```
{share,3,3,<<67,36,222,150,208>>},
{share,3,4,<<56,15,170,12,128>>},
{share,3,5,<<158,251,146,1,54>>},
{share,3,6,<<181,186,32,251,137>>},
{share,3,7,<<19,78,24,246,63>>}]
```

Using shares 1 2 and 5 we can reconstruct the secret as follows:

```
> shamir:recover([lists:nth(1,L),lists:nth(2,L),lists:nth(5,L)]).
<<"hello">>
```

The algorithm fails if we don't give it three different shares:

```
> shamir:recover([lists:nth(1,L),lists:nth(2,L),lists:nth(2,L)]).
** exception error: no function clause matching
  shamir:recover(3,[{1,10},{2,199}]) (shamir.erl, line 50)
  in function  shamir:'-recover/1-lc$^2/1-1-'/2 (shamir.erl, line 48)
  in call from shamir:recover/1 (shamir.erl, line 48)
```

Again the shared secret should be a password that unlocks or validates the content of another file.

7 Applications

7.1 Application 1: Challenge-Response

The Challeng-Response algorithm ensures that no plain text passwords are sent over the wire.

Here's an example. Assume that `joe` has password `"bingo"`. The interaction between a client and server is as follows:

```
          {login,"joe"}
Client ---------->---------- Server

          {challenge,"zq12i3"}
Client ----------<----------  Server

         {response,md5("bingo"++"zq12i3")}
Client ------------>---------------------- Server

               login_ok | login_error
Client --------------<---------------------- Server
```

In response to a request `{login,"joe"}` the server generates a random string and sends it to the client. The client responds by computing the MD5 checksum of the random string and the shared secret. The server can check the responds using the shared secret and authenticate the user.

challenge.erl

```
client(Server, Who, Parent) ->
    Server ! {self(), {login,Who}},
    receive
        {Server, {challenge, X}} ->
            Server ! {self(), {response,
                               erlang:md5(X ++ secret(Who))}},
            receive
                {Server, Response} ->
                    Parent ! Response
            end
    end.
```

And the server is like this:

challenge.erl

```
server() ->
    receive
        {Pid, {login,Who}} ->
            Ran = binary_to_list(crypto:strong_rand_bytes(10)),
            Pid ! {self(), {challenge, Ran}},
            receive
                {Pid, {response, R}} ->
                    case erlang:md5(Ran ++ secret(Who)) of
                        R ->
                            Pid ! {self(), login_ok};
                        _ ->
                            Pid ! {self(), login_error}
                    end
            end
    end.
```

This is called "one-way authentication" - the server has validated the identity of the client, but not the other way around. The server has not proved to the client that it **is** the server. In "two way authentication" the algorithm is run twice, once in each direction. In the first pass the server authenticates the client, in the second pass the roles of the client and server are reversed and the client authenticates the server.

The main problem with this is that the server needs to store plain text passwords - better methods exist.

7.2 Application 2: Authentication Algorithms

We'll first talk about how to authenticate a single file.

The easiest way to authenticate something is to generate a checksum of the file and sign the checksum with your private key. I'll assume the public and private keys are stored in files:

ez_crypt.erl

```
sign_file(File, PrivateKeyFile, Password) ->
    SHA = sha1_checksum_of_file(File),
    PriKey = read_private_key(PrivateKeyFile, Password),
```

```
    Sig = rsa_encrypt(PriKey, SHA),
    file:write_file(File ++ ".sig", term_to_binary(Sig)).
```

To validate the file, a user needs the file `File` and the signature `Sig` and the public key of the signer. The file is authenticated with:

ez_crypt.erl

```
validate_file(PubKeyFile,  File) ->
    SHA1 = sha1_checksum_of_file(File),
    {ok, Bin} = file:read_file(File ++ ".sig"),
    Sig = binary_to_term(Bin),
    PubKey = read_public_key(PubKeyFile),
    io:format("PubKey=~p~n",[PubKey]),
    SHA2 = rsa_decrypt(PubKey, Sig),
    Ret = case SHA1 of
              SHA2 -> true;
              _    -> false
          end,
    io:format("Validate:~s ~p~n",[File, Ret]),
    Ret.
```

We can validate several files by storing the filenames and their checksums catalog and, then signing the catalog. For example the cataloger could be a list of Erlang terms:

```
    {file,"this.erl", "a23121tsu128368136"}.
    {file,"that.erl", "1293879127391732"}.
```

Making this is easy:

ez_crypt.erl

```
sign_current_dir() ->
    F = erl_files(),
    L = [{file,I,sha1_checksum_of_file(I)} || I <- F],
    io:format("L=~p~n",[L]),
```

```
    write_term("catalog", L),
    sign_file("catalog", "joe.pri", <<"verysecret">>).
```

To validate this we first validate the catalog, and if it is correct we know the SHA1 checksums of the individual files. Then we check the SHA1's of each of the files.

7.3 Application 3: Secret message streams

Secret message streams are streams of messages sent to a server where only the sever can decode the messages. The clients are supplied with a pre-compiled version of the server public key. This has the advantage that no password management in the client is necessary.

Each message is encrypted with a new random key. The key is encoded with the public key of the server.

ez_crypt.erl

```
encrypt_message(Who, Msg) ->
    Ran = random_session_key(),
    EncMessage = aes_encrypt(Ran, Msg),
    PubKey = read_public_key(Who),
    EncKey = rsa_encrypt(PubKey, Ran),
    term_to_binary({EncKey,EncMessage}).
```

Decoding the message is easy:

ez_crypt.erl

```
decrypt_message(Who, Password, Bin) ->
    {EncKey,EncMessage} = binary_to_term(Bin),
    PriKey = read_private_key(Who, Password),
    Key = rsa_decrypt(PriKey, EncKey),
    Message = aes_decrypt(Key, EncMessage),
    Message.
```

We can do a quick test to show that this works

```
1> C=ez_crypt:encrypt_message("joe.pub",<<"hello">>).
<<131,104,2,109,0,0,0,187,131,104,2,110,128,0,170,153,221,
  253,81,86,2,138,59,10,204,163,156,185,191,...>>
2> ez_crypt:decrypt_message("joe.pri",<<"verysecret">>,C).
<<"hello">>
```

7.4 Application 4: TLS (Transport Layer Security)

TLS, very much simplified works like this:

After the key exchange is over both sides know `S1` and `S2`. `S1` is used to encrypt `client->server` messages and `S2` for `server->client` messages.

These are the basic ideas involved. The actual protocol is far more messy than this simple diagram might imply. In the real TSL there is a phase of protocol negotiation, and packet envelopes and encoding/decoding of data has to be agreed upon. A pure Erlang implementation of a subset of the protocol is very easy to implement and understand.

We use RSA to encode and decode the session keys, we don't negotiate the protocols and we use `term_to_binary` to encode the massages.

Here's the entire thing in a few lines of Erlang:

tls.erl

```
client(Server) ->
    io:format("Client requesting key~n"),
    Server ! {self(), hello},
    receive
        {Server, ServerPub} ->
            S1       = random_session_key(),
            {E,D,N}  = make_rsa_key(1024),
            TmpPri   = {D,N},
            TmpPub   = {E,N},
            Bin1     = term_to_binary({S1,TmpPub}),
            Msg1     = {key1, rsa_encrypt(ServerPub, Bin1)},
            io:format("Client sending   S1:~p~n",[S1]),
            Server ! {self(),Msg1},
            receive
                {Server, {key2, Int}} ->
                    S2 = rsa_decrypt(TmpPri, Int),
                    io:format("Client recovered S2:~p~n",[S2]),
                    client_loop(S1, S2)
            end
    end.
```

And the server:

tls.erl

```
server() ->
  receive
    {Client, hello} ->
      io:format("Server sending public key~n"),
      Client ! {self(), server_public_key()},
      receive
        {Client, {key1,Int}} ->
          Bin1 = rsa_decrypt(server_private_key(), Int),
          {S1, TmpPub} = binary_to_term(Bin1),
          io:format("Server recovered S1:~p~n",[S1]),
          S2 = random_session_key(),
          io:format("Server sending   S2:~p~n",[S2]),
          Client ! {self(), {key2,rsa_encrypt(TmpPub, S2)}},
          server_loop(S1, S2)
      end
  end.
```

And we can run it like this:[18].

```
4> tls:test().
Client requesting key
Server sending public key
<0.49.0>
Client sending   S1:<<106,76,214,102,3,172,229,70,86,129,223,
                      156,134,223,14,104,6,88,7,242>>
Server recovered S1:<<106,76,214,102,3,172,229,70,86,129,223,
                      156,134,223,14,104,6,88,7,242>>
Server sending   S2:<<192,132,4,3,69,169,132,252,71,60,111,200,
                      29,166,75,59,170,181,250,129>>
Client recovered S2:<<192,132,4,3,69,169,132,252,71,60,111,200,
                      29,166,75,59,170,181,250,129>>
```

7.5 Application 5: SFS Self-certifying File System

The Self-certifying File System (SFS) is a distributed file system using a protocol described in the David Mazière's PhD Thesis http://pdos.csail.mit.

[18] Take a look in tls.erl for more details

edu/~ericp/doc/sfs-thesis.ps.

The key idea in this thesis is to publish the SHA1 checksum of the public key of a server rather than the public key itself. It's conceptually similar TLS but with a simple twist. The client does not initially know the public key of the server. Instead it knows the SHA1 checksum of the public key of the server.

It's called "Self certifying" since the server provides a public key which is not signed by a certification authority.

The client can then request the public key from anywhere that claims to know what the public key of the server is. Once it has obtained a response to the public key request it can check the key using the SHA1 checksum to make sure that the key is correct. Anybody can provide the key and it can be cached by the client.

Only the server can decode messages encoded with the public key.

The advantage of this is that we only need to distribute the SHA1 checksum of the public key of the server and NOT the public key itself. This is splendid since the checksum is only 20 bytes and can be easily communicated by any out of band method.[19] This is not true of the public key itself which is several KBytes long.

The client code to request the key is:

sfs.erl

```
client(Server, SHA) ->
    Server ! {self(), public_key_request},
    receive
        {Server, {public_key_response, PubKey}} ->
            case sha1(PubKey) of
                SHA ->
                    connect(Server, PubKey);
                _  ->
                    exit(badKey)
            end
    end.
```

[19] It's short enough so you can write it down on a sheet of paper.

and the corresponding server code:

sfs.erl

```
server() ->
    receive
        {Client, public_key_request} ->
            Client ! {self(),
                      {public_key_response,
                       public_key()}},
            wait_connect(Client)
    end.
```

To flesh this out into a functioning server we need to combine the TSL code with this code and add some key manipulation.[20]

8 Experiments

8.1 Experiment 1 - Make some random integers

Making random integers is a difficult problem. We can either trust some library to provide good random numbers, or use a combination of a software random number generator together with a physical source of randomness. We could for example, get the user to type in keystrokes and take the low-order bits in the time intervals between keystrokes, or take a digital photo and take the low order bits of the image, then destroy the image. No method of generating random integers is foolproof and indeed systems have made less secure by hacking into the part of the system that creates random numbers.

In this tutorial I'll assume that the random number generator provided in the `crypto` application is sound. As an exercise you can combine this with (say)

[20] This is left as an exercise – if you've been paying attention this should be easy!

a keystroke timing algorithm to make a better algorithm. If you do this you can check the result into the project archive and send me a push request and I'll take a look at it.

`ez_crypto_math:random_integer(Len)` calls the crypto random number generator to generate a random binary of `Len` bytes and converts it to an integer. For example:

```
1> ez_crypt_math:random_integer(20).
162882318606168104519781637228123393036335515557
```

8.2 Experiment 2 - Generating a random integer that can be represented in a specific number of bits

The next thing we might want to do is create a random integer of *exactly* `K` bits. This is done with `ez_crypt_math:k_bit_random_integer`. For example:

```
1> I = ez_crypt_math:k_bit_random_integer(40).
792296059411
2> ez_crypt_math:bsize(I).
40
3> ez_crypt_math:bsize(ez_crypt_math:k_bit_random_integer(100)).
100
4> ez_crypt_math:bsize(ez_crypt_math:k_bit_random_integer(2048)).
2048
```

`bsize(N)` returns the number of bits necessary to represent the integer `N`.

We use `ez_crypt_math:k_bit_random_integer` to generate RSA keys integers.

8.3 Experiment 3 - Test an integer to see if it's a prime

Prime number testing is tricky.

First we test the number to see if it is a multiple of a small prime. The function `small_primes()` returns a hard-wired list of the first 2000 primes. If

the number being tested is in this list then we can say it definitely is a prime or if it's a multiple of a number in the list then we know its not a prime so we can return `true` or `false`. If it's not a multiple of a small prime we perform the Miller-Rabin test and call `ez_crypt_miller_rabin:is_probably_prime`

Miller-Rabin is an expensive test, which is why we tested against a list of known primes before performing the test.

ez_crypt_primes.erl

```
is_prime(0) -> false;
is_prime(1) -> false;
is_prime(X) ->
    case is_multiple_of_small_prime(X) of
        {yes, X}  -> true;
        {yes,_}   -> false;
        no        -> ez_crypt_miller_rabin:is_probably_prime(X)
    end.

is_multiple_of_small_prime(N) ->
    is_multiple_of_small_prime(N, small_primes()).

is_multiple_of_small_prime(N, [H|T]) ->
    case N rem H of
        0 -> {yes, H};
        _ -> is_multiple_of_small_prime(N, T)
    end;
is_multiple_of_small_prime(_, []) ->
    no.
```

Note that `is_prime(N)` willfully lies and returns `true` or `false` and not (`true`, `false` or `probably`).[21]

[21] Some things we'll never be sure about!

8.4 Experiment 4 - Make some random integers with a fixed bit length

Many algorithms want to have primes with a fixed bit length. For example primes whose binary representation fits into to exactly `K` bits. Why is this? - the main reason is (I think) to make analysis of the algorithms simpler and to know in advance how much space needs to be allocated in fixed length data structures.

Making a random prime with an exact number of bits is a tad more tricky than making a random prime.

I start by making a random number of exactly `K` bits. This is done by generating a random integer known to have more than `K` bits, then computing the length and chopping of one bit at a time until the integer has the required number of bits.

The first odd integer `P` of `K` bits is used to initialize a generate and test algorithm. Every time the test on a prime `P` fails we test `P+2` until we hit a prime. Hopefully `P+2` will have `K` bits if `P` has `K` bits (why is this?) - for large `P` this is true, but for small `P` it is false, so for small `P` I use an entirely different algorithm.

To be absolutely sure the generated prime has `K` bits I do a final test before returning and if the generated prime does not have exactly `K` bits I start over and do everything again. This is highly unlikely, but it could happen and I do

want some guarantees here. The return value *must have exactly K bits.*[22]

9 ez_crypt.erl

`ez_crypt.erl` is "work in progress" - it's not finished and does not contain production quality code. If you want to base code on it then fine - do so, but don't blame me if it has bugs.

My intention here is to teach cryptographic techniques using small understandable code fragments, it's not to produce production quality code.

10 Miscellaneous

10.1 A little math

I'm not going to tell you what a prime number is, if you don't know you're probably reading the wrong tutorial.

A lot of crypto algorithms involve computing A^B and $A^B \ mod \ C$.

Why is this?

Let's meet Alice and Bob.[23] Alice and Bob both know two numbers x and y.

[22] I can't prove this mathematically, but I can empirically test it in code, so I'm a happy bunny.

[23] Cryptographers always talk about Alice, Bob, Carol, Dan and on. Can you guess why?

Alice chooses some random number R and computes a message $M = R^x$. Alice sends M to Bob. Bob receives M and computes $S = M^y$. Alice can also compute $S = M^y$. S is now a "shared secret" - both Alice and Bob know S anybody watching the communication sees only M so they cannot figure out the value of S without knowing both x and y.

In its various forms the fact that $(K^x)^y = (K^y)^x$ pops up in many algorithms and is the basic reason why RSA and Diffie-Hellman work.

In practice we'll compute $A^B mod\ N$ since values of A^B can be extremely large - taking the exponentiation modulo N bounds all the values to a maximum value or N which makes things easier to work with.

10.2 A couple of theorems

The correctness of RSA depends upon two theorems:

Fermat's little theorem: $a^p \equiv\ a\ mod\ p$ if p is a prime number.

Eulers theorem: $a^{\phi(N)}\ \equiv 1(mod\ n)$ if a and n are coprime. $\phi(N)$ is Eulers totient function.[24]

10.3 Prime number testing

Prime number testing makes use of Fermats little theorem.

Recall that $a^p \equiv\ a\ mod\ p$ if p is a prime number.

So to test if p is prime we choose several different values of a and apply the Fermat test. Unfortunately if p is composite for certain values of a the Fermat equivalence is obeyed. And for some particularly nasty values of p[25] the equivalence is obeyed for all values of a.

Tests where the Fermat equivalence is obeyed but when p is composite are called "false witnesses."

The Miller Rabin test is a probabilistic variant of the Fermat test. Each iteration of the test decreases the probability of error this is why we can say that a large number p is "probably prime" not that it is "definitely a prime."

[24] the number of integers from 1 to N that are coprime to N.

[25] The Carmichael numbers

We can however say that a large number is composite without being able to compute the factors.

This is one of the frustrating things about RSA - we can easily prove that the modules is composite - but we can't compute the factors. This is a good thing ^{TM}- if it were false the worlds financial system would break down.

10.4 Why RSA works

By construction

$$ed \equiv 1 \ mod \ \phi(N)$$

So there exists some k such that:

$$ed \equiv 1 + k\phi(N)$$

Suppose:

$$c = m^e \ mod \ N$$

Then:

$c^d = m^{ed} \ mod \ N$
$= m^{1+k\phi(N)} \ mod \ N$
$= m^1.(m^{\phi(N)} \ mod \ N)^k$

But $m^{\phi(N)} mod N = 1$ is Euler's theorm[26], thus:

$c^d = m^1.(1)^k$
$= m$

Also N is the product of two primes P and Q then $\phi(N) = (P-1) * (Q-1)$

Note: this proof is not quite correct, since we must also show that $gcd(m, N) = 1$ which requires a longer explanation ...

[26] also known as the Fermat-Euler theorem of Eulers's totient theorem

10.5 Further reading

- Montague arithmetic
- Galoir Fields GF(256)

10.6 How big RSA keys?

How large should an RSA key be? The WikiPedia says the world record was set for RSA-768[27] and has stood since 2009. This took the equivalent of 2000 years on a single-core 2.2 GHZ AMD Operon.

Adding additional bits to the modulus increases the complexity of the problem. The difficulty of attacking RSA reduces to the difficulty of factoring the RSA modulus which is known to be a hard problem.

2048 bits is recommended if you're expecting aliens.[28] For my purposes I reckon a hundred bits more than the world record is good enough for most purposes.[29]

At high levels of security guarding against "side channel attacks" and software bugs is far more important than agonizing over bit lengths.

10.7 How big Hashes, which hash should I choose?

I've used SHA1 in this tutorial. And *Yes I know it's not recommended.* SHA256 is now recommended.

10.8 What is a good symmetric encryption algorithm?

I've used AES256 – I think it's ok. I'm not actually writing "super duper secure systems" so I just want to stop *script kiddies* from messing with my stuff - not professional attackers.

[27] The modulus is 768 bits.
[28] I've seen Mars Attacks, but since we won, I'm not particularly worried about them.
[29] I'd check this every year or so, to see what's happening here.

10.9 Sidechannel attacks

Most crypto systems are broken not because somebody managed to break the crypto system but because some property of the system was exploited that the creator of the system had not thought of.

A crypto system is as strong as its weakest link, so although the math in the crypto algorithms might be sound side channel attacks are possible.

Here are some example of side channels:

- Bribing a sysadmin who knows the system passwords
- Spying on memory while a crypto program is running
- Planting a password sniffer in the firmware of the keyboard
- Measuring the timing of internal signals when a password is entered
- Trying all the small strings on your hard disk to see if they are passwords
- Analyzing the swap area of your disk
- Torture

As an example of side channels you might like to consult the validation procedure SET transactions (ref).

True story: A few years ago I was involved in an pre-study for an "electronic wallet" project. We wanted to put a crypto-chip into our phones and partner with a major bank to make a secure payment system.

The bank said - "we won't trust your chip. We can make the chip which you put in your phones" - we said "You're not going to put your chip in our phones,

we don't trust your chip, it might mess up our phones." So the project didn't happen.

11 lin.erl

This crypto tutorial started off many years ago with a module called `lin.erl` it has a sub-set of the routines in `ez_crypt_math.erl` and is somewhat easier to understand so I've included it here.

Many cryptography algorithms need some bignum integer functions. In particular we need some support for the `inv` function which is used to compute the RSA exponents.

For example, suppose we want to find X such that $28 * X \equiv 1 mod 75$ The solution is $X = 28^{-1} mod 75$. This we can compute with the `inv/2` function in the Erlang shell:

```
1> lin:inv(28, 75)
67.
```

So 67 is the "modular multiplicative inverse of 28 modulo 75" – we can check this in the Erlang shell:

```
2> 28*67 rem 75.
1
```

Modular arithmetic in the integer domain keeps all values constrained to the integer domain which is why it is nice for cryptography.[30]

[30] Even more fun is the finite field GF(256) used in the AES standard. In this field values

`lin.erl` exports the following functions:

`pow(A, B, M) -> V`
: Computes `V = (A^B) mod M`

`inv(A, B) -> C | no_inverse`
: Computes `C` such that `A*C mod B = 1` If such C exists.

`solve(A, B) => {X, Y} | insoluble`
: Solves the linear congruence `A * X - B * Y = 1` if it is solvable.

`str2int(Str)`
: Converts a string to a base 256 integer.

`int2str(N)`
: Converts a base 256 integer to a string

`gcd(A, B)`
: Computes the greater common denominator of `A` and `B`

Some of these are pretty simple, `pow(A, B, M)` computes `A^B mod M`. To compute this we proceed as follows: if `B` is even we compute `pow(A, B div 2, M)` and square the result (modulo `M`). If `B` is odd and greater than one we compute `P = pow(A, (B-1)div 2, M)` and then `P*P*A mod M`:

lin.erl

```
pow(A, 1, M) ->
    A rem M;
pow(A, 2, M) ->
    A*A rem M;
pow(A, B, M) ->
    B1 = B div 2,
    B2 = B - B1,
    %% B2 = B1 or B1+1
    P = pow(A, B1, M),
    case B2 of
        B1 -> (P*P) rem M;
```

are constrained to the 0..255 domain – but this is way too complex for an introductory tutorial.

```
        _  -> (P*P*A) rem M
    end.
```

gcd is also easy:

lin.erl

```
gcd(A, B) when A < B -> gcd(B, A);
gcd(A, 0) -> A;
gcd(A, B) ->
    gcd(B, A rem B).
```

As are conversions between strings and integers:

lin.erl

```
str2int(Str) -> str2int([$z|Str], 0).

str2int([H|T], N) -> str2int(T, N*256+H);
str2int([], N) -> N.

int2str(N) ->
    "z" ++ S = int2str(N, []),
    S.

int2str(N, L) when N =< 0 -> L;
int2str(N, L) ->
    N1 = N div 256,
    H = N - N1 * 256,
    int2str(N1, [H|L]).
```

`solve/2` requires some thought, before launching into the code we give some examples:

Solve `12x - 5y = 1` for integer `x` and `y`, solution `x = -2` and `y = -5` (check `12.-2 -5.-5 = 1` as required.

Solve `28x - 25y = 1` for integer `x` and `y`, solution `x = -8` and `y = -9` (check `28.-8 - (25.-9) = 1` as required.

These solutions are computed as follows:

```
> lin:solve(12,5).
{-2,-5}
> lin:solve(28,25).
{-8,-9}
```

To see how to solve these congruences we give a simple example

```
solve            12x - 5y = 1          (1)
or               (2*5 + 2)x - 5y = 1
regrouping       2x + 5(2x - y) = 1

let a = 2x - y                         (2)

then             2x + 5a = 1           (3)
or               2x + (2*2 + 1)a = 1
regrouping       2(x + 2a) + a   = 1

let b = x + 2a                        (4)

then          2b + a = 1        (5)

A solution to this is b = 1, a = -1

Then from (4) x = b - 2a = 1 - 2(-1) = 3    (6)
and  from (2) y = 2x - a = 2*3 -(-1) = 7.   (7)

So a solution is (x, y) = (3, 7)

Check 12*3 - 5*7 = 1 as required
```

This gives us the key idea as to how to solve linear congruences.

In order to solve `12x - 5y = 1` (equation 1) we make a substitution (equation 2) to reduce this to a simpler form, then we have to solve the simpler sub problem which is `2x + 5a = 1` (equation 3) . This is a simpler problem because

the magnitude of the arguments are less. Eventually the process terminates when a trivial subproblem (equation 5) is encountered. Having found the solution to the sub-problem we back substitute (equations 6 and 7) to obtain the final result.

Note that some linear congruences are not solvable; `Ax - By = 1` is not soluble if `A mod B = 0`

The above algorithm is easily encoded as:

lin.erl

```
gcd(A, B) when A < B -> gcd(B, A);
gcd(A, 0) -> A;
gcd(A, B) ->
    gcd(B, A rem B).
```

lin.erl

```
solve(A, B) ->
    case catch s(A,B) of
        insoluble -> insoluble;
        {X, Y} ->
            case A * X - B * Y of
                1     -> {X, Y};
                _Other -> error
            end
    end.

s(_, 0)  -> throw(insoluble);
s(_, 1)  -> {0, -1};
s(_, -1) -> {0, 1};
s(A, B)  ->
    K1 = A div B,
    K2 = A - K1*B,
    {Tmp, X} = s(B, -K2),
    {X, K1 * X - Tmp}.
```

Fortunately Erlang has bignums so that:

```
> lin:solve(2812971937912739173,21037891739173971937917391739173).
{-997308564012181922485842000,-1333499116160234237}
```

Finally `inv(A, B)` which computes `C` such that `A*C mod B = 1` if such an inverse exists.

lin.erl

```
inv(A, B) ->
    case solve(A, B) of
        {X, _} ->
            if X < 0 -> X + B;
               true  -> X
            end;
        _ ->
            no_inverse
    end.
```

12 LIBNACL

This is an "easy to use" crypto library.

Here is a quote from the paper of Bernstein, Lange and Schwabe which describes the library:

```
A typical cryptographic library uses several steps to authenticate and encrypt
a packet. Consider, for example, the following typical combination of RSA, AES,
etc.:

- Alice generates a random AES key.
- Alice uses the AES key to encrypt the packet.
- Alice hashes the encrypted packet using SHA-256.
- Alice reads her RSA secret key from ‘‘wire format.’’
- Alice uses her RSA secret key to sign the hash.
- Alice reads Bob’s RSA public key from wire format.
- Alice uses Bob’s public key to encrypt the AES key, hash, and signature.
- Alice converts the encrypted key, hash, and signature to wire format.
- Alice concatenates with the encrypted packet.
```

Quote from: http://cr.yp.to/highspeed/coolnacl-20120725.pdf

You'll notice the similarity between this list and some of the contortions we've been through earlier in this tutorial.

- Erlang binding to NaCl in the form of libsodium https://github.com/jlouis/erlang-nacl.
- NACL http://nacl.cr.yp.to/.

13 Things I have not talked about

13.1 Keyrings - Certificate Chains, Certifying authorities.

Once we've understood the ideas of public key encryption and authentication the ideas of a key-chain or certifying authority is rather easy.

We start with a "root certificate" (*The Key of the world*) and use this to sign sub-certificates. The sub-certificates can in their turn create child certificates. The certificate contain backwards pointer to their parents and a "best before" date.

13.2 Galois Arithmetic

Integer multiplication is problematic. If we multiply two positive integer together where both is in the range `0..255` we sometimes get an integer that is outside this range.

Integer division is even more horrible. If we divide two integers we sometimes get an integer, we sometimes get a float, and sometimes it's impossible.[31]

A *Galois field* is an algebraic structure where the following rules apply:

- Multiplication, division, addition and subtraction of any two elements in the field result in a third value that is within the field.
- Each element in the field has an inverse. So if A is in the field there exists some element B such that $A * B = 1$. Usually we write B as A^{-1}

The Galois Field GF(256) contains the integers `0..255` if we perform any arithmetic operation on integers in the Galois field we get another integer in the field. - Amazingly algorithms for inverting matrices and solving linear equations which were first used in the integer domain also work in the Galois field. In the Galois field things like division of elements within the field stay within the field.[32]

Arithmetic in the Galois field cannot overflow or fail with precision problems and is used widely in various crypto-algorithms.

Note: The math behind RSA is relatively straightforward.[33] The math behind "Elliptic Curve Cryptography" is however not so simple.[34]

[31] When we try to divide by zero.

[32] Unlike the division of integers which can take you outside the integer domain and into the domain of real numbers.

[33] Which is why I like it.

[34] This is an understatement, a PhD in number theory would be a good prerequisite here.

We can either use thigns like RSA where the math is reasonable simple and code uses Erlang bignums and "understand it ourselves" OR we can trust math we do not understand and programs we cannot reasonably be expected to validate.[35]

So this is where I end.

To delve deeper I can recommend the following:

- https://www.schneier.com/books/applied_cryptography/
- https://www.crypto101.io/

[35] And yes, we are programmers, and the code is open source and we can read it, but it is beyond the state of the art to prove that it is correct and even if we could prove that the program corresponds to the math, we probably could not understand the math.