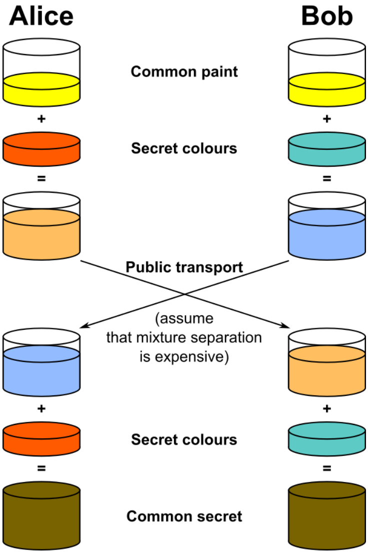
An efficient implementation of Diffie-Hellman key exchange protocol on UDOO

Sachin Rathod and Sahaj Biyani
{rathod,sahajbiyani}@cs.ucsb.edu

Diffie-Hellman Key Exchange

- How can two parties agree on a secret value when all of their messages might be overheard by an eavesdropper?
- The Diffie-Hellman [1] key agreement protocol (1976) was the first practical method for establishing a shared secret over an unsecured communication channel.
- The point is to agree on a key that two parties can use for a symmetric encryption, in such a way that an eavesdropper cannot obtain the key.
- The Diffie-Hellman algorithm accomplishes this, and is still widely used.

Diffie-Hellman Algorithm Analogy



Diffie-Hellman Algorithm

Steps in the Algorithm:

1. Alice and Bob agree on a prime number p and a base g.
2. Alice chooses a secret number a, and sends Bob $(g^a \bmod p)$
3. Bob chooses a secret number b, and sends Alice $(g^b \bmod p)$
4. Alice computes $((g^b \bmod p)^a \bmod p)$
5. Bob computes $((g^a \bmod p)^b \bmod p)$

Implementation Methods

We tried different exponentiation methods to compute the key values to compare their performance on different platforms.

Three methods of exponentiation:

1. Binary Exponentiation (Implemented)
2. Montgomery Exponentiation (Implemented)
3. OpenSSL (Used from library)

Implementation

- For managing arbitrary length numbers, we used OpenSSL's BIGNUM structure [2] and its library functions.
- This library performs operations on integers of arbitrary size. The operations include arithmetic (add, multiply etc.), comparison, conversion to different formats etc.

Binary Exponentiation Method

One of the methods we used for analysis is binary exponentiation. The binary exponentiation method is explained by the following algorithm:

Input: $M, e, n.$

Output: $C = M^e \ mod \ n.$

Step 1. $if \ e_{k-1} = 1 \ then \ C = M \ else \ C = 1$

Step 2. $if \ i = k-2 \ downto \ 0$

2a. $C = C.C \ (mod \ n)$

2b. $if \ e_i = 1 \ then \ C = C.M \ (mod \ n)$

Step 3. return C

Montgomery Exponentiation Method

Another method we used for analysis is montgomery exponentiation. The montgomery exponentiation method is explained by the following algorithm:

function MonPro($\bar{a}, \bar{b}$)

Step 1. $t = \bar{a}.\bar{b}$

Step 2. $m = t.n' \ mod \ r$

Step 3. $u = (t + m.n)/r$

Step 4. **if** $u \geq n$ **then return** $u - n$

else return u

Montgomery Exponentiation Method

function ModExp(M, e, n) { n is odd }

Step 1. Compute n' using Euclid's algorithm

Step 2. $\bar{M} = M.r \ mod \ n$

Step 3. $\bar{C} = 1.r \ mod \ n$

Step 4. **for** $i = k - 1$ **down to** 0 **do**

Step 5. $\quad \bar{C} = MonPro(\bar{C}, \bar{C})$

Step 6. $\quad$ **if** $e_i = 1$ **then** $\bar{C} = MonPro(\bar{M}, \bar{C})$

Step 7. $C = MonPro(\bar{C}, 1)$

Step 8. return C

Hardware Specifications: UDOO Board

Results are compared between UDOO board and standard PC with following configurations:

	UDOO	PC
CPU	1 x [ARMv7 Processor rev 10 (v7l)]	4 x [Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz]
Physical Memory	800 MB	3.7 GB
OS	Ubuntu 12.04 32-bit	Ubuntu 14.04 64-bit

Diffie-Hellman Parameters

- Prime p and generator g:
  1. IETF standard 1024 and 2048-bit primes and corresponding generators (having 160-bit and 224-bit prime order subgroups). RFC5114 [3]
  2. Random 'safe' primes generated using OpenSSL library having given number of bits and generator g is taken as 5). (Safe primes are of the form $2p + 1$, where p is also prime)
- Safe primes are of the form $2p + 1$, where p is also prime. Safe primes offers security against Pohlig and Hellman attacks, but require more computation.
- Parameters a and b : random primes with given number of bits

Results on UDOO

Avg time required for key generation on UDOO (in seconds):

Key-size (bits)	Binary Exponentiation	Montgomery Exponentiation	OpenSSL Exponentiation
256	0.005414833	0.009804000	0.001707833
512	0.023968332	0.047772333	0.008993666
1024	0.148043826	0.284063160	0.058445834
2048	0.294208169	0.564812660	0.114655666

Comparing UDOO and PC

Avg time required for key generation (in seconds):

Key-size (bits)	Binary Exponentiation	Montgomery Exponentiation	OpenSSL Exponentiation
1024 [UDOO]	0.148043826	0.284063160	0.058445834
1024 [PC]	0.007844172	0.018422132	0.001439296
2048 [UDOO]	0.294208169	0.564812660	0.114655666
2048 [PC]	0.015397863	0.036434080	0.002855158

Conclusions

D-H key generation performance:

- Binary exponentiation 2-3 times faster than Montgomery exponentiation.
- OpenSSL implementation of exponentiation is 3 times faster than our binary exponentiation.
- This could be because OpenSSL implementation is highly efficient than our implementation.

Key Learnings from the Project

We learnt a lot from this project. Some of the learnings are as follows:

- Hands-on development on the UDOO platform.
- The use of OpenSSL library for handling arbitrary length integer operations in C programming language.
- The implementation of security protocols and operations in secure and efficient manner.

Future Work

Future iterations of this project can include:

- Improving efficiency of Montgomery exponentiation implementation for UDOO board.
- Using the key exchange implementation to communicate messages between remote clients and testing its security.

References

[1] Diffie, W.; Hellman, M. (1976). "New directions in cryptography". IEEE Transactions on Information Theory 22 (6): 644 - 654. doi:10.1109/TIT.1976.1055638

[2] Open SSL - Cryptography and SSL/TLS Toolkit [https://www.openssl.org/]

[3] IETF Standard RFC5114 [http://tools.ietf.org/html/rfc5114]