Introduction

Public-key Algorithms

Definition (Public-key Algorithms)

Public-key cryptographic algorithms use different keys for encryption and decryption.

- Bob’s public key: $P_B$
- Bob’s secret key: $S_B$

\[
E_{P_B}(M) = C \\
D_{S_B}(C) = M \\
D_{S_B}(E_{P_B}(M)) = M
\]
Public Key Protocol

- Key-management is the main problem with symmetric algorithms – Bob and Alice have to somehow agree on a key to use.
- In public key cryptosystems there are two keys, a public one used for encryption and one private one for decryption.

1. Alice and Bob agree on a public key cryptosystem.
2. Bob sends Alice his public key, or Alice gets it from a public database.
3. Alice encrypts her plaintext using Bob’s public key and sends it to Bob.
4. Bob decrypts the message using his private key.

Public Key Encryption: Key Distribution

- Advantages: $n$ key pairs to communicate between $n$ parties.
- Disadvantages: Ciphers (RSA,...) are slow; keys are large

A Hybrid Protocol

- In practice, public key cryptosystems are not used to encrypt messages – they are simply too slow.
- Instead, public key cryptosystems are used to encrypt keys for symmetric cryptosystems. These are called session keys, and are discarded once the communication session is over.

1. Bob sends Alice his public key.
2. Alice generates a session key $K$, encrypts it with Bob’s public key, and sends it to Bob.
3. Bob decrypts the message using his private key to get the session key $K$.
4. Both Alice and Bob communicate by encrypting their messages using $K$.
Hybrid Encryption Protocol...

Alice

Bob

1. **Introduction**

2. **RSA**
   - Algorithm
   - Example
   - Correctness
   - Security

3. **GPG**

4. **Elgamal**
   - Algorithm
   - Example
   - Correctness
   - Security

5. **Diffie-Hellman Key Exchange**
   - Diffie-Hellman Key Exchange
   - Example
   - Correctness
   - Security

6. **Summary**

---

**RSA**

- RSA is the best known public-key cryptosystem. Its security is based on the (believed) difficulty of factoring large numbers.
- Plaintexts and ciphertexts are large numbers (1000s of bits).
- Encryption and decryption is done using modular exponentiation.

**RSA: Algorithm**

- **Bob** (Key generation):
  1. Generate two large random primes \( p \) and \( q \).
  2. Compute \( n = pq \).
  3. Select a small odd integer \( e \) relatively prime with \( \phi(n) \).
  4. Compute \( \phi(n) = (p - 1)(q - 1) \).
  5. Compute \( d = e^{-1} \mod \phi(n) \).
- \( P_B = (e, n) \) is Bob’s RSA public key.
- \( S_B = (d, n) \) is Bob’s RSA private key.

- **Alice** (encrypt and send a message \( M \) to Bob):
  1. Get Bob’s public key \( P_B = (e, n) \).
  2. Compute \( C = M^e \mod n \).

- **Bob** (decrypt a message \( C \) received from Alice):
  1. Compute \( M = C^d \mod n \).
RSA: Algorithm Notes

- How should we choose $e$?
  - It doesn’t matter for security; everybody could use the same $e$.
  - It matters for performance: 3, 17, or 65537 are good choices.
- $n$ is referred to as the **modulus**, since it’s the $n$ of $\mathbb{Z}_n$.
- You can only encrypt messages $M < n$. Thus, to encrypt larger messages you need to break them into pieces, each $< n$.
- Throw away $p$, $q$, and $\phi(n)$ after the key generation stage.
- Encrypting and decrypting requires a single modular exponentiation.

RSA Example: Key Generations

1. Select two primes: $p = 47$ and $q = 71$.
2. Compute $n = pq = 3337$.
3. Compute $\phi(n) = (p - 1)(q - 1) = 3220$.
4. Select $e = 79$.
5. Compute $d = e^{-1} \mod \phi(n)$
   $$d = 79^{-1} \mod 3220 = 1019$$
6. $P = (79, 3337)$ is the RSA public key.
7. $S = (1019, 3337)$ is the RSA private key.

RSA Example: Encryption

1. Encrypt $M = 6882326879666683$.
2. Break up $M$ into 3-digit blocks:
   $$m = \langle 688, 232, 687, 966, 668, 003 \rangle$$
   Note the padding at the end.
3. Encrypt each block:
   $$c_1 = m_1^e \mod n$$
   $$= 688^{79} \mod 3337$$
   $$= 1570$$
   We get:
   $$c = \langle 1570, 2756, 2091, 2276, 2423, 158 \rangle$$

RSA Example: Decryption

1. Decrypt each block:
   $$m_1 = c_1^d \mod n$$
   $$= 1570^{1019} \mod 3337$$
   $$= 688$$
In-Class Exercise: Goodrich & Tamassia R-8.18

Show the result of encrypting $M = 4$ using the public key $(e, n) = (3, 77)$ in the RSA cryptosystem.

In-Class Exercise: Goodrich & Tamassia R-8.20

- Alice is telling Bob that he should use a pair of the form $(3, n)$ or $(16385, n)$ as his RSA public key if he wants people to encrypt messages for him from their cell phones.
- As usual, $n = pq$, for two large primes, $p$ and $q$.
- What is the justification for Alice’s advice?

In-Class Exercise: Stallings pp. 270-271

1. Generate an RSA key-pair using $p = 17$, $q = 11$, $e = 7$.
2. Encrypt $M = 88$.
3. Decrypt the result from 2.

RSA Correctness

- We have
  
  $$C = M^e \mod n$$
  $$M = C^d \mod n.$$  

- To show correctness we have to show that decryption of the ciphertext actually gets the plaintext back, i.e that, for all $M < n$

  $$C^d \mod n = (M^e)^d \mod n$$
  $$= M^{ed} \mod n$$
  $$= M.$$
From the key generation step we have
\[ d = e^{-1} \mod \phi(n) \]
from which we can conclude that
\[ ed \mod \phi(n) = 1 \]
\[ ed = k\phi(n) + 1 \]

Case 1, \( M \) is relatively prime to \( n \):
\[ C^d \mod n = M^{ed} \mod n = M^{k\phi(n)+1} \mod n = M\cdot(M^{\phi(n)})^k \mod n = M \cdot 1^k \mod n = M \mod n = M \]

\[ M^{\phi(n)} \mod n = 1 \] follows from Euler’s theorem.

Theorem (Euler)
Let \( x \) be any positive integer that’s relatively prime to the integer \( n > 0 \), then
\[ x^{\phi(n)} \mod n = 1 \]

Assume that \( M \) is not relatively prime to \( n \), i.e. \( M \) has some factor in common with \( n \), since \( M < n \).

There are two cases:
1. \( M \) is relatively prime with \( q \) and \( M = ip \), or
2. \( M \) is relatively prime with \( p \) and \( M = iq \).

We consider only the first case, the second is similar.
**RSA Correctness: Case 2...**

- We can now prove Case 2, for \( M = ip \):

\[
C^d \mod n = M^{ed} \mod n = M^{k\phi(n)+1} \mod n = (M + Mhq) \mod n = (M + (ip)hq) \mod n = (M + (ih)pq) \mod n = (M + (ih)n) \mod n = (M \mod n) + ((ih)n \mod n) = M \mod n = M
\]

**RSA Security**

- Summary:
  1. Compute \( n = pq \), \( p \) and \( q \) prime.
  2. Select a small odd integer \( e \) relatively prime with \( \phi(n) \).
  3. Compute \( \phi(n) = (p-1)(q-1) \).
  4. Compute \( d = e^{-1} \mod \phi(n) \).
  5. \( P_B = (e, n) \) is Bob’s RSA public key.
  6. \( S_B = (d, n) \) is Bob’s RSA private key.

Since Alice knows Bob’s \( P_B \), she knows \( e \) and \( n \).

- If she can compute \( d \) from \( e \) and \( n \), she has Bob’s private key.
- If she knew \( \phi(n) = (p-1)(q-1) \) she could compute \( d = e^{-1} \mod \phi(n) \) using Euclid’s algorithm.
- If she could factor \( n \), she’d get \( p \) and \( q \)!

**Security of Cryptosystems by Failed Cryptanalysis**

- Propose a cryptographic scheme.
- If an attack is found, patch the scheme. GOTO 2.
- If enough time has passed \( \Rightarrow \) The scheme is secure!
- How long is enough?
  1. It took 5 years to break the Merkle-Hellman cryptosystem.
  2. It took 10 years to break the Chor-Rivest cryptosystem.

- If we can factor \( n \), we can find \( p \) and \( q \) and the scheme is broken.
- As far as we know, factoring is hard.
- We need \( n \) to be large enough, 2,048 bits.
### RSA Factoring Challenge

#### On December 3, 2003, a team of researchers in Germany and several other countries reported a successful factorization of the challenge number RSA-576.

The factors are:

```
3980750864240649373971255050550386491199064362
342526708406385189579586389957261766583317
472772146107435302356223071973048224632914695
3209791164598521711030271725636550397527
```

### RSA Factoring Challenge

#### The factors are:

```
The factors are:
```

### RSA Factoring Challenge

#### On December 3, 2003, a team of researchers in Germany and several other countries reported a successful factorization of the challenge number RSA-576.

The factors are:

```
302097150656996221305168759307650257059
```

### RSA Factoring Challenge

#### The factors are:

```
16347336458092538443133838650908598417836700330
923121811180532933310104508151212181167515579
190087128186482211312685157393541397547189678996
51549366685390808270380210449895719126146575
```

### RSA Factoring Challenge

#### The effort took approximately 30 2.2GHz-Opteron-CPU years according to the submitters, over five months of calendar time.

### RSA Factoring Challenge

#### The factors are:

```
5079250470826529678092941651854908494259377522750
527859194993870037695575868449338127796130892303925609525326160282
3676490316035531714470932347180669680969
```

### RSA Factoring Challenge

#### The factors are:

```
191954852900733724822783525742388645014691733602477652346609
```

### RSA Factoring Challenge

#### The factors are:

```
923121811180532933310104508151212181167515579
190087128186482211312685157393541397547189678996
51549366685390808270380210449895719126146575
```

### RSA Factoring Challenge

#### The factors are:

```
191954852900733724822783525742388645014691733602477652346609
```

### RSA Factoring Challenge

#### The factors are:

```
16347336458092538443133838650908598417836700330
923121811180532933310104508151212181167515579
190087128186482211312685157393541397547189678996
51549366685390808270380210449895719126146575
```
RSA Security: How to use RSA

- Two plaintexts $M_1$ and $M_2$ are encrypted into ciphertexts $C_1$ and $C_2$.
- But, RSA is deterministic!
- If $C_1 = C_2$ then we know that $M_1 = M_2$!
- Also, side-channel attacks are possible against RSA, for example by measuring the time taken to encrypt.

Outline

1. Introduction
2. RSA
   - Algorithm
   - Example
   - Correctness
   - Security
3. GPG
   - Algorithm
   - Example
   - Correctness
   - Security
4. Diffie-Hellman Key Exchange
   - Diffie-Hellman Key Exchange
   - Example
   - Correctness
   - Security
5. Summary

Software – GPG

- gpg is a public domain implementation of pgp.
- Supported algorithms:
  - **Pubkey:** RSA, RSA-E, RSA-S, ELG-E, DSA
  - **Cipher:** 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH, CAMELLIA128, CAMELLIA192, CAMELLIA256
  - **Hash:** MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
  - **Compression:** Uncompressed, ZIP, ZLIB, BZIP2


Key generation: Bob

```bash
> gpg --gen-key
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
What keysize do you want? (2048)
Key is valid for? (0)
Key does not expire at all
Real name: Bobby
Email address: bobby@gmail.com
Comment: recipient
You need a Passphrase to protect your secret key.
Enter passphrase: Bob rocks
Repeat passphrase: Bob rocks
```
Key generation: Alice

> gpg --gen-key

Please select what kind of key you want:
(1) RSA and RSA (default)
(2) DSA and Elgamal
(3) DSA (sign only)
(4) RSA (sign only)
Your selection? 1
What keysize do you want? (2048)
Key is valid for? (0)
Key does not expire at all
Real name: Alice
Email address: alice@gmail.com
Comment: sender
You need a Passphrase to protect your secret key.
Enter passphrase: Alice is cute
Repeat passphrase: Alice is cute

Exporting the Key

> gpg --armor --export Bobby

-----BEGIN GPG PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.11 (Darwin)
mQENBE83U28BCADTV0kHpNjWzk7yEzMh1NJcm0tmUYfn4hzgYTDsP2ot10UhfJ4qEZCuPoxECIZ479k3yPbVzMN2JC48Ht9j1kVnDPLCRongyRsKo04wAG70YAyHwa7/USejGwJzOMUuM3SqwHdo1/OXs3P8LABTQXrQf9kF8UNL1AhriIVBcaeIK44NML6

----------
EBHmAM7iiWgWi6/6qEmN462QEmoR86vWhQL3LQ6p/FUaBA==
=FZ78

-----END GPG PUBLIC KEY BLOCK-----

Encryption

We can encrypt a message using Bobby’s key:

> cat message

Attack at dawn

> gpg --recipient bobby --armor --encrypt message

-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.11 (Darwin)
hQEMA97v9lbZUpHvAQf/a9Qk1XMiMzBWy5yyZBtNrg7FcrIq+xgXVUXNN86tZtERF42elU6QvamDzfCHq+3e0er4Y5xN+sp/L91txi6uwFohrgCGJq/AFUKgqyKH2e4gR8Y1BuPm9b1c7uXzMRMOUB8t5KquYQBLybeP29ttD9iIZ/LJ1izSPjSjE1700Gp7bPEBatStVotunYW/fX0zXudU8XN1lknsnZn2t1XnO3QFMu8Do/tF511RfTEcl4SrtV4vshgXbNSpTg9sZaIZUyvU2CJqyYkCtgT77TdtzK3fta8UN+CYQvU2Q8nntFhYwBNmPQheFNdDqezB+P0RqDuDllYuNJaVij3CL7kgRtNyRkGaQ7Qmkg2nlw2jQY2G2E8CC7pNXY9U3KYM19hKAl6U5fPo08ndFp8vowBbB2swzjxJ5Y7zIR2uxdLYd7W4m
=BJ1A

-----END PGP MESSAGE-----

Decryption

Bobby can now decrypt the message using his private key:

> gpg --decrypt message.asc

You need a passphrase to unlock the secret key for user:
"Bobby (recipient) <bobby@gmail.com>"
2048-bit RSA key, ID D95291EF, created 2012-02-12
(main key ID 9974031B)
Enter passphrase: Bob rocks

gpg: encrypted with 2048-bit RSA key, ID D95291EF, created 2012-02-12
"Bobby (recipient) <bobby@gmail.com>"
Attack at dawn
The keyring

> gpg --list-keys
/Users/collberg/.gnupg/pubring.gpg

----------------------------------
pub 2048R/9974031B 2012-02-12
uid Bobby (recipient) <bobby@gmail.com>
sub 2048R/D95291EF 2012-02-12

pub 2048R/4EC8A0CB 2012-02-12
uid Alice (sender) <alice@gmail.com>
sub 2048R/B901E082 2012-02-12

GPG41/83

The keyring...

> gpg --list-secret-keys
/Users/collberg/.gnupg/secring.gpg

----------------------------------
sec 2048R/9974031B 2012-02-12
uid Bobby (recipient) <bobby@gmail.com>

sec 2048R/4EC8A0CB 2012-02-12
uid Alice (sender) <alice@gmail.com>

GPG42/83

Sign and Encrypt

- Bob can sign his message before sending it to Alice:

  > gpg -se --recipient alice --armor message

  You need a passphrase to unlock the secret key for user: "Bobby (recipient) <bobby@gmail.com>"
  2048-bit RSA key, ID 9974031B, created 2012-02-12

  Enter passphrase: Bob rocks

  >>>>BEGIN PGP MESSAGE
  Version: GnuPG v1.4.11 (Darwin)

  hQEMA7osp1S5aCCAGAsQsSs+Urf0f3KHT7p7cTwugpcJ9oUAGxk/KQ0DHIEOv
  8XEAsCwZ8aZK11XqB5d/9hCm9Mup2NECih08crVyff7NTWFysTBegAm10q3y46o
  QpIgPbcdYzqt58e/8wP6x1MzUtzzxKLB+Rj/Zg35ZVioYL
  =oiY8
  
  >>>>END PGP MESSAGE

GPG43/83

Check Signature and Decrypt

- Alice can now decrypt the message and check the signature:

  > gpg --decrypt message.asc

  You need a passphrase to unlock the secret key for user: "Alice (sender) <alice@gmail.com>"
  2048-bit RSA key, ID B901E082, created 2012-02-12 (main key ID 4EC8A0CB)

  Enter passphrase: Alice is cute

  gpg: encrypted with 2048-bit RSA key, ID B901E082, created 2012-02-12
  "Alice (sender) <alice@gmail.com>"

  Attack at dawn
  gpg: Signature made Sat Feb 11 23:10:59 2012 MST using RSA key ID 9974031B
  gpg: Good signature from "Bobby (recipient) <bobby@gmail.com>"

GPG44/83
Symmetric Encryption Only

> gpg --cipher-algo=AES --armor --symmetric message
Enter passphrase: sultana
Repeat passphrase: sultana
> cat message.asc
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.11 (Darwin)

jA0EbwMCgZ3PbfSZxJ1g0ksBBboTMLEVQ2q9HkTR5y9FIoX9nbsyohr0XeQLF1cfwWcg+dzlMS6D7DE3w2Cw2LX50kYcU17MUc8wJLD4zAdRqPAGDma+sP4=
=UtI4
-----END PGP MESSAGE-----

> gpg message.asc
> gpg: AES encrypted data
Enter passphrase: sultana
gpg: encrypted with 1 passphrase

> cat message
Attack at dawn

Deleting Keys

> gpg --delete-secret-keys bobby
sec 2048R/9974031B 2012-02-12 Bobby (recipient) <bobby@gmail.com>
Delete this key from the keyring? (y/N) y
This is a secret key! - really delete? (y/N) y

> gpg --delete-keys bobbypub 2048R/9974031B 2012-02-12 Bobby (recipient) <bobby@gmail.com>
Delete this key from the keyring? (y/N) y

Generating Primes

- Generate a prime number of the given number of bits:

> gpg --gen-prime 1 16
C487
> gpg --gen-prime 1 1024
D34D4347ED013242E06811BC561C6587D75AED33D1BE9C964DE648E22
9D88B5B0AFL93445F9B48B135B99C88A8C0E5331C6226CBF6D70031
4A8CC84C7B363BEB7D7BBB29E545D199339263F5F2E9F184BA9D5
05B579858FC6149CFO9E6C569730C3BD1E62B378C8DFAF423388DC
BA999A21EC9C4BF8C60AACDCBC607AC5

Generating Random Numbers

- Generate 100 (base64 encoded) random bytes:

> gpg --armour --gen-random 0 100
e0zV16jbe/Dma9VF201MgZxE1RA4S8TwWw6Kp8+o1kJdtBm2
AjKFSVaj/d3zG/9KqmNj76symEUZ3e0fWzAqLExzJusur5sK
CBomfPus2QyJjNOgVbpJ7XKL4t1iWNJtnw==
Print Message Digests

> gpg --print-mds message

**MD5** = 36 D1 A5 12 17 CD 34 FC 04 F5 6C C4 91 39 C7 59
**SHA1** = 6DA4 473A 00CE 7AB6 7B6F 884D 1F75 6633 C21A 56DB
**RMD160** = D1DE 4194 C0CD 3AED 30F3 38CD 68F3 800F CCF0 3B87
**SHA224** = B4E94780 1AA1A9C3 418F72D8 651BA995 83284003 EBE8E183A 589702EE
**SHA256** = 83EF405 07696578 9D4BBDA7 D9337200 5F2AE6CB A2696FDE 69694D12 AFE70E4A
**SHA384** = 7AC39A0C 945844F1 1316BB46 9CF7EEAE E892A178 2D20E4CA E78E686C 1A91C8C F1BD0FD1 3D42BEA2 88AF6A4F E3705474
**SHA512** = 7AC39A0C 945844F1 1316BB46 9CF7EEAE E892A178 2D20E4CA E78E686C 1A91C8C F1BD0FD1 3D42BEA2 88AF6A4F E3705474

Goal: Read a message encrypted with gpg

1. Decrypt the message itself (OR)
2. Determine symmetric key used to encrypt the message by other means (OR)
3. Get recipient to help decrypt message (OR)
4. Obtain private key of recipient.

http://www.schneier.com/paper-attacktrees-fig7.html

Goal: Read a message encrypted with gpg...

**GPG**: Print Message Digests

Decrypted message itself:

1. Break asymmetric encryption (OR)
   1. Brute force break asymmetric encryption (OR)
   2. Mathematically break asymmetric encryption (OR)
   3. Break RSA (OR)
   4. Factor RSA modulus/calculate Elgamal discrete log
   5. Cryptanalyze asymmetric encryption (OR)
       1. General cryptanalysis of RSA/Elgamal (OR)
       2. Exploit weakness in RSA/Elgamal (OR)
       3. Timing attack on RSA/Elgamal
2. Break symmetric-key encryption
   1. Brute force break symmetric-key encryption
   2. Cryptanalysis of symmetric-key encryption

Goal: Read a message encrypted with gpg...

Determine symmetric key by other means:

1. Fool sender into encrypting message using public key whose private key is known (OR)
   1. Convince sender that fake key (with known private key) is the key of the intended recipient
   2. Convince sender to encrypt with more than one key—the real key of the recipient and a key whose private key is known
   3. Have the message encrypted with a different public key in the background, unbeknownst to the sender
2. Have the recipient sign the encrypted public key (OR)
3. Monitor the sender’s computer memory (OR)
4. Monitor the receiver’s computer memory (OR)
5. Determine key from pseudo-random number generator (OR)
   1. Determine state of randseed during encryption (OR)
   2. Implant virus that alters the state of randseed. (OR)
   3. Implant software that affects the choice of symmetric key.
   4. Implant virus that that exposes public key.
What immediately becomes apparent from the attack tree is that breaking the RSA or IDEA encryption algorithms are not the most profitable attacks against PGP. There are many ways to read someone’s PGP-encrypted messages without breaking the cryptography. You can capture their screen when they decrypt and read the messages (using a Trojan horse like Back Orifice, a TEMPEST receiver, or a secret camera), grab their private key after they enter a passphrase (Back Orifice again, or a dedicated computer virus), recover their passphrase (a keyboard sniffer, TEMPEST receiver, or Back Orifice), or simply try to brute force their passphrase (I can assure you that it will have much less entropy than the 128-bit IDEA keys that it generates).

In the scheme of things, the choice of algorithm and the key length is probably the least important thing that affects PGP’s overall security. PGP not only has to be secure, but it has to be used in an environment that leverages that security without creating any new insecurities.

http://www.schneier.com/paper-attacktrees-fig7.html
Elgamal

The Elgamal cryptosystem relies on the inherent difficulty of calculating discrete logarithms. It is a probabilistic scheme:
- a particular plaintext can be encrypted into multiple different ciphertexts;
- ⇒ ciphertexts become twice the length of the plaintext.

Elgamal: Algorithm

- **Bob** (Key generation):
  1. Pick a prime $p$.
  2. Find a generator $g$ for $\mathbb{Z}_p$.
  3. Pick a random number $x$ between 1 and $p - 2$.
  4. Compute $y = g^x \mod p$.
     - $P_B = (p, g, y)$ is Bob's RSA public key.
     - $S_B = x$ is Bob’s RSA private key.
- **Alice** (encrypt and send a message $M$ to Bob):
  1. Get Bob’s public key $P_B = (p, g, y)$.
  2. Pick a random number $k$ between 1 and $p - 2$.
  3. Compute the ciphertext $C = (a, b)$:
     - $a = g^k \mod p$
     - $b = M g^k \mod p$
- **Bob** (decrypt a message $C = (a, b)$ received from Alice):
  1. Compute $M = b(a^x)^{-1} \mod p$.

Elgamal: Algorithm Notes

- Alice must choose a different random number $k$ for every message, or she’ll leak information.
- Bob doesn’t need to know the random value $k$ to decrypt.
- Each message has $p - 1$ possible different encryptions.
- The division in the decryption can be avoided by use of Lagrange’s theorem:

  $$M = b \cdot (a^x)^{-1} \mod p = b \cdot a^{p-1-x} \mod p$$
Elgamal: Finding the generator

Computing the generator is, in general, hard.

We can make it easier by choosing a prime number with the property that we can factor \( p - 1 \).

Then we can test that, for each prime factor \( p_i \) of \( p - 1 \):

\[
g^{(p-1)/p_i} \mod p \neq 1
\]

If \( g \) is not a generator, then one of these powers will \( \neq 1 \).

Elgamal Example: Key generation

1. Pick a prime \( p = 13 \).
2. Find a generator \( g = 2 \) for \( Z_{13} \) (see next slide).
3. Pick a random number \( x = 7 \).
4. Compute \( y = g^x \mod p = 2^7 \mod 13 = 11 \).
5. \( P_B = (p, g, y) = (13, 2, 11) \) is Bob’s public key.
6. \( S_B = x = 7 \) is Bob’s private key.

Powers of Integers, Modulo 13

2 is a primitive root modulo 13 because for each integer \( i \in Z_{13} = \{1, 2, 3, \ldots, 12\} \) there’s an integer \( k \), such that \( i = 2^k \mod 13 \):

<table>
<thead>
<tr>
<th>( a^1 )</th>
<th>( a^2 )</th>
<th>( a^3 )</th>
<th>( a^4 )</th>
<th>( a^5 )</th>
<th>( a^6 )</th>
<th>( a^7 )</th>
<th>( a^8 )</th>
<th>( a^9 )</th>
<th>( a^{10} )</th>
<th>( a^{11} )</th>
<th>( a^{12} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>9</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>2</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

Elgamal Example: Encryption

Encrypt the plaintext message \( M = 3 \).

1. Alice gets Bob’s public key \( P_B = (p, g, y) = (13, 2, 11) \).
2. To encrypt:
   1. Pick a random number \( k = 5 \):
   2. Compute:

\[
a = g^k \mod p = 2^5 \mod 13 = 6
\]
\[
b = My^k \mod p = 3 \cdot 11^5 \mod 13 = 8
\]

3. The ciphertext \( C = (a, b) = (6, 8) \).
Elgamal Example: Decryption

- Bob’s private key is \( S_B = x = 7 \).
- Bob receives the ciphertext \( C = (a, b) = (6, 8) \) from Alice.
- Bob computes the plaintext \( M \):
  \[
  M = b \cdot (a^x)^{-1} \mod p \\
  = b \cdot a^{p-1-x} \mod p \\
  = 8 \cdot 6^{13-1-7} \mod 13 \\
  = 8 \cdot 6^5 \mod 13 \\
  = 3
  \]

Elgamal Correctness

- Show that \( M = b \cdot (a^x)^{-1} \mod p \) decrypts.
- We have that
  \[
  a = g^k \mod p \\
  b = My^k \mod p \\
  y = g^x \mod p
  \]
- We get
  \[
  b \cdot (a^x)^{-1} \mod p = (My^k) \cdot ((g^x)^k)^{-1} \mod p \\
  = (My^k) \cdot (g^{kx})^{-1} \mod p \\
  = (M((g^x)^k)) \cdot (g^{kx})^{-1} \mod p \\
  = Mg^{kx} \cdot (g^{kx})^{-1} \mod p \\
  = Mg^{kx} \cdot g^{-kx} \mod p \\
  = M \mod p \\
  = M
  \]

Elgamal Security

- The security of the scheme depends on the hardness of solving the discrete logarithm problem.
- Generally believed to be hard.

In-Class Exercise

- Pick the prime \( p = 13 \).
- Find the generator \( g = 2 \) for \( Z_{13} \).
- Pick a random number \( x = 9 \).
- Compute \( y = g^x \mod p = 2^9 \mod 13 = 5 \)
- \( P_B = (p, g, y) = (13, 2, 5) \) is Bob’s public key.
- \( S_B = x = 9 \) is Bob’s private key.
- Encrypt the message \( M = 11 \) using the random number \( k = 10 \).
- Decrypt the ciphertext from 1.
Key Exchange

A key exchange protocol (or key agreement protocol) is a way for parties to share a secret (such as a symmetric key) over an insecure channel.

With an active adversary (who can modify messages) we can’t reliably share a secret.

With a passive adversary (who can only eavesdrop on messages) we can share a secret.

A passive adversary is said to be honest but curious.

Diffie-Hellman Key Exchange

A classic key exchange protocol.

Based on modular exponentiation.

The secret $K_1 = K_2$ shared by Alice and Bob at the end of the protocol would typically be a shared symmetric key.

Diffie-Hellman: Algorithm

1. All parties (set-up):
   1. Pick $p$, a prime number.
   2. Pick $g$, a generator for $\mathbb{Z}_p$.

2. Alice:
   1. Pick a random $x \in \mathbb{Z}_p, x > 0$.
   2. Compute $X = g^x \mod p$.
     Send $X$ to Bob.

3. Bob:
   1. Pick a random $y \in \mathbb{Z}_p, x > 0$.
   2. Compute $Y = g^y \mod p$.
     Send $Y$ to Alice

4. Alice computes the secret: $K_1 = Y^x \mod p$.
5. Bob computes the secret: $K_2 = X^y \mod p$. 

Example

1. Pick $p = 13$, a prime number.
2. Pick $g = 2$, a generator for $\mathbb{Z}_{13}$.
3. **Alice:**
   1. Pick a random $x = 3$.
   2. Compute $X = g^x \mod p = 2^3 \mod 13 = 8$.
4. **Bob:**
   1. Pick a random $y = 7$.
   2. Compute $Y = g^y \mod p = 2^7 \mod 13 = 11$.
5. **Alice** computes: $K_1 = Y^x \mod p = 11^3 \mod 13 = 5$.
6. **Bob** computes: $K_2 = X^y \mod p = 8^7 \mod 13 = 5$.

$\Rightarrow K_1 = K_2 = 5$.

Diffie-Hellman Key Exchange

In-Class Exercise

1. Let $p = 19$.
2. Let $g = 10$.
3. Let Alice’s secret $x = 7$.
4. Let Bob’s secret $y = 15$.
5. **Alice** computes $K_1$.
6. **Bob** computes $K_2$.

Diffie-Hellman Correctness

- Alice has computed

  $$X = g^x \mod p$$
  $$K_1 = Y^x \mod p.$$

- Bob has computed

  $$Y = g^y \mod p$$
  $$K_2 = X^y \mod p.$$

$$\Rightarrow K_1 = K_2.$$
Diffie-Hellman Security

- The security of the scheme depends on the hardness of solving the discrete logarithm problem.
- Generally believed to be hard.
- **Diffie-Hellman Property:**
  - Given \( p, X = g^x, Y = g^y \)
  - computing \( K = g^{xy} \mod p \)
  - is thought to be hard.

---

Diffie-Hellman: Man-In-The-Middle attack

1. **Alice:**
   - Send \( X = g^x \mod p \) to Bob.
2. **Eve:**
   - Intercept \( X = g^x \mod p \) from Alice.
   - Pick a number \( t \) in \( \mathbb{Z}_p \).
   - Send \( T = g^t \mod p \) to Bob.
3. **Bob:**
   - Send \( Y = g^y \mod p \) to Alice
4. **Eve:**
   - Intercept \( Y = g^y \mod p \) from Bob.
   - Pick a number \( s \) in \( \mathbb{Z}_p \).
   - Send \( S = g^s \mod p \) to Alice.

---

5. Alice and Eve:
   - Compute \( K_1 = g^{sS} \mod p \)
6. Bob and Eve:
   - Compute \( K_2 = g^{yT} \mod p \)

---

Alice: Send \( C = E_{K_1}(M) \) to Bob

Eve:

1. Intercept \( C \).
2. Decrypt: \( M = D_{K_1}(C) \)
3. Re-encrypt: \( C' = E_{K_2}(M) \)
4. Send \( C' \) to Bob.

Bob: Send \( C = E_{K_2}(M) \) to Alice

Eve:

1. Intercept \( C \).
2. Decrypt: \( M = D_{K_2}(C) \)
3. Re-encrypt: \( C' = E_{K_1}(M) \)
4. Send \( C' \) to Alice.
Acknowledgments

Additional material and exercises have also been collected from these sources:

1. Igor Crk and Scott Baker, *620—Fall 2003—Basic Cryptography*.