CSE 40567 / 60567: Computer Security

Cryptography 4
Homework #2 has been released. It is due Thursday, Feb. 6th at 11:59PM

See Assignments Page on the course website for details
Advanced Encryption Standard (AES)

• If you need a symmetric key algorithm, this is the one to use
• Based on Rijndael, winner of the 2001 NIST AES competition
• Key sizes: 128-, 192- or 256-bit
• Block size: 128-bit
• Rounds: 10, 12 or 14 (depending on key size)
AES is a substitution-permutation network

• Works via a combination of both substitution and permutation operations
  ‣ Fast in both hardware and software

• Operates on 4x4 column-major order matrix of bytes (state)

\[
\begin{bmatrix}
  b_0 & b_4 & b_8 & b_{12} \\
  b_1 & b_5 & b_9 & b_{13} \\
  b_2 & b_6 & b_{10} & b_{14} \\
  b_3 & b_7 & b_{11} & b_{15}
\end{bmatrix}
\]
High-level overview of AES

1. **KeyExpansions** — round keys are derived from $k$ using a key schedule

2. **InitialRound**
   1. AddRoundKey

3. **Rounds**
   1. SubBytes
   2. ShiftRows
   3. MixColumns
   4. AddRoundKey

4. **Final Round (no MixColumns)**
   1. SubBytes
   2. ShiftRows
   3. AddRoundKey
SubBytes Step

Each byte in the state is replaced with its entry in a fixed 8-bit lookup table (a substitution box), $S$; $b_{i,j} = S(a_{i,j})$. 
# ShiftRows Step

Bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row.

<table>
<thead>
<tr>
<th>No change</th>
<th>Shift 1</th>
<th>Shift 2</th>
<th>Shift 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{0,0}$</td>
<td>$a_{1,0}$</td>
<td>$a_{2,0}$</td>
<td>$a_{3,0}$</td>
</tr>
<tr>
<td>$a_{0,1}$</td>
<td>$a_{1,1}$</td>
<td>$a_{2,1}$</td>
<td>$a_{3,1}$</td>
</tr>
<tr>
<td>$a_{0,2}$</td>
<td>$a_{1,2}$</td>
<td>$a_{2,2}$</td>
<td>$a_{3,2}$</td>
</tr>
<tr>
<td>$a_{0,3}$</td>
<td>$a_{1,3}$</td>
<td>$a_{2,3}$</td>
<td>$a_{3,3}$</td>
</tr>
</tbody>
</table>

![ShiftRows Diagram]

The diagram shows the shifted state after the ShiftRows step.
MixColumns Step

Each column of the state is multiplied with a fixed polynomial $c(x)$. 

\[
\begin{array}{c|c|c|c}
    a_{0,1} & a_{0,2} & a_{0,3} \\
    a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\
    a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\
    a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
    b_{0,1} & b_{0,2} & b_{0,3} \\
    b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\
    b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\
    b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\
\end{array}
\]
AddRoundKey Step

Each byte of the state is XORed with a byte of the round subkey.
AES Compatible Stream Modes

Cipher Feedback (CFB) Mode, a variation of CBC that turns a block cipher into a self-synchronizing stream cipher

Downside: Slow
AES Compatible Stream Modes

Output Feedback (OFB) Mode also turns a block cipher into a self-synchronizing stream cipher

Downside: **Slow**
AES Compatible Stream Modes

Counter (CTR) Mode generates the next keystream block by encrypting successive values of a counter combined with a nonce (IV)

*Use this mode when a stream cipher is needed*
AES functions in LibreSSL / OpenSSL

```c
#include <openssl/aes.h>

int AES_set_encrypt_key(const unsigned char *userKey, const int bits,
             AES_KEY *key);
int AES_set_decrypt_key(const unsigned char *userKey, const int bits,
             AES_KEY *key);

int private_AES_set_encrypt_key(const unsigned char *userKey, const int bits,
             AES_KEY *key);
int private_AES_set_decrypt_key(const unsigned char *userKey, const int bits,
             AES_KEY *key);

void AES_cbc_encrypt(const unsigned char *in, unsigned char *out,
             size_t length, const AES_KEY *key,
             unsigned char *ivec, const int enc);

void AES_ctr128_encrypt(const unsigned char *in, unsigned char *out,
             size_t length, const AES_KEY *key,
             unsigned char ivec[AES_BLOCK_SIZE],
             unsigned char ecount_buf[AES_BLOCK_SIZE],
             unsigned int *num);
```
Public Key Cryptography

- A symmetric algorithm is like a safe
  - The key is the combo
  - Anyone with the combo can open the safe
  - Anyone without the combo must learn safecracking

1976: Whitfield Diffie and Martin Hellman introduce alternative paradigm with two keys:

- Public (sharable): $k_P$
- Private (secret): $k_S$
Sending a message using public key crypto

1. Alice and Bob agree on a public key crypto system

2. Bob sends Alice his public key
Sending a message using public key crypto

3. Alice encrypts her message using Bob’s public key and sends it back to Bob

$$X, k_{P,B} \quad \rightarrow \quad \{X\}^k_{P,B} \quad \rightarrow$$

4. Bob decrypts Alice’s message using his private key

$$\{X\}^k_{P,B, B} \quad \rightarrow \quad X$$
Sending a message using public-key crypto and a central key repository

• The protocol we just saw is clunky: Alice needs to contact Bob before sending him a message

• If public keys are stored in an accessible database, the protocol is simplified to three steps:

1. Alice gets Bob’s key from the database
Sending a message using public-key crypto and a central key repository

2. Alice encrypts her message using Bob’s public key and sends it back to Bob

3. Bob decrypts Alice’s message using his private key

Bob isn’t involved until he reads his message
Practical public key crypto application: email encryption via GPG

- Free implementation of OpenPGP standard (RFC4880)
- Public key encryption and signing of data and communications
- Versatile key management system

# apt-get install gnupg
How does public key crypto work?

• How do we generate two keys that work together?
• How do we make sure the public key doesn’t reveal any information about the private key?
• How can we design the algorithm to be resilient to chosen plaintext attacks: an attacker can choose any message to encrypt
Trapdoor Function

\[ f : \text{easy} \]

\[ f^{-1} : \text{hard} \]

\[ f_t^{-1} : \text{easy with trapdoor } t \]
RSA

• First full-fledged public key encryption algorithm
  - 1978: Ron Rivest, Adi Shamir, and Leonard Adleman

• If you need a public key encryption algorithm, this is one to use

• Simple to understand and implement
  - Security comes from the difficulty of factoring large numbers


RSA Key Generation

1. Choose two random large prime numbers of equal length, $p$ and $q$.

2. Compute the product (modulus): $n = p \cdot q$

3. Randomly choose the encryption key $e$ such that $e$ and $(p - 1)(q - 1)$ are relatively prime

4. Use the extended Euclidean algorithm to compute the decryption key, $d$, such that:

$$ed \equiv 1 \mod (p - 1)(q - 1)$$

$$d = e^{-1} \mod ((p - 1)(q - 1))$$
RSA Key Generation

- \( d \) and \( n \) are also relatively prime
- The numbers \( e \) and \( n \) are the public key \( k_P \)
- The number \( d \) is the private key \( k_S \)
- The two primes, \( p \) and \( q \), are no longer needed. They must be discarded and never revealed.
RSA Encryption

1. Divide a message $X$ into numerical blocks $x_i$ that are smaller than $n$ (with binary data, choose the largest power of 2 less than $n$).

   The encrypted messages, $\{X\}$, will be made up of similarly sized message blocks $\{x_i\}$, of the same length.

2. Apply the encryption formula:

   $$\{x_i\} = x_i^e \mod n$$
RSA Decryption

1. Take each encrypted block \( \{x_i\} \) and apply the decryption formula:

\[
x_i = \{x_i\}^d \mod n
\]

The message could have also been encrypted with \( d \) and decrypted with \( e \). The use of the keys is arbitrary!
A short example

Key Generation:

1. \( p = 47 \) and \( q = 71 \)

2. \( n = p \cdot q = 3337 \)

3. The encryption key \( e \) must have no factors in common with: \( (p - 1)(q - 1) = 46 \cdot 70 = 3220 \)

4. Choose \( e \) at random to be 79. Calculate \( d \):
   \[ d = 79^{-1} \mod 3220 = 1019 \quad \leftarrow \text{solved via extended Euclidean Alg.} \]

5. Publish \( e \) and \( n \), keep \( d \) secret. Discard \( p \) and \( q \).
A short example

Encryption:

1. $X = 6882326879666683$

2. Break $X$ into small blocks (3 digits here):

   $x_1 = 688$
   $x_2 = 232$
   $x_3 = 687$
   $x_4 = 966$
   $x_5 = 668$
   $x_6 = 003$

3. The first block is encrypted as: $688^{79} \mod 3337 = 1570 = \{x_1\}$

4. Repeating the encryption operation on subsequent blocks yields: $\{X\} = 1570 2756 2091 2276 2423 158$
A short example

Decryption:

1. First block: $1570^{1019} \mod 3337 = 688 = x_1$

2. Subsequent blocks are recovered in the same manner
How do we generate prime numbers?

• If we always need different prime numbers, won’t we run out?
  ‣ No: there are approximately $10^{151}$ primes 512 bits in length or less (by comparison, there are only $10^{77}$ atoms in the universe)

• What if two people accidentally pick the same prime number?
  ‣ Improbable (you have much better odds at the Powerball)

• What if somebody creates a database of all primes?
  ‣ Impossible (would exceed physical limits of the universe)
How do we generate prime numbers?

1. Select a random number of a desired length

2. Apply a **Fermat primality test** (best with base 2 for speed optimization)

3. Apply a certain number of **Miller-Rabin primality tests** (depending on the length and allowed error rate)

Pre-selection: test divisions by small prime numbers (up to few hundreds) or sieve out primes up to 10,000 - 1,000,000 considering many prime candidates of the form $b + 2i$

- large number
- up to a few thousands
Speed of RSA

Slow — use to transfer symmetric session keys for the bulk of the encryption

3.1 Ghz Intel Core i7
The numbers are in 1000s of bytes per second processed.

$ openssl speed rsa

```
sign   verify   sign/s verify/s
rsa  512 bits  0.000134s  0.000010s  7437.2   103127.2
rsa 1024 bits  0.000454s  0.000022s  2203.4   45325.5
rsa 2048 bits  0.002358s  0.000063s  424.0    15980.3
rsa 4096 bits  0.014172s  0.000200s  70.6     4993.3
```

$ openssl speed aes

```
type       16 bytes    64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc 145688.81k 149145.51k 150745.15k  147818.26k  150304.14k
aes-192 cbc 127548.47k 128854.67k 130738.82k  130399.58k  129314.59k
aes-256 cbc 111384.28k 107228.01k 111353.87k  113593.73k  116542.79k
```
RSA key sizes

“Attacks always get better; they never get worse.”
-NSA Aphorism (as related by Bruce Schneier)

What is considered to be secure in 2019?

Research suggests 1024-bit moduli are too small (i.e., NSA can factor them):


If you are protecting 128-bit AES keys, 2,048-bit moduli are adequate
# RSA functions in LibreSSL / OpenSSL

```c
#include <openssl/rsa.h>

RSA * RSA_new(void);
void RSA_free(RSA *rsa);

int RSA_public_encrypt(int flen, unsigned char *from,
unsigned char *to, RSA *rsa, int padding);
int RSA_private_decrypt(int flen, unsigned char *from,
unsigned char *to, RSA *rsa, int padding);
int RSA_private_encrypt(int flen, unsigned char *from,
unsigned char *to, RSA *rsa, int padding);
int RSA_public_decrypt(int flen, unsigned char *from,
unsigned char *to, RSA *rsa, int padding);

RSA *RSA_generate_key(int num, unsigned long e,
void (*callback)(int,int,void *), void *cb_arg);

int RSA_check_key(RSA *rsa);
```