Symmetric Key
Cryptography

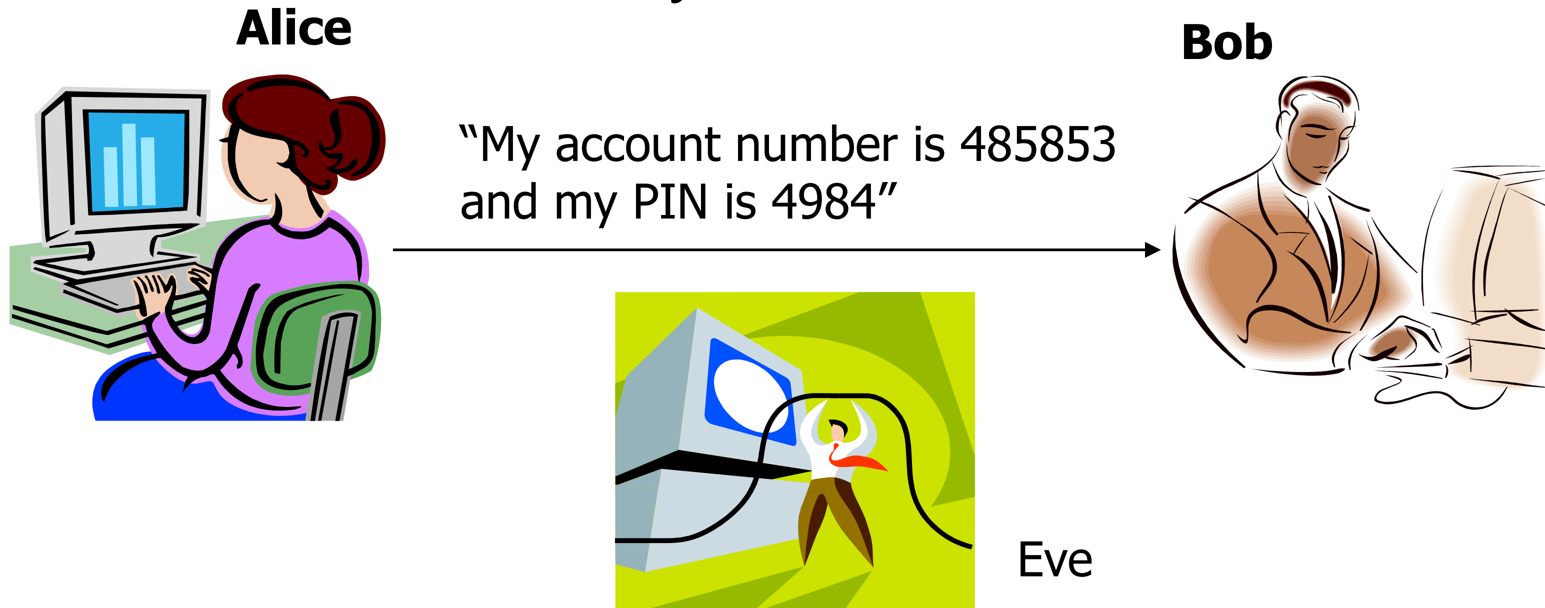
Agenda

- *Cryptography* (crypto)– study of how to mathematically encode & decode messages
- *Cryptographic primitive* (low-level) = algorithm

- Applied Cryptography – how to use crypto to achieve security goals (e.g. confidentiality)
- Primitives build up higher-level protocols (e.g. *digital signature* – only constructible by signer)
- Symmetric Encryption: Alice, Bob use same key

12.1. Introduction to Cryptography

- Goal: Confidentiality



- Message "sent in clear": Eve can overhear
- Encryption unintelligible to Eve; only Bob can decipher with his secret key (shared w/ Alice)

12.1.1. Substitution Ciphers

- Plaintext: **`meet me at central park`**
- Ciphertext: **`phhw ph dw fhqwudo sdun`**

- Plain: `abcdefghijklmnopqrstuvwxyz`
- Cipher: `defghijklmnopqrstuvwxyzabc`

- Key is 3, i.e. shift letter right by 3
- Easy to break due to frequency of letters
- Good encryption algorithm produces output that looks random: equal probability any bit is 0 or 1

12.1.2. Notation & Terminology

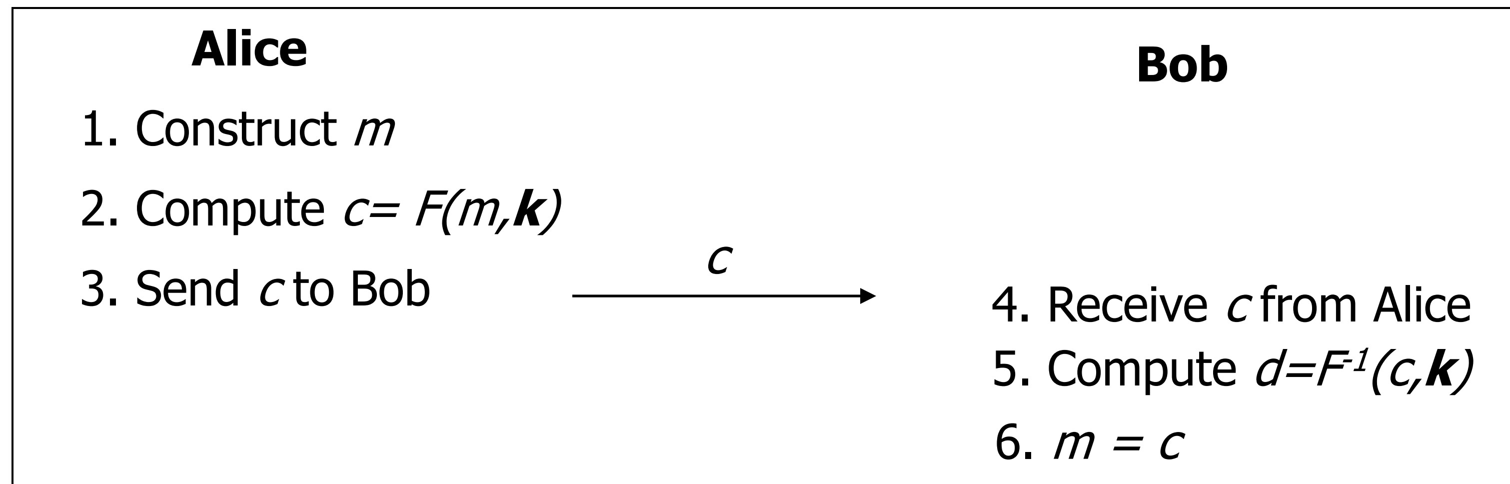
- m = message (plaintext), c = ciphertext
- F = encryption function
- F^{-1} = decryption function
- k = key (secret number)

Cipher (refers to F and F^{-1})

- $c = F(m,k) = F_k(m)$ = encrypted message
- $m = F^{-1}(c,k) = F^{-1}{}_k(c)$ = decrypted message
- Symmetric cipher: $F^{-1}(F(m,k), k) = m$, same key

Symmetric Encryption

- Alice encrypts a message with the ***same*** key that Bob uses to decrypt.



- Eve can see *c*, but cannot compute *m* because *k* is only known to Alice and Bob

12.1.3. Block Ciphers

- Blocks of bits (e.g. 256) encrypted at a time
- Examples of several algorithms:
  - Data Encryption Standard (DES)
  - Triple DES
  - Advanced Encryption Standard (AES) or Rijndael
- Internal Data Encryption Algorithm (IDEA), Blowfish, Skipjack, many more… (c.f. Schneier)

12.1.3. DES

- Adopted in 1977 by NIST

- Input: 64-bit plaintext, 56-bit key (64 w/ parity)
- Parity Bits: redundancy to detect corrupted keys
- Output: 64-bit ciphertext
- Susceptible to Brute-Force (try all 2^{56} keys)
  - 1998: machine Deep Crack breaks it in 56 hours
  - Subsequently been able to break even faster
  - Key size should be at least 128 bits to be safe

12.1.3. Triple DES

- Do DES thrice w/ 3 different keys (slower)
- $c = F(F^{-1}(F(m, k_1), k_2), k_3)$ where $F = DES$
  - Why decrypt with k_2?
  - Backwards compatible w/ DES, easy upgrade
- Keying Options: Key Size (w/ Parity)
  - $k_1 \neq k_2 \neq k_3$: 168-bit (192-bit)
  - $k_1 = k_3 \neq k_2$: 112-bit (128-bit)
  - $k_1 = k_2 = k_3$: 56-bit (64-bit) (DES)

12.1.3. AES (Rijndael)

- Invented by 2 Belgian cryptographers
- Selected by NIST from 15 competitors after three years of conferences vetting proposals
- Selection Criteria:
  - Security, Cost (Speed/Memory)
  - Implementation Considerations (Hardware/Software)
- Key size & Block size: 128, 192, or 256 bits (much larger than DES)
- Rely on algorithmic properties for security, not obscurity

12.1.4. Security by Obscurity: Recap

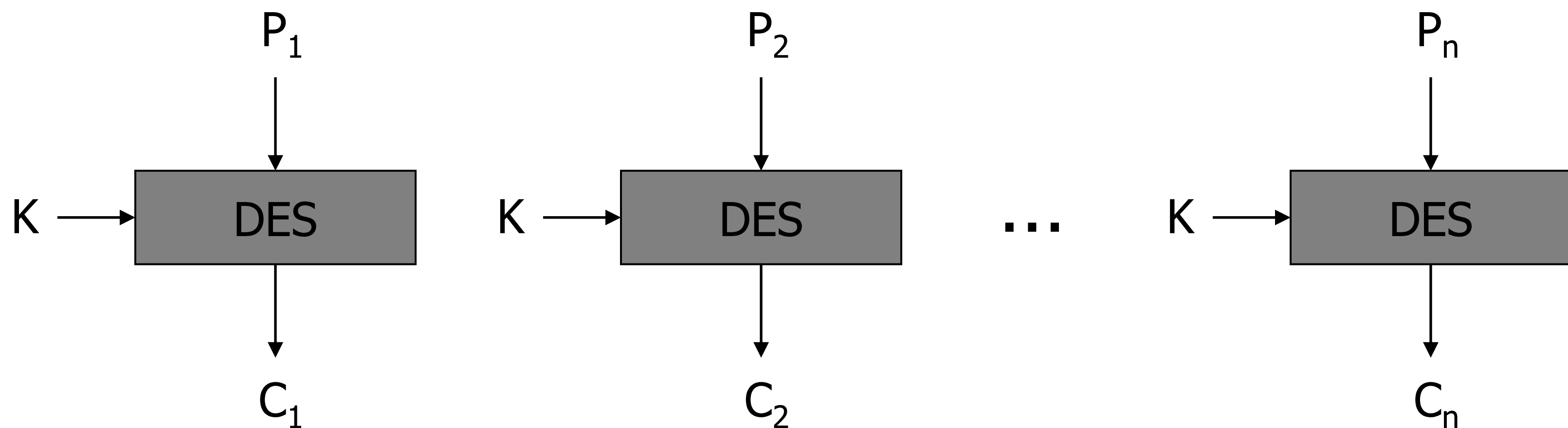
- Design of DES, Triple DES algorithms public
  - Security not dependent on secrecy of implementation
  - But rather on secrecy of key

- Benefits of Keys:
  - Easy to replace if compromised
  - Increasing size by one bit, doubles attacker's work

- If invent own algorithm, make it public! Rely on algorithmic properties (math), not obscurity.

12.1.5. Electronic Code Book

- Encrypting more data: ECB encrypt blocks of data in a large document



- Leaks info about structure of document (e.g. repeated plaintext blocks)

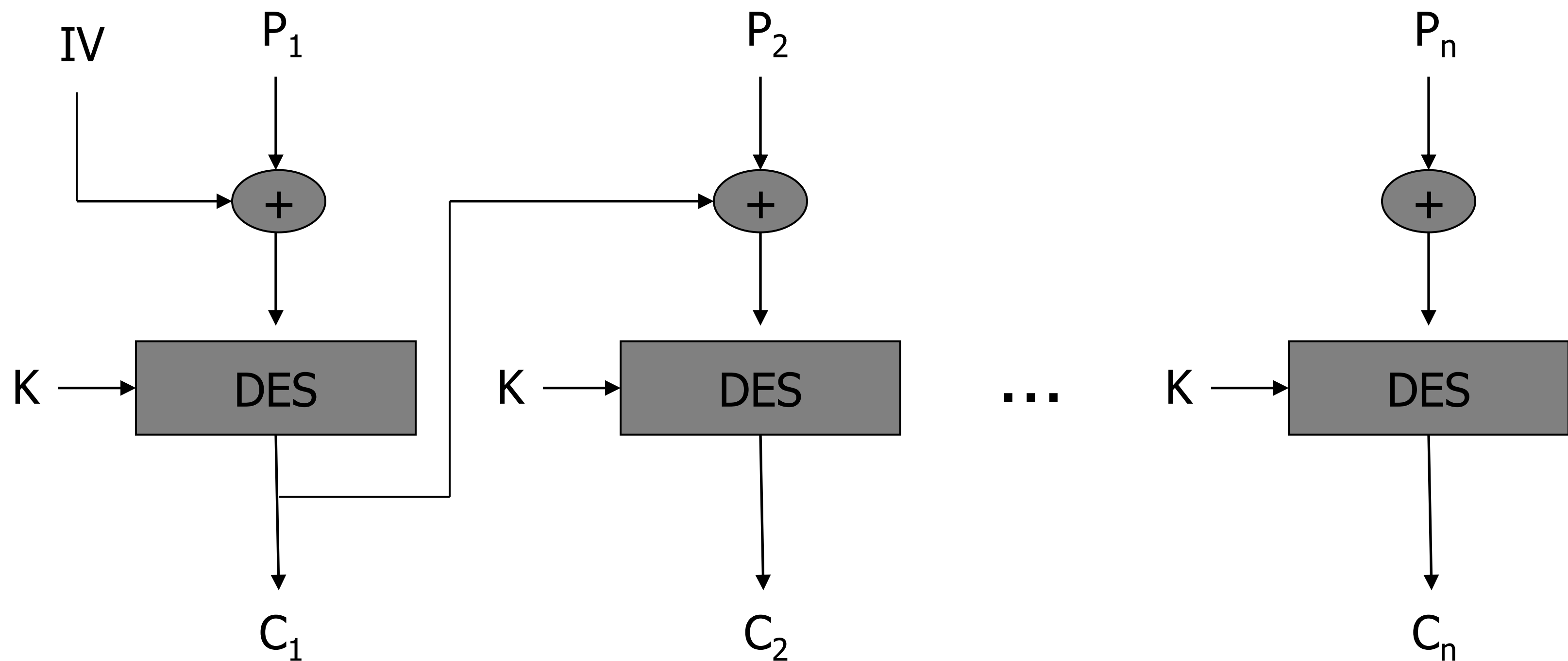
12.1.5. Review of XOR

- Exclusive OR (either x or y but not both)
- Special Properties:
  - x XOR y = z
  - z XOR y = x
  - x XOR z = y

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

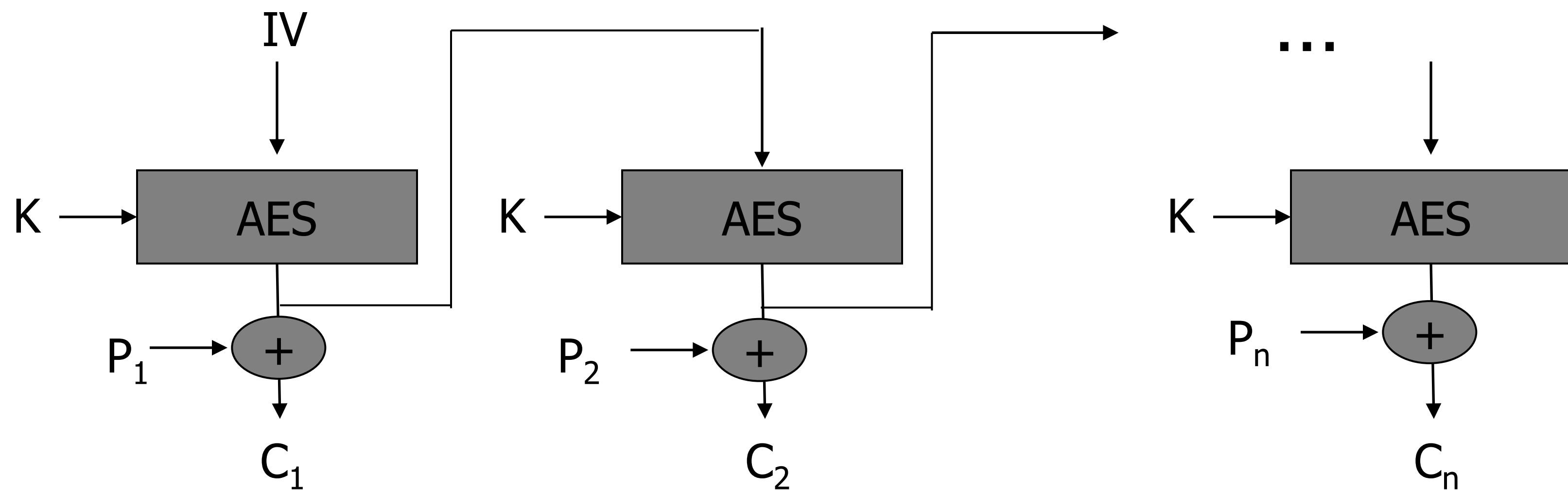
12.1.5. Cipher Block Chaining

- CBC: uses XOR, no patterns leaked!
- Each ciphertext block depends on prev block



12.1.5. Output Feedback (OFB)

- Makes block cipher into stream cipher
- Like CBC, but do XOR after encryption



12.1.6. AES Code Example

- Example Java Class: `AESEncrypter`

- Command-line utility:
  - Create AES key
  - Encrypt & Decrypt with key
  - AES in CBC mode

- Arguments: `<command> <keyfile>`
  - `command` = `createkey|encrypt|decrypt`
  - Input/output from `stdin` and `stdout`

12.1.6. Using AESEncrypter

- Alice generates a key and encrypts a message:

```
$ java AESEncrypter createkey mykey
$ echo "Meet Me At Central Park" |
java AESEncrypter encrypt mykey > ciphertext
```

- She gives Bob `mykey` over *secure* channel, then can send `ciphertext` over insecure channel

- Bob can decrypt Alice's message with `mykey`:

```
$ java com.learnsecurity.AESEncrypter decrypt mykey < ciphertext
Meet Me At Central Park
```

12.1.6. AESEncrytper: Members & Constructor

```
/* Import Java Security & Crypto packages, I/O library */

public class AESEncrypter {
    public static final int IV_SIZE = 16; // 128 bits
    public static final int KEY_SIZE = 16; // 128 bits
    public static final int BUFFER_SIZE = 1024; // 1KB
    Cipher cipher; /* Does encryption and decryption */
    SecretKey secretKey;
    AlgorithmParameterSpec ivSpec; /* Initial Value - IV */
    byte[] buf = new byte[BUFFER_SIZE];
    byte[] ivBytes = new byte [IV_SIZE]; /* inits ivSpec */

    public AESEncrypter(SecretKey key) throws Exception {
        cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        /* Use AES, pad input to 128-bit multiple */
        secretKey = key;
    }
    // ... Methods Follow ...
}
```

12.1.6. AESEncrypter: encrypt()

```
public void encrypt(InputStream in,
                    OutputStream out) throws Exception {
   ivBytes = createRandBytes(IV_SIZE); // create IV & write to output
   out.write(ivBytes);
   ivSpec = new IvParameterSpec(ivBytes);
   cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
   // cipher initialized to encrypt, given secret key, IV

   // Bytes written to cipherOut will be encrypted
   CipherOutputStream cipherOut = new CipherOutputStream(out, cipher);

   // Read in the plaintext bytes and write to cipherOut to encrypt
   int numRead = 0;
   while ((numRead = in.read(buf)) >= 0) // read plaintext
      cipherOut.write(buf, 0, numRead); // write ciphertext
   cipherOut.close(); // padded to 128-bit multiple
}
```

12.1.6. AESEncryptor: decrypt()

```
public void decrypt(InputStream in,
                    OutputStream out) throws Exception {
    // read IV first
    System.in.read(ivBytes);
    ivSpec = new IvParameterSpec(ivBytes);

    cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
    // cipher initialized to decrypt, given secret key, IV

    // Bytes read from in will be decrypted
    CipherInputStream cipherIn = new CipherInputStream(in, cipher);

    // Read in the decrypted bytes and write the plaintext to out
    int numRead = 0;
    while ((numRead = cipherIn.read(buf)) >= 0) // read ciphertext
       out.write(buf, 0, numRead); // write plaintext
    out.close();
}
```

12.1.6. AESEncryptor: main()

```
public static void main (String[] args) throws Exception {
    if (args.length != 2) usage(); // improper usage, print error
    String operation = args[0]; // createkey|encrypt|decrypt
    String keyFile = args[1]; // name of key file
    if (operation.equals("createkey")) {
        FileOutputStream fos = new FileOutputStream(keyFile);
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(KEY_SIZE*8); // key size in bits
        SecretKey skey = kg.generateKey();
        fos.write(skey.getEncoded()); // write key
        fos.close();
    } else {
        byte[] keyBytes = new byte[KEY_SIZE];
        FileInputStream fis = new FileInputStream(keyFile);
        fis.read(keyBytes); // read key
        SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
        AESEncrypter aes = new AESEncrypter(keySpec); // init w/ key
        if (operation.equals("encrypt")) {
            aes.encrypt(System.in, System.out); // Encrypt
        } else if (operation.equals("decrypt")) {
            aes.decrypt(System.in, System.out); // Decrypt
        } else usage(); // improper usage, print error
    }
}
```

12.1.6. AESEncryptor: Helpers

```
/* Generate numBytes of random bytes to use as IV */
public static byte[] createRandBytes(int numBytes)
    throws NoSuchAlgorithmException {
    byte[] bytesBuffer = new byte[numBytes];
    SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
    sr.nextBytes(bytesBuffer);
    return bytesBuffer;
}


/* Display error message when AESEncryptor improperly used */
public static void usage () {
    System.err.println("java com.learnsecurity.AESEncrypter " +
                       "createkey|encrypt|decrypt <keyfile>");
    System.exit(-1);
}
```

12.1.6. AESEncryptor Recap

- Java class `KeyGenerator` can be used to construct strong, cryptographically random keys

- `AESEncrypter`: no integrity protection
  - Encrypted file could be modified
  - So in practice, should tag on a MAC
  - Use different keys for MAC and encryption

- Key Distribution is a challenge (c.f. Ch. 13-14)

12.2. Stream Ciphers

- Much faster than block ciphers
- Encrypts one byte of plaintext at a time
- *Keystream*: infinite sequence (never reused) of random bits used as key
- Approximates theoretical scheme: one-time pad, trying to make it practical with finite keys

12.2.1 One-Time Pad

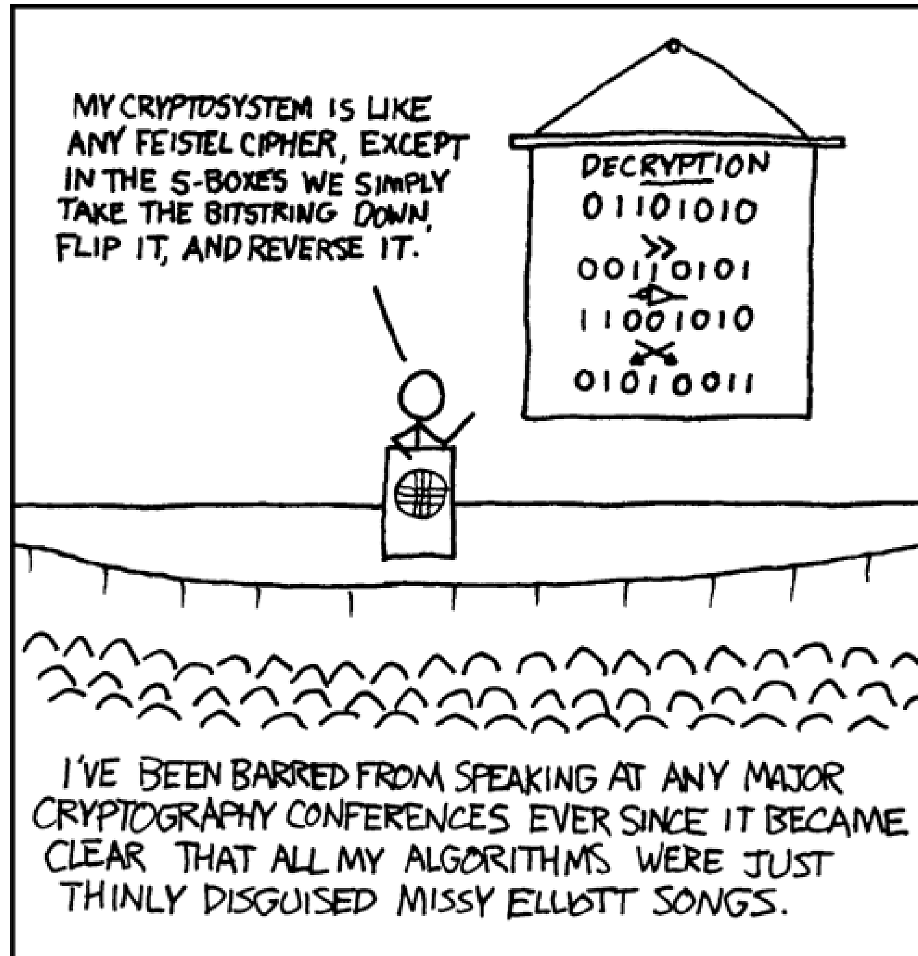
- Key as long as plaintext, random stream of bits
  - Ciphertext = Key XOR Plaintext
  - Only use key once!
- Impractical having key the same size as plaintext (too long, incurs too much overhead)
- Theoretical Significance: “perfect secrecy” (Shannon) if key is random.
  - Under brute-force, every decryption equally likely
  - Ciphertext yields no info about plaintext (attacker’s a priori belief state about plaintext is unchanged)

12.2.2. RC4

- Most popular stream cipher: 10x faster than DES
- Fixed-size key “seed” to generate infinite stream
- *State Table S* that changes to create stream
- Ex: 256-bit key used to seed table (fill it)

```
i = (i + 1) mod 256
j = (j + S[i]) mod 256
swap (S[i],S[j])
t = (S[i]+S[j]) mod 256
K = S[t]
```

12.2.2. … and other ciphers…



Source:
http://xkcd.com/153/

12.2.2. RC4 Pitfalls

- Never use the same key more than once!
- Clients & servers should use different RC4 keys!
  - C -> S: P XOR k [Eve captures P XOR k]
  - S -> C: Q XOR k [Eve captures Q XOR k]
  - Eve: (P XOR k) XOR (Q XOR k) = P XOR Q!!!
  - If Eve knows either P or Q, can figure out the other
- Ex: Simple Mail Transfer Protocol (SMTP)
  - First string client sends server is `HELO`
  - Then Eve could decipher first few bytes of response

12.2.2. More RC4 Pitfalls

- Initial bytes of key stream are “weak”
  - Ex: WEP protocol in 802.11 wireless standard is broken because of this
  - Discard first 256-512 bytes of stream
- Active Eavesdropper
  - Could flip bit without detection
  - Can solve by including MAC to protect integrity of ciphertext