CS61A Lecture #36: Cryptography

Cryptography: Purposes

- Source: Ross Anderson, *Security Engineering*.
- Cryptography—the study of the design of ciphers—is a tool used to help meet several goals, among them:
  - Privacy: others can't read our messages.
  - Integrity: others can't change our messages without us knowing.
  - Authentication: we know whom we're talking to.
- Some common terminology: we convert from *plaintext* to *ciphertext* (encryption) and back (decryption).
- Although we typically think of text messages as characters, our algorithms generally process streams of *numbers* or *bits*, making use of standard encodings of characters as numbers.

Substitution

- Simplest scheme is just to permute the alphabet:

```
␣abcdefghijklmnopqrstuvwxyz
tyler␣duniabcfghjkmopqsvwxz
```

- So that

```
“so␣long␣and␣thanks␣for␣all␣the␣fish” =>
“ohtchgutygrtpnygbotdhmtycctpn␣tdion”
```

- Problem: If we intercept ciphertext for which we know the plaintext (e.g., we know a message ends with name of the sender), we learn part of the code.
- Even if we have only ciphertext, we can *guess* encoding from letter frequencies.

Stream Ciphers

- **Idea:** Use a different encoding for each character position. Enigma was one example.
- Extreme case is the *One-Time Pad:* Receiver and sender share random key sequence at least as long as all data sent. Each character of the key specifies an unpredictable substitution cipher.
- Example:

```
Messages: attack at dawn|oops cancel that order|attack is back on
Key:      vnchkjskruwisn|tjcdktjdjsahtjkdhjrizn|akjqltpotpfhsdjrsqieha...
Cipher:   vfvhmtrkjtzin |gxrvjvjqlwglqkwgxhlcd|acbqncowkoghuniee
```

(key of 'z' means 'a' $\mapsto$ 'z', 'b' $\mapsto$ '␣', 'c' $\mapsto$ 'a', etc.)

- Unbreakable, but requires lots of shared key information.
- Integrity problems: If I know message is "Pay to Paul N. Hilfinger $100.00" can alter it to "Pay to Paul N. Hilfinger $999.00" [How?]

Aside: A Simple Reversible Combination

- The cipher in the last slide essentially used addition modulo alphabet size as the way to combine plaintext with a key.
- Usually, we use a different method of combining streams: *exclusive or (xor),* which is the "not equal" operations on bits, defined on individual bits by $x \oplus y = 0$ if x and y are the same, else 1.
  Fact: $x \oplus y \oplus x = y$. So,

$$\begin{array}{r} 01100011 \\ \oplus\ 10110101 \\ \hline 11010110 \end{array} \qquad \begin{array}{r} 11010110 \\ \oplus\ 10110101 \\ \hline 01100011 \end{array}$$

- In Python, C, and Java, this operation is written $x\hat{\ }y$.

Using Random-Number Generators

- Python provides a pseudo-random number generator (used for the Hog project, e.g.): from an initial value, produces any number of "random-looking" numbers.
- Consider a function that creates pseudo-random number generators that produce bits, e.g.:

```
import random
def bit_stream(seed):
    r = random.Random(seed)
    return lambda: r.getrandbits(1)
```

- If two sides of a conversation share the same key to use as a seed, can create the same approximation to a one-time pad, and thus communicate secretly.

Example

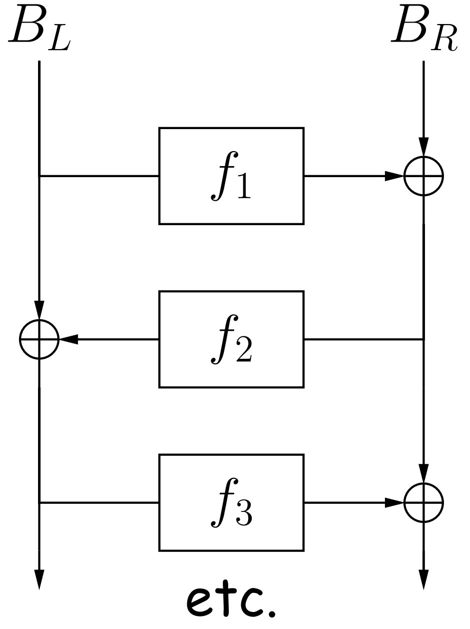
Message	H	e	l	l	o	,		w	o	r	l	d
Message bytes (hex)	48	65	6c	6c	6f	2c	20	77	6f	72	6c	64
Random bytes	5b	49	96	1d	93	eb	6e	2d	a4	1a	52	fb
Encrypted bytes	13	2c	fa	71	fc	c7	4e	5a	cb	68	3e	9f
Encrypted message	?	,	?	q	?	?	N	Z	?	h	>	?

(? in place of non-ASCII)

- Advantage: key can be much shorter than total amount of data.
- Disadvantage: stream of bits isn't really random; may be subject to clever attack (cryptanalysis). This is especially true of standard random number generators like Python's.
- Was used in SSL (Secure Socket Layer) for "secure" web communications.

Block Ciphers

- So far, have encoded bit-by-bit (or byte-by-byte). Another approach is to map blocks of bits at a time, allowing them to be mixed and swapped as well as scrambled.
- Feistel Ciphers: a strategy for generating block ciphers. Break message into $2N$-bit chunks, and break each chunk into N-bit left and right halves, B_L and B_R. Then, put the result through a number of *rounds:*



  - Each f_i is some function mapping N-bit blocks to N-bit blocks that is chosen by your key.
  - f_i does not have to be invertible.
  - Nice feature: to decrypt, run backwards.
  - If the f_i are really chosen well enough, these are very good ciphers with enough rounds.
- The Data Encryption Standard (DES) used this strategy with 12 rounds.

Public Key Cryptography

- So far, our ciphers have been *symmetric:* both sides of a conversation share the same secret information (a key).
- If I haven't contacted someone before, how can we trade secret keys so as to use one of these methods?
- One idea is to use *public keys* so that everyone knows enough to communicate with us, but not enough to listen in when others communicate with us..
- Here, information is *asymmetric:* we publish a *public key* that everyone can know, and keep back a *private key.*
- Rely on it being easy to decipher messages knowing the private key, but impractically difficult without it.
- Unfortunately, we haven't actually proved that any of these *public-key systems* really are essentially impractical to crack, and quantum computing (if made to work at scale) would break the most common one.
- But for now, all is well.

RSA Encryption: The Math

- Fermat's Little Theorem: For p a prime, and $0 \leq a < p$,

$$a^p \bmod p = a.$$

  (for our purposes, mod is remainder (%).

- This generalizes to non-prime numbers as well (Euler's Theorem). In particular, if
  - $n = pq$, where p and q are two (different) primes, and
  - $\lambda(n) =$ lcm$(p-1, q-1)$ (least common multiple), and
  - $0 < e < \lambda(n)$ has no common factors with $\lambda(n)$, and
  - d is computed by solving $d \cdot e \bmod \lambda(n) = 1$, then

  Then for any $M < n$.

$$(M^e)^d \bmod n = M$$

- So
  - Encrypt M by computing $c = m^e \bmod n$
  - Decrypt c by computing $c^d \bmod n = M$.

RSA Public-Key Encrytion

- Basis for an *asymmetric cipher*: Alice picks p, q, and e, and computes d.
- She *publishes* n (which is pq) and e, but keeps d, p, and q secret.
- Anyone can send Alice an encrypted message M by sending $c = M^e \bmod n$.
- But d is *very hard* to compute without knowing p and q, so only Alice can decrypt c, using $M = c^d \bmod n$.

Signatures

- Suppose I receive a message, M, that supposedly comes from you. How do I know it does?
- Using public-key methods, this is relatively easy.
- The RSA scheme works in either direction, $(M^e)^d \bmod n = (M^d)^e \bmod n = M$.
- So, Alice could sign a message by encoding it with her *private* key.
- Bob decodes the message using Alice's *public* key.
- If the result is a legitimate message, Bob can be sure only Alice could have produced it.
- Otherwise, the decoded message will be garbage.
- In practice, we use a rather different scheme, but the underlying idea is based on the same principle.

Authentication on the Web

- When you talk to Amazon, how do you know that the exchange is private and that it really is Amazon you are talking to? (On the Internet, nobody knows you are a dog.)
- Certain companies (such as Verisign) issue signed certificates that say (in effect) “Verisign certifies that Amazon has public key X.”
- Your browser comes equipped with Verisign’s public key so you can verify their signature.
- At that point, you know how to talk to Amazon in a way that only they can understand.
- (As usual, the actual protocol is rather more complex.)

Special Effects: Playing Cards Over the Phone?

- How do I play a card game over the phone, so that neither side can (undetectably) cheat?
- To keep it simple, assume we have a two-person game between Alice and Bob where all cards get revealed.
- For each game, let each side choose a secret encryption key, and assume an algorithm that is *commutative:* if a message is encrypted by secret key A and then by key B, it can be decrypted by the two keys in either order.

Playing Cards Over the Phone: Method

- Alice shuffles and encrypts a deck of cards, and sends them to Bob.
- Bob encrypts the encrypted cards, shuffles them, and sends them back to Alice (doubly encrypted).
- Alice deals cards to Bob by selecting and decrypting them, and sending them to Bob, who can decrypt them.
- Alice deals cards to herself by sending them to Bob, having him decrypt them and send them (now singly encrypted) back to Alice.
- At the end of the game, all information can be revealed, and both sides can check for consistency.

Extra: Zero-Knowledge Proofs

- *Zero-Knowledge Proofs* involve another kind of keeping information hidden even as one communicates certain characteristics of it.
- Suppose I possess the answer to a puzzle, and want to convince you that I have the answer *without* revealing anything about what it is.
- This is an example of a *zero-knowledge proof* (Abadi, Goldwasser, and Rackoff).
- Many uses, such as authentication (I want to prove who I am), or enforcing honesty while maintaining privacy.

Illustration

Example (from Jean-Jacques Quisquater via Wikipedia): Peggy wishes to prove to Victor that she knows the password to get through the hidden door.

1. Peggy chooses to go left or right at random, without Victor seeing.
2. Victor then shouts out which side he wants her to come out.
3. Peggy uses her knowledge or not as necessary to emerge from the desired side.

- After several rounds, Victor is convinced that Peggy knows the password, but doesn't know it himself.
- An observer doesn't know if Peggy and Victor are colluding, and so does not even learn that Peggy knows the password.

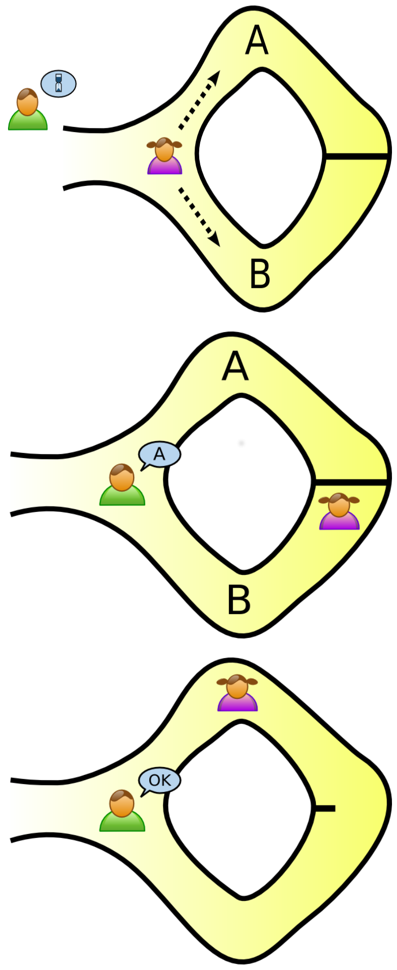


Image sources: CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=313643; https://commons.wikimedia.org/w/index.php?curid=313645; https://commons.wikimedia.org/w/index.php?curid=313648

Illustration

- Example: Prove that I know how to 3-color a graph.
- Given a graph (a network of nodes connected by edges) a *3-coloring* is an assignment of colors to nodes (from a palette of three) such that no nodes joined by an edge have the same color.
- Don't always exist, and hard to find when they do.
- Can I prove to you that I know how to color a particular large graph without letting you know how?
- Strategy: I randomly recolor the graph by substituting colors in my solution (e.g., red becomes green, green becomes blue, blue becomes red), and hide the coloring from you. You select an edge at random, and look at the colors of the nodes at the ends of the edge.
- If I indeed know the solution, the two nodes will always be the same color; otherwise, is a chance you'll pick an edge where the nodes are the same color. Repeating the procedure as many times as you want reveals nothing about my full color scheme, but eventually convinces you that I know a solution.
- Demo: `http://web.mit.edu/~ezyang/Public/graph/svg.html`