Cryptography

- What is cryptography?
- Related fields:
  - Cryptography ("secret writing"): Making secret messages
    - Turning plaintext (an ordinary readable message) into ciphertext (secret messages that are "hard" to read)
  - Cryptanalysis: Breaking secret messages
    - Recovering the plaintext from the ciphertext
- Cryptology is the science that studies these both
- The point of cryptography is to send secure messages over an insecure medium (like the Internet)

Comes from greek for secret writing. Cryptography is just making a plaintext message secret. Cryptography is a part of Cryptology which also includes cryptoanalysis which is when we try to do the reverse.

Dramatis Personae

- When talking about cryptography, we often use a standard cast of characters
- Alice, Bob, Carol, Dave
  - People (usually honest) who wish to communicate
- Eve
  - A passive eavesdropper, who can listen to any transmitted messages
- Mallory
  - An active Man-In-The-Middle, who can listen to, and modify, insert, or delete, transmitted messages
- Trent
  - A Trusted Third Party

We standardize our fictional characters.

- Alice, Bob, Carol, Dave - good honest hacker
- Eve - a passive eavesdropper that can't do anything
- Mallory - man in the middle
- Trent - trusted third party

Building blocks

- Cryptography contains three major types of components
    - Confidentiality components
        - Preventing Eve from reading Alice's messages
    - Integrity components
        - Preventing Mallory from modifying Alice's messages without being detected
    - Authenticity components
        - Preventing Mallory from impersonating Alice

Confidentiality means we want to prevent Even from reading Alice's messages. We can do this using a cryptosystem

Integrity says we want to prevent Mallory from modifying Alice's messages. We can do this through message authentication (MACs), signature schemes, or cryptographic hash functions (not very secure since hackers can do this easily).

Privacy says we want to prevent MAllory from impersonating Alice. We can do this by passwords or biometrics and such, or through challenge response protocols.

A secret-key system or a public-key system. With a public key system we have two keys. A public key that anyone can use to encrypt something, and a secret private key that can be used to decrypt stuff. A secret key system can be block (requiring data to be a fixed width) or stream.

Kerckhoffs' Principle (19th c.)

- The security of a cryptosystem should not rely on a secret that's hard (or expensive) to change
- So don't have secret encryption methods
  - Then what do we do?
  - Have a large class of encryption methods, instead
    - Hopefully, they're all equally strong
  - Make the class public information
  - Use a secret key to specify which one you're using
  - It's easy to change the key; it's usually just a smallish number

We should only ever rely on a simple secret key that is easy to change. We could have a bunch of different encryption functions. This is not always practical so we encrypt with a long key that varies each time. A system is only as secure as the possible number of keys (we want the widest variety of keys to prevent brute forcing). Even then it is always possible to brute force no matter how long your key is.

Kerckhoffs' Principle (19th c.)

- This has a number of implications:
    - The system is at most as secure as the number of keys
    - Eve can just try them all, until she finds the right one
    - A strong cryptosystem is one where that's the best Eve can do
        - With weaker systems, there are shortcuts to finding the key
    - Example: newspaper cryptogram has 403,291,461,126,605,635,584,000,000 possible keys
    - But you don't try them all; it's way easier than that!

A **strong crypto system** is one where iterating through every possible key is the best possible attack.

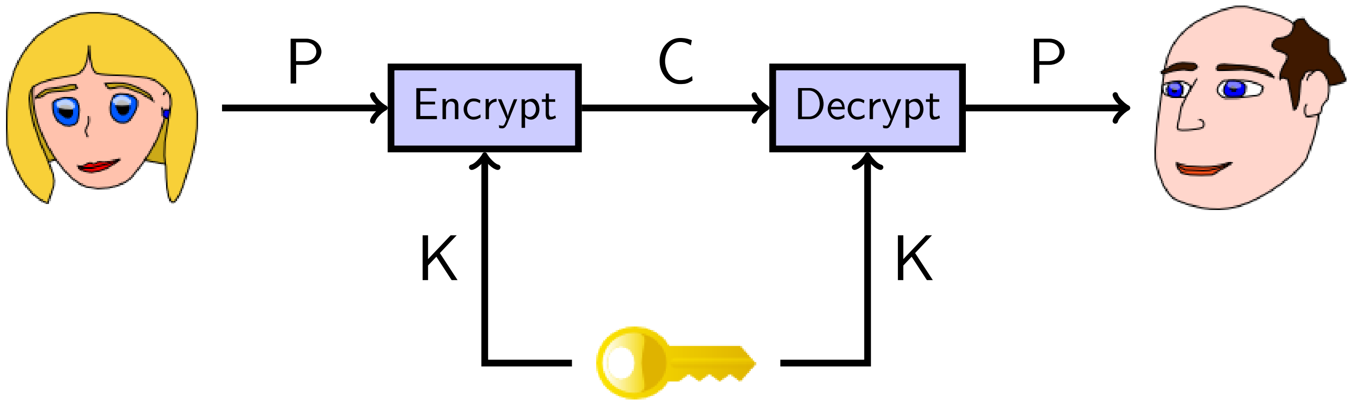
Strong cryptosystems

- What information do we assume the attacker (Eve) has when she's trying to break our system?
- She may:
    - Know the algorithm (the public class of encryption methods)
    - Know some part of the plaintext
    - Know a number (maybe a large number) of corresponding plaintext/ciphertext pairs
    - Have access to an encryption and/or decryption oracle
- And we still want to prevent Eve from learning the key!

Frequently the attacker can get a small bit of plaintext from cypher text. For instance HTTP headers tend to be fairly standard so intercepting an encrypted request allows you to figure out a little bit. Breaking the enigma machine was done by getting a bunch of plaintext cyphertext pairs and analyzing them for a bit before breaking the code. To get these plain/cypher pairs they listened to this one base in russia that would send the same message every day ("weather report: all clear").

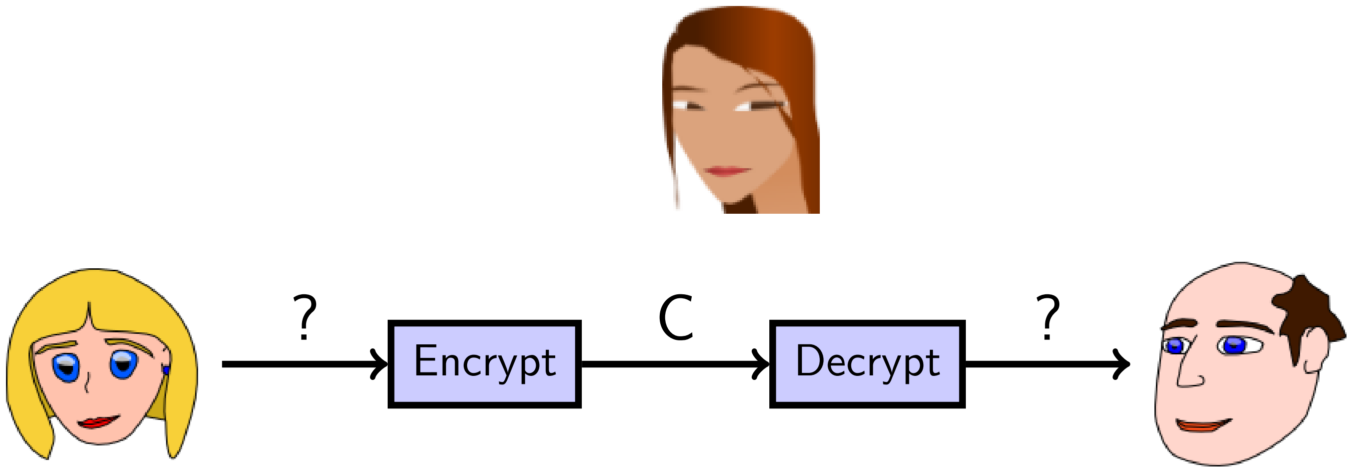
Secret-key encryption

- Secret-key encryption is the simplest form of cryptography
- Also called symmetric encryption
- Used for thousands of years
- The key Alice uses to encrypt the message is the same as the key Bob uses to decrypt it



Secret-key encryption

- Eve, not knowing the key, should not be able to recover the plaintext



Perfect secret-key encryption

- Is it possible to make a completely unbreakable cryptosystem?
- Yes: the One-Time Pad
- It's also very simple:
  - The key is a truly random bitstring of the same length as the message
  - The "Encrypt" and "Decrypt" functions are each just XOR

One-time pad

- Q: Why does “try every key” not work here?
- It’s very hard to use correctly
  - The key must be truly random, not pseudorandom
  - The key must never be used more than once!
    - A “two-time pad” is insecure!
- Q: How do you share that much secret key?
- Used in the Washington / Moscow hotline for many years

Computational security

- In contrast to OTP's "perfect" or "information-theoretic" security, most cryptosystems have "computational" security
  - This means that it's certain they can be broken, given enough work by Eve
- How much is "enough"?
- At worst, Eve tries every key
  - How long that takes depends on how long the keys are
  - But it only takes this long if there are no "shortcuts"!

Some data points

- One computer can try about 17 million keys per second
- A medium-sized corporate or research lab may have 100 computers
- The BOINC project has 13 million computers

BOINC Berkeley Open Infrastructure for Network Computing

- Remember that most computers are idle most of the time (they're waiting for you to type something); getting them to crack keys in their spare time doesn't actually cost anything extra

40-bit crypto

- This was the US legal export limit for a long time
- $2^{40} = 1{,}099{,}511{,}627{,}776$ possible keys
- One computer: 18 hours
- One lab: 11 minutes
- BOINC: 5 ms

A guy came up with pgp which allowed any key length. He ended up getting sued by the US government because encryption was classified as munition.

56-bit crypto

- This was the US government standard (DES) for a long time
- $2^{56} = 72{,}057{,}594{,}037{,}927{,}936$ possible keys
- One computer: 134 years
- One lab: 16 months
- BOINC: 5 minutes

A comity came up with the stupid number of 56 because the government wanted 48 so that they could break it with super computers but people wanted 64 so it would be more secure, so they split the difference.

Cracking DES

www.eff.org
THE ELECTRONIC FRONTIER FOUNDATION
Protecting Rights in the Electronic Frontier
Cracking DES

“DES cracker” machine of Electronic Frontier Foundation

People wanted to show that DES was not all that secure so they held a contest for people to try to break it. Eventually the Electronic Frontier Foundation did it using a special machine. It took about 2 days to crack a key. Researchers eventually maked a system called copacobana that used fpgas (120, cost 10k) that could crack DES in about a week.

128-bit crypto

- This is the modern standard
- $2^{128} =$ 340,282,366,920,938,463,463,374,607, 431,768,211,456 possible keys

- One computer: 635 thousand million million million years

- One lab: 6 thousand million million million years

- BOINC: 49 thousand million million years

Nowadays we use 128 bits. This is so huge that you cannot brute force attack it. So if the encryption is secure it is “unhackable”. That being said, computers are continually getting faster by moore’s law. So in 132 years you can break it in a day.

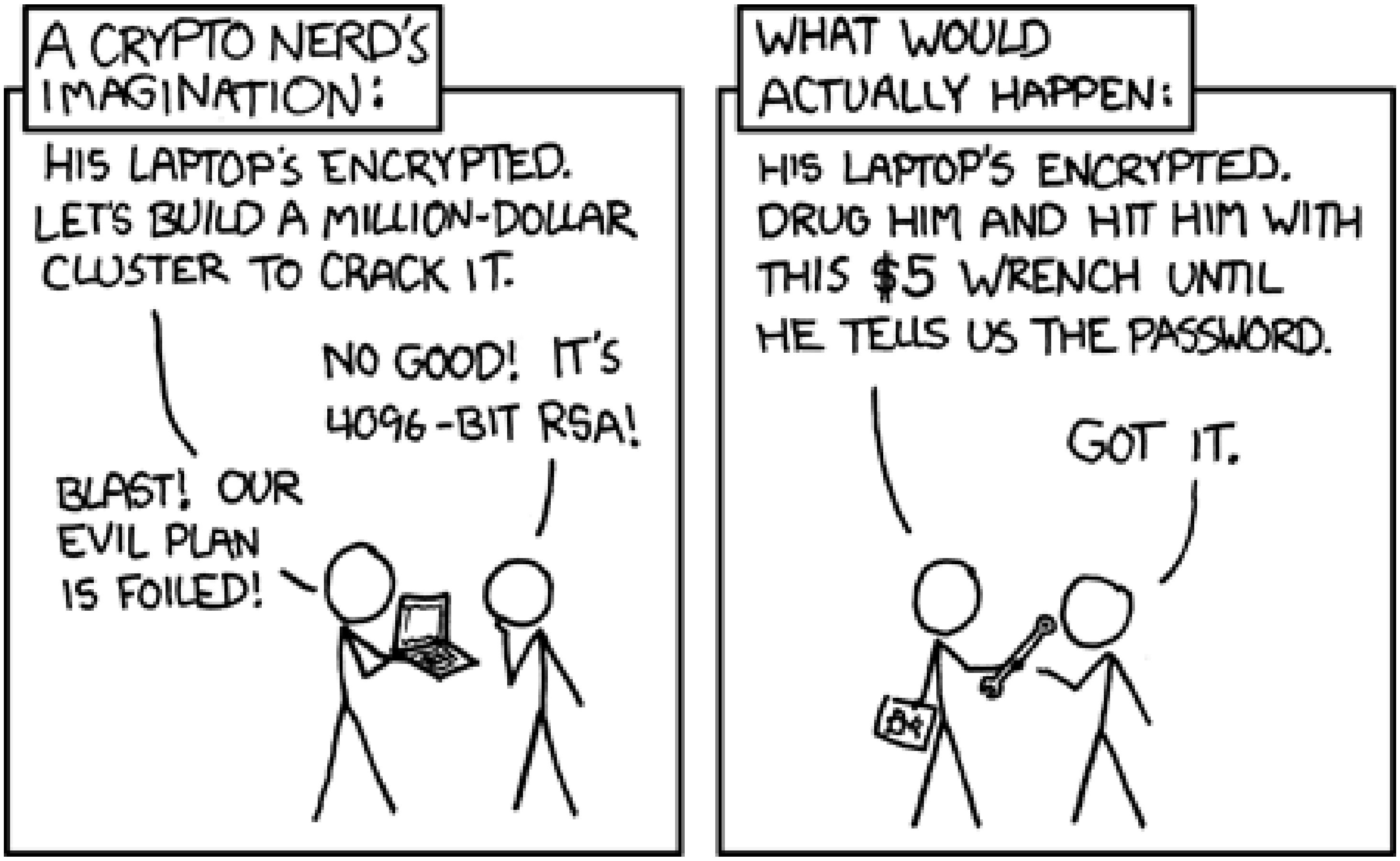
Well, we cheated a bit

- This isn't really true, since computers get faster over time
  - A better strategy for breaking 128-bit crypto is just to wait until computers get 2^{88} times faster, then break it on one computer in 18 hours.
  - How long do we wait? Moore's law says 132 years.
  - If we believe Moore's law will keep on working, we'll be able to break 128-bit crypto in 132 years (and 18 hours) :-)
    - Q: Do we believe this?

An even better strategy

- Don't break the crypto at all!
- There are always weaker parts of the system to attack
    - Remember the Principle of Easiest Penetration
- The point of cryptography is to make sure the information transfer is not the weakest link

Rubber hose cryptanalysis

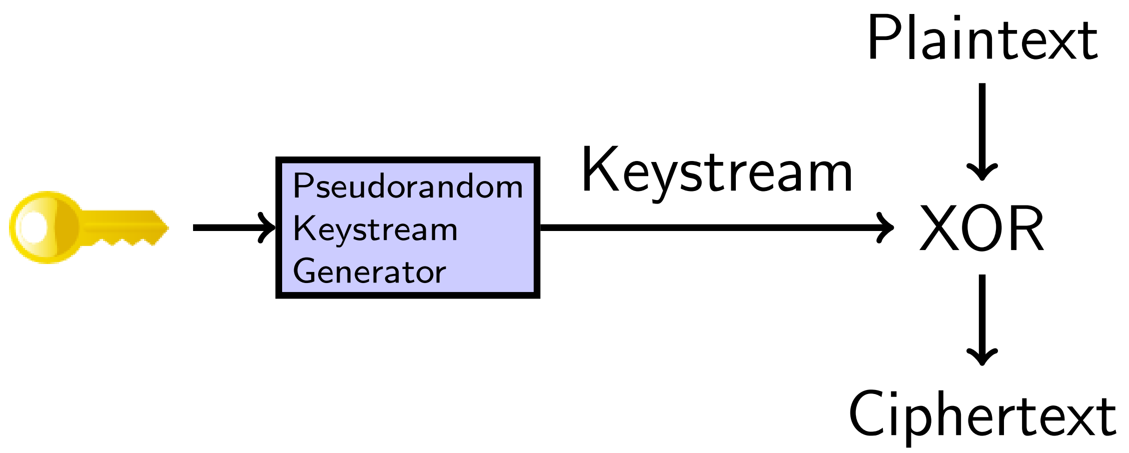

A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

Types of secret-key cryptosystems

- Secret-key cryptosystems come in two major classes
  - Stream ciphers
  - Block ciphers

Stream ciphers

- A stream cipher is what you get if you take the One-Time Pad, but use a pseudorandom keystream instead of a truly random one



- RC4 is the most commonly used stream cipher on the Internet today

RC4 is the most commonly used stream cipher, but it has a ton of problems (new hacks every year).

Stream ciphers

- Stream ciphers can be very fast
  - This is useful if you need to send a lot of data securely
- But they can be tricky to use correctly!
  - What happens if you use the same key to encrypt two different messages?
  - How would you solve this problem without requiring a new shared secret key for each message? Where have we seen this technique before?
- WEP, PPTP are great examples of how not to use stream ciphers

Stream ciphers use the otimes operator since computers are super fast at them. They work using salts, but you have to be very sure that the salts are unique. Take you plaintext, otimes with key and add the salt. The encrypted text is sent along with the salt. Important to note, the salt is public knowledge.

Block ciphers

- Note that stream ciphers operate on the message one bit at a time
- What happens in a stream cipher if you change just one bit of the plaintext?
- We can also use block ciphers
  - Block ciphers operate on the message one block at a time
  - Blocks are usually 64 or 128 bits long
- AES is the block cipher everyone should use today
  - Unless you have a really, really good reason

If we know the location in the string of the value you want to fuck with you can flip bits and such to change that number.

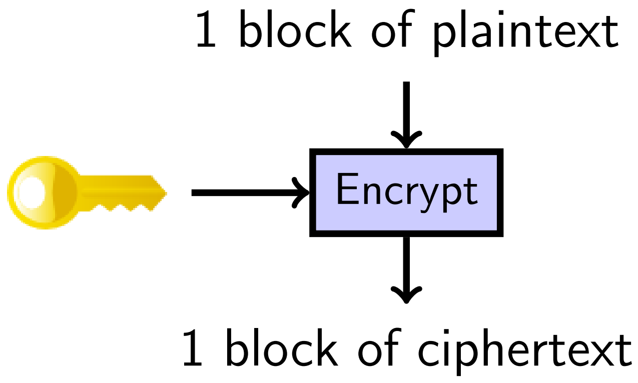
Most block ciphers (including AES) are based on substitution-permutation networks. They repeat their encoding steps a number of times around and around.

1. otimes key bit state
2. apply s-boxes (substitution box) which just takes a box of data and map it to something else (usually every 4 bits)
3. permute bits

At some point it was revealed that by switching to using the NSA's sbox mapping DES became more resistant to attacks around sboxes. This showed that they were about 10 years ahead of academic researchers.

Modes of operation

- Block ciphers work like this:



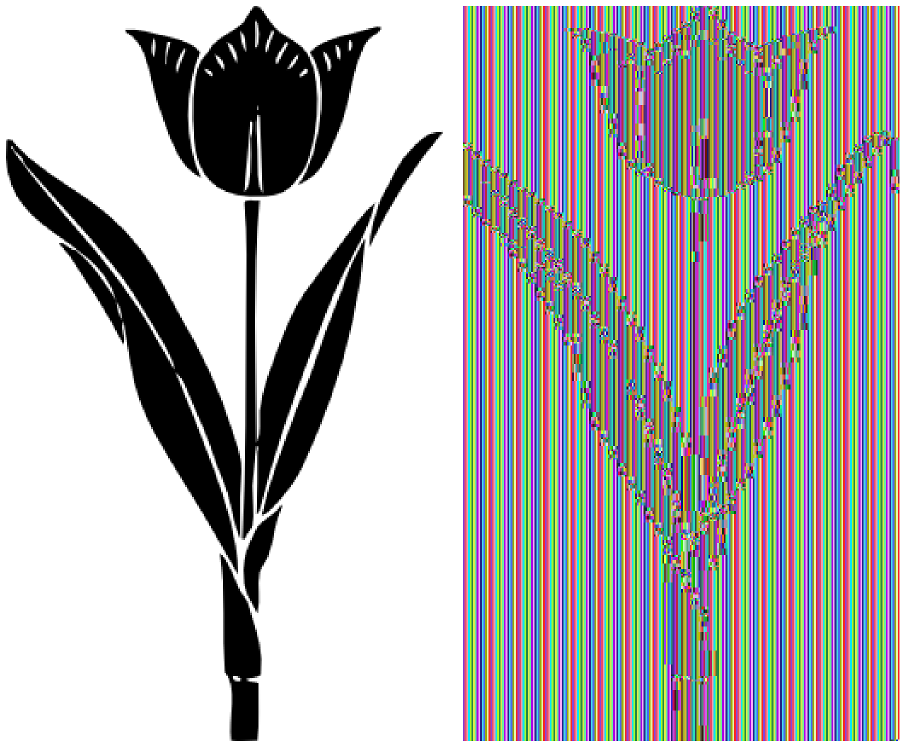
- But what happens when the plaintext is larger than one block?
  - The choice of what to do with multiple blocks is called the mode of operation of the block cipher

We have a block of plaintext, put it through cipher and get garbled mess. Problem arises when the block of plaintext is not the correct length (block must be multiple of some number).

Modes of operation

- The simplest thing to do is just to encrypt each successive block separately.
    - This is called Electronic Code Book (ECB) mode

- But if there are repeated blocks in the plaintext, you'll see the same repeating patterns in the ciphertext:



Take your message and break it up into a bunch of fixed widths and just encrypt each one. This isn't that good because if any of those blocks are the same they will be encrypted the same. These patterns will appear in the encrypted text giving away more information than was intended. Think of a black and white image.

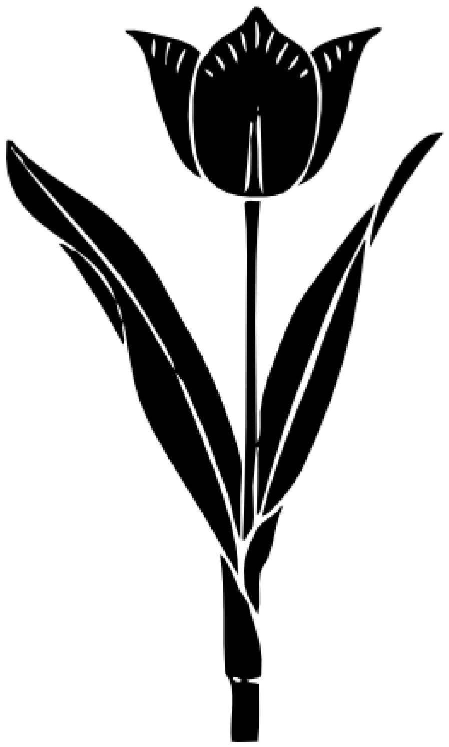
Never use ECB mode for encryption because it does this.

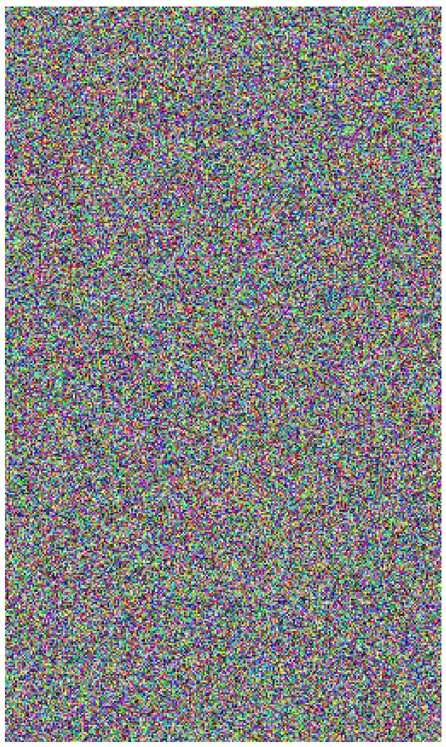
Modes of operation

- There are much better modes of operation to choose from
    - Common ones include Cipher Block Chaining (CBC), Counter (CTR), and Galois Counter (GCM) modes

- Patterns in the plaintext are no longer exposed

- But you need an IV (Initial Value), which acts much like a salt





A solution for this is to make each chunk dependent on the other chunks. Take the cipher text from the previous block and and it with the key for the next block to use as the key for that block. The message becomes a block longer because you need some starting cipher text to include.

Key exchange

- How do Alice and Bob share the secret key?
  - Meet in person; diplomatic courier
  - In general this is very hard
- Or, we invent new technology...

There is a ton of key information flying around so we need a good way to get it to the people who need them.

Public-key cryptography

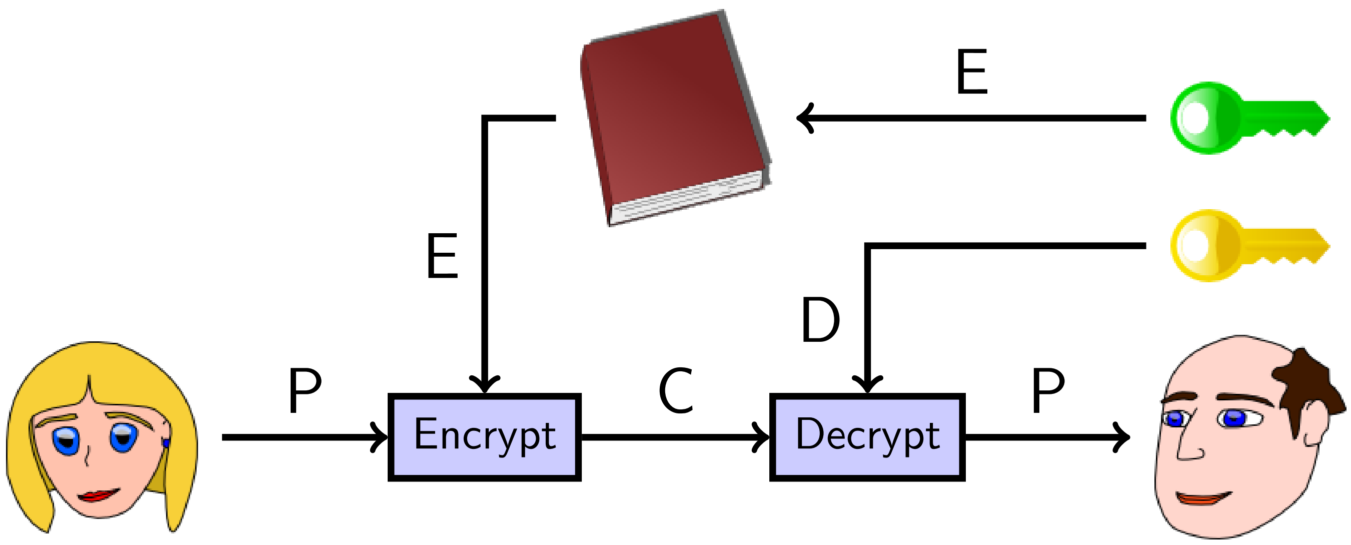
- Invented (in public) in the 1970's
- Also called asymmetric cryptography
  - Allows Alice to send a secret message to Bob without any prearranged shared secret!
  - In secret-key cryptography, the same (or a very similar) key encrypts the message and also decrypts it
  - In public-key cryptography, there's one key for encryption, and a different key for decryption!
- Some common examples:
  - RSA, ElGamal, ECC, NTRU, McEliece

Public-key cryptography

- How does it work?
  - Bob gives everyone a copy of his public encryption key. Alice uses it to encrypt a message, and sends the encrypted message to Bob
  - Bob uses his private decryption key to decrypt the message.
    - Eve can't decrypt it; she only has the encryption key.
    - Neither can Alice!
- So with this, Alice just needs to know Bob's public key in order to send him secret messages
  - These public keys can be published in a directory somewhere

One key lets you encrypt and a different one lets you decrypt. An example is RSA. Public key = everyone knows it, secret key = no one knows it

Public-key cryptography



This is hackable if Mallory publishes a key to Alice to replace Bob's key with her own. When Alice looks up a public key Mallory replaces the encryption key with her own. Then she knows the decryption key so she can just listen in on the conversation.

RSA was published in 1977 by Clifford Cox. It works by generating two large prime numbers, p and q that are inverses of eachother. So n = pq and ab=1 mod(p-1)(q-1). The public key is then (n, a) and the private key is (p,q,b). Encryption is $x^a mod n$ and decryption is $x^b mod n$. You want these numbers to be huge, but this makes doing the calculations very slow.

It is important that these are prime numbers else you could just factor the modulus.

Public key sizes

- Recall that if there are no shortcuts, Eve would have to try 2^{128} things in order to read a message encrypted with a 128-bit key
- Unfortunately, all of the public-key methods we know do have shortcuts
  - Eve could read a message encrypted with a 128-bit RSA key with just 2^{33} work, which is easy!
  - If we want Eve to have to do 2^{128} work, we need to use a much longer public key

Public key systems have to have very large keys because people know them.

Public key sizes

Comparison of key sizes for roughly equal strength

AES	RSA	ECC
80	1024	160
116	2048	232
128	2600	256
160	4500	320
256	14000	512

To have equivalent security parameters the public key crypto will take 3 orders of magnitude more time to encryption than a symmetric crypto system.

Hybrid cryptography

- In addition to having longer keys, public-key cryptography takes a long time to calculate (as compared to secret-key cryptography)
  - Using public-key to encrypt large messages would be too slow, so we take a hybrid approach:
    - Pick a random 128-bit key K for a secret-key cryptosystem
    - Encrypt the large message with the key K (e.g., using AES)
    - Encrypt the key K using a public-key cryptosystem
    - Send the encrypted message and the encrypted key to Bob
  - This hybrid approach is used for almost every cryptography application on the Internet today

To get the best of both worlds we combine symmetric and public encryption. Say we have some plain text P that is a very long message. We generate an AES key called the session key. We get cipher text C by encrypting P using the session key. We then encrypt the session key using Bob's public key using a public key system like RSA.

Is that all there is?

- It seems we've got this "sending secret messages" thing down pat. What else is there to do?
    - Even if we're safe from Eve reading our messages, there's still the matter of Mallory
    - It turns out that even if our messages are encrypted, Mallory can sometimes modify them in transit!
    - Mallory won't necessarily know what the message says, but can still change it in an undetectable way
        - e.g. bit-flipping attack on stream ciphers
    - This is counterintuitive, and often forgotten
        - The textbook even gets this wrong!
- How do we make sure that Bob gets the same message Alice sent?

We still run into integrity problems. Just because it is secure does not mean that the message contents have not been altered in path.

Integrity components

- How do we tell if a message has changed in transit?
- Simplest answer: use a checksum
    - For example, add up all the bytes of a message
    - The last digits of serial numbers (credit card, ISBN, etc.) are usually checksums
    - Alice computes the checksum of the message, and sticks it at the end before encrypting it to Bob. When Bob receives the message and checksum, he verifies that the checksum is correct

This doesn’t work!

- With most checksum methods, Mallory can easily change the message in such a way that the checksum stays the same
- We need a “cryptographic” checksum
- It should be hard for Mallory to find a second message with the same checksum as any given one

Checksums are usually very simple so they really only work for accidentally errors, most attackers will just compute the checksum and set it properly,

Hash functions are nice because they don't have any key at all.

Authentication codes (mentioned earlier as MACs) have a shared secret key.

Signature schemes have a private signing algorithm and a public verification algorithm.

Cryptographic hash functions

- A hash function h takes an arbitrary length string x and computes a fixed length string $y = h(x)$ called a message digest
  - Common examples: MD5, SHA-1, SHA-2, SHA-3 (AKA Keccak, from 2012 on)
- Hash functions should have three properties:
  - Preimage-resistance:
    - Given y, it's hard to find x such that $h(x) = y$ (i.e., a "preimage" of x)
  - Second preimage-resistance:
    - Given x, it's hard to find $x' \neq x$ such that $h(x) = h(x')$ (i.e., a "second preimage" of $h(x)$)
  - Collision-resistance:
    - It's hard to find any two distinct values x, x' such that $h(x) = h(x')$ (a "collision")

MD5 is shit, do not use.

SHA is a set of standards published that hash algorithms should follow. Currently there have been no fully breaking attacks on SHA-1 just weakening ones, we want to all be moving to SHA-3.

Preimage resistance you should not be able to find any input that gives this output.

Second preimage resistance you should not be able to find another input that gives this output

Collision resistance you shouldn't be able to find an input or its output

We really want to make is so that hackers cannot find any collisions (when we accomplish this it is a strong hash function).

What is “hard”?

- For SHA-1, for example, it takes 2^{160} work to find a preimage or second preimage, and 2^{80} work to find a collision using a brute-force search
  - However, there are faster ways than brute force to find collisions in SHA-1 or MD5
- Collisions are always easier to find than preimages or second preimages due to the well-known birthday paradox

Preimage resistance has us think of a bunch of inputs and try to get the given output of y (do the same for second preimage). Collision resistance is much easier since you just want to see if any two inputs have matching outputs, so it takes a square root number of times that preimage checks need.

Birthday Paradox The odds of two people sharing a birthday in a room are significantly higher than the odds of someone in the room having the same birthday as me. This only takes roughly 23 people to get 50% probability of two sharing a birthday.

You can weaken these hashes if they preserver character frequencies.

What is “hard”?

- For SHA-1, for example, it takes 2^{160} work to find a preimage or second preimage, and 2^{80} work to find a collision using a brute-force search
  - However, there are faster ways than brute force to find collisions in SHA-1 or MD5
- Collisions are always easier to find than preimages or second preimages due to the well-known birthday paradox

Currently this does not preserve integrity because Mallory knows the hash function so she can just rewrite the message, hash it, and swap thatin.

Cryptographic hash functions

- You can't just send an unencrypted message and its hash to get integrity assurance
  - Even if you don't care about confidentiality!
- Mallory can change the message and just compute the new message digest herself

Cryptographic hash functions

- Hash functions provide integrity guarantees only when there is a secure way of sending and/or storing the message digest
    - For example, Bob can publish a hash of his public key (i.e., a message digest) on his business card
    - Putting the whole key on there would be too big
    - But Alice can download Bob's key from the Internet, hash it herself, and verify that the result matches the message digest on Bob's card
- What if there's no external channel to be had?
    - For example, you're using the Internet to communicate

So hash functions are only good for preserving integrity when they cannot be messed with.

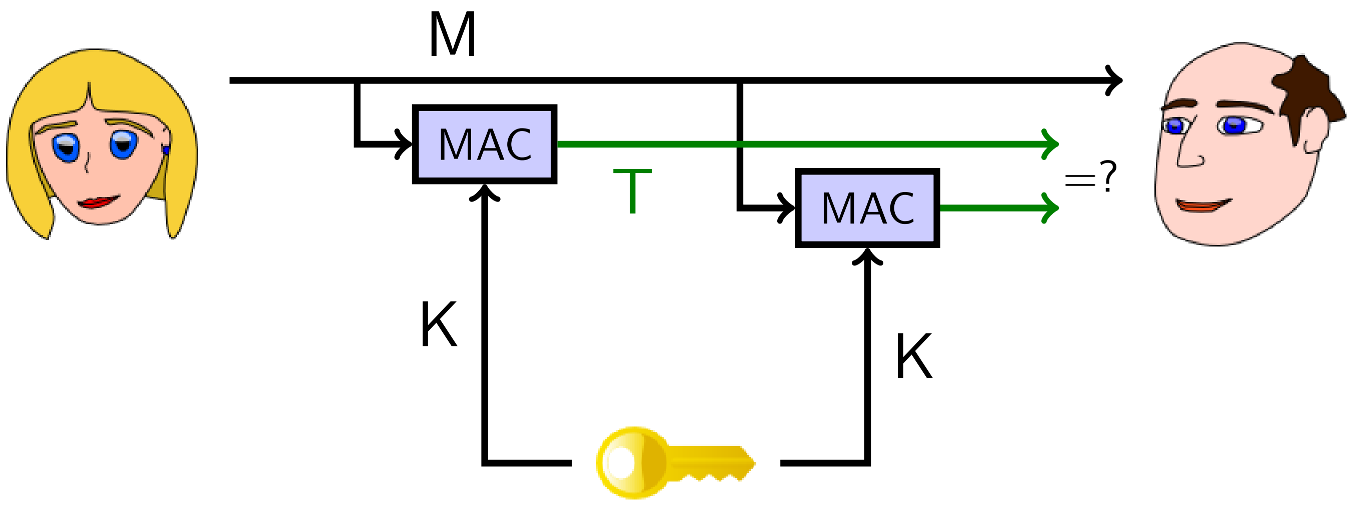
Message authentication codes

- We do the same trick as for encryption: have a large class of hash functions, and use a shared secret key to pick the "correct" one
- Only those who know the secret key can generate, or even check, the computed hash value (sometimes called a tag)
- These "keyed hash functions" are usually called Message Authentication Codes, or MACs
- Common examples:
    - SHA-1-HMAC, SHA-256-HMAC, CBC-MAC

When we encrypt the message we take the last block of cipher text and that becomes the MAC

HMAC returns the hash of (key otimes opad append hash of (key otimes ipad append message)). opad and ipad are fixed public settings.

Message authentication codes



Alice wantes to get message to Bob so she sends it using the shared secret key k. Alice also sends him the MAC of the message, from this Bob calculates the MAC of the message that Alice sent him and checks that this matches the MAC she sent him. Because the MAC calculation requires knowledge of the secret key you can use this to ensure integrity.

Combining ciphers and MACs

- In practice we often need both confidentiality and message integrity
- There are multiple strategies to combine a cipher and a MAC when processing a message
  - Encrypt-then-MAC, MAC-then-Encrypt, Encrypt-and-MAC
- Encrypt-then-MAC is the recommended strategy
- Ideally your crypto library already provides an authenticated encryption mode that securely combines the two operations so you don't have to worry about getting it right
  - E.g., GCM, CCM (used in WPA2, see later), or OCB mode

Encrypt then MAC You can encrypt the message first and then MAC it, return encrypt(message) append MAC(encrypt(message))

MAC then encrypt encrypt(message append MAC(message))

encryot and MAC encrypt(message) append MAC(message)

Encrypt then MAC is the best of the options.

Repudiation

- Suppose Alice and Bob share a MAC key K, and Bob receives a message M along with a valid tag T that was computed using the key K
  - Then Bob can be assured that Alice is the one who sent the message M, and that it hasn't been modified since she sent it!
  - This is like a "signature" on the message
  - But it's not quite the same!
  - Bob can't show M and the tag T to Carol to prove Alice sent the message M

Repudiation

- Alice can just claim that Bob made up the message M, and calculated the tag T himself
- This is called repudiation; and we sometimes want to avoid it
- Some interactions should be repudiable
  - Private conversations
- Some interactions should be non-repudiable
  - Electronic commerce

Digital signatures

- For non-repudiation, what we want is a true digital signature, with the following properties:
- If Bob receives a message with Alice's digital signature on it, then:
  - Alice, and not an impersonator, sent the message (like a MAC)
  - the message has not been altered since it was sent (like a MAC), and
  - Bob can prove these facts to a third party (additional property not satisfied by a MAC).
- How do we arrange this?
  - Use similar techniques to public-key cryptography

Digital signatures are used when we don't want repudiation. It works using a public key system where the key is separated into two parts, a public and a private key.

Making digital signatures

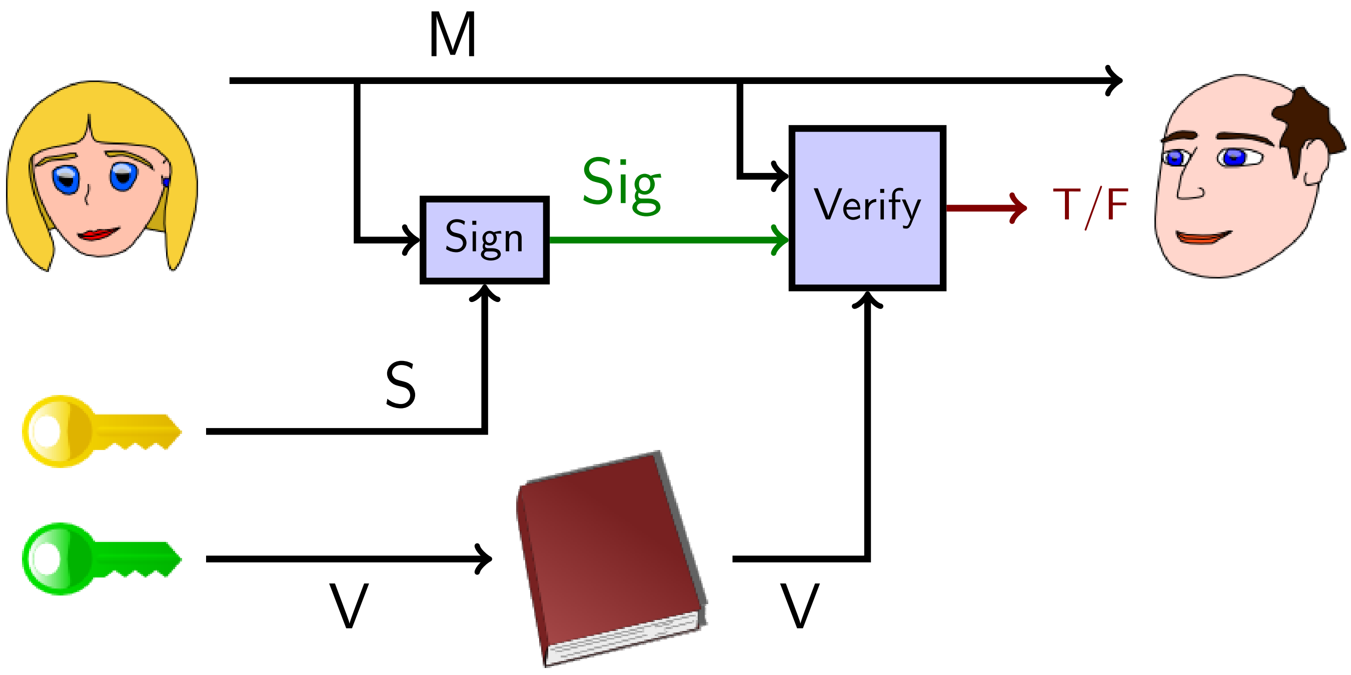
- Remember public-key crypto:
  - Separate keys for encryption and decryption
  - Give everyone a copy of the encryption key
  - The decryption key is private
- To make a digital signature:
  - Alice signs the message with her private signature key
- To verify Alice's signature:
  - Bob verifies the message with his copy of Alice's public verification key
  - If it verifies correctly, the signature is valid

private key = signing

public key = verification

The private key is not for decryption, it is for signing things. So Alice privately signs the document and Bob can use the public key to verify the signature.

Making digital signatures



Alice puts her secret key into some signing algorithm that also takes the message in. It outputs the signature that goes to Bob. Bob then uses the public validation key into a validation algorithm along with the signature which returns true or false if Alice is valid.

$$ab = 1\%(p-1)(q-1)$$
$$S = M^b\%n$$
$$V = S^a == M\%n$$

S = signature, V = is valid, M = message, b = secret key, a = public key, n = private key * public key, p and q are large prime numbers

Its very tempting to think that signing is decryption and verification is encryption because they work similarly, but they are different. There is a tone of nuance that happens in actual signing schemes to make them secure which is what makes it different from en/decryption. The textbook gets this wrong, ignore the textbook.

DSA and ECDSA are the most commonly used signature schemes. DSA is messy to talk about so we are going to look at its precursor El gammal (this let to Schnorr which led to DSA).

p = some large prime

α = generator for random int

$\beta = \alpha^a\%p$

the combination of p, α, and β are the public key and a is the private key.

$$\begin{aligned} Sign_k(M) &= (\gamma, \delta) \\ \gamma &= \alpha^r\%p \\ \delta &= (M - a\gamma)r^{-1}\%(p-1) \\ (\gamma, \delta)- &\to Bob \\ Ver_k(M, \gamma, \delta)) &= \beta^\gamma\gamma^\delta == \alpha^M\%p \end{aligned}$$

Hybrid signatures

- Just like encryption in public-key crypto, signing large messages is slow
- We can also hybridize signatures to make them faster:
  - Alice sends the (unsigned) message, and also a signature on a hash of the message
  - The hash is much smaller than the message, and so it is faster to sign and verify
- Remember that authenticity and confidentiality are separate; if you want both, you need to do both

Math is hard and very costly, so optimization is important.

Alice generates a hash from some hash function on her message. She then creates a signature for that hash and sends the signature and the initial message to Bob. He can then hash the message and use that to verify Alice's signature. We hash things so that the data we are running our signature and verification algorithms on is much shorter to make them much faster. We do not send the hash through to Bob because Mallory could intercept it and change it along the way.

Collisions still suck. A signature will still work for anything that has a hash collision. This means that Alice might be verifying things that she did not mean to.

Combining public-key encryption and digital signatures

- Alice has two different key pairs: an (encryption, decryption) key pair and a (signature, verification) key pair
  - So does Bob
- Alice uses Bob's encryption key to encrypt a message destined for Bob
- She uses her signature key to sign the ciphertext
- Bob uses Alice's verification key to check the signature
- He uses his decryption key to decrypt the ciphertext
- Similarly for reverse direction

The recommended method is to sign then encrypt.

Relationship between key pairs

- Alice's (signature, verification) key pair is long-lived, whereas her (encryption, decryption) key pair is short-lived
  - Gives perfect forward secrecy (see later)
- When creating a new (encryption, decryption) key pair, Alice uses her signing key to sign her new encryption key and Bob uses Alice's verification key to verify the signature on this new key
- If Alice's communication with Bob is interactive, she can use secret-key encryption and does not need an (encryption, decryption) key pair at all (see TLS or SSH)

You should periodically rotate your key pair which will give you perfect forward secrecy (if the fbi takes your hard drive they still can't decrypt everything). Everytime we rotate you publish you encryption key signed so that everyone can get your encryption key and know that it is from you.

The Key Management Problem

- One of the hardest problems of public-key cryptography is that of key management
- How can Bob find Alice's verification key?
  - He can know it personally (manual keying)
    - SSH does this
  - He can trust a friend to tell him (web of trust)
    - PGP does this
  - He can trust some third party to tell him (CA's)
    - TLS / SSL do this

We don’t care if we leak our public key, but when we get a public key we want to make sure that we are getting the correct public key and not from some man in the middle.

Certificate authorities

- A CA is a trusted third party who keeps a directory of people's (and organizations') verification keys
- Alice generates a (signature, verification) key pair, and sends the verification key, as well as a bunch of personal information, both signed with Alice's signature key, to the CA
- The CA ensures that the personal information and Alice's signature are correct
- The CA generates a certificate consisting of Alice's personal information, as well as her verification key. The entire certificate is signed with the CA's signature key

Certificate authorities

- Everyone is assumed to have a copy of the CA's verification key, so they can verify the signature on the certificate
- There can be multiple levels of certificate authorities; level n CA issues certificates for level n+1 CAs
  - Public-key infrastructure (PKI)
- Need to have only verification key of root CA to verify certificate chain

When you generate a key pair you send the public key, signed, to the CA (along with identification information). The CA is supposed to check you id (like meeting in person), but usually they post a file to a domain and check that you get it. There is some higher authority that authorizes CAs (by holding their keys). Its basically a giant tree of trust. There are hundreds of root CAs

Putting it all together

- We have all these blocks; now what?
- Put them together into protocols
- This is HARD. Just because your pieces all work, doesn't mean what you build out of them will; you have to *use* the pieces correctly
- Common mistakes include:
  - Using the same stream cipher key for two messages
  - Assuming encryption also provides integrity
  - Falling for replay attacks or reaction attacks
  - *LOTS* more!

Security controls using cryptography

- Q: In what situations might it be appropriate to use cryptography as a security control?
- A: Remember that there needs to be some separation, since any secrets (like the key) need to be available to the legitimate users but not the adversaries
- In some situations, this may make secret-key crypto problematic
- If your web browser can decrypt its file containing your saved passwords, then an adversary who can read your web browser probably can, too
- Q: How is this solved in practice?

Program and OS security

- Using secret-key crypto can be problematic for the above reason
    - But public-key is OK, if the local machine only needs access to the public part of the key
    - So only encryption and signature verification; no decryption or signing
    - E.g., apps can be installed only if digitally signed by the vendor (BlackBerry) or upgraded only if signed by the original developer (Android)
    - OS may allow execution of programs only if signed (iOS)

Encrypted code

- There is research into processors that will only execute encrypted code
- The processor will decrypt instructions before executing them
- The decryption key is processor-dependent
- Malware won't be able to spread without knowing a processor's key
- Downsides?

Encrypted data

- Harddrive encryption protects data when laptop gets lost/stolen
- It often does not protect data against other users who legitimately use laptop
- Or somebody installing malware on laptop
- Or somebody (maybe physically) extracting the decryption key from the laptop's memory

Some processors have security guard extensions which encrypts your instructions in such a way that only that processor can decrypt. Everytime you download code you have to tell it your processor id so that you can encrypt it. This way if malware gets on your machine it doesn't get run because it isnt encrypted. This is mostly a reasearch idea. Most of these processors do not run everything in encrypted mode.

There is an attack known as the **cold boot attack** which is physically taking the ram from another machine and reading the values on your machine since the keys must be loaded into ram. Use liquid nitrogen on the pins of the machine to cool it down so that the values are retained in memory longer.

OS authentication

- Authentication mechanisms often use cryptography
  - E.g., salted hashes (see Module 3)
- Unfortunately, people are bad at doing cryptography in their heads, so some mechanisms require hardware token

Photo from `http://itc.ua/`

The secure remote password protocol (challenge response) includes encryption to gain authentication. Two factor authentication is very helpful. An authentication token is really helpful here in that they just generate a key based on what time it is with some algorithm. In 2013 some hackers broke into the security company that makes the fobs and got their copy of their key value.

Network security and privacy

- The primary use for cryptography
    - “Separating the security of the medium from the security of the message”
- Entities you can only communicate with over a network are inherently less trustworthy
    - They may not be who they claim to be

Data in transit is the main use for crypto. The data channel has to also be secure.

Network security and privacy

- Network cryptography is used at every layer of the network stack for both security and privacy applications:
  - Link
    - WEP, WPA, WPA2
  - Network
    - VPN, IPsec
  - Transport
    - TLS / SSL, Tor
  - Application
    - ssh, Mixminion, PGP, OTR

Link-layer security controls

- Intended to protect local area networks
- Widespread example: WEP (Wired Equivalent Privacy)
- WEP was intended to enforce three security goals:
  - Confidentiality
    - Prevent an adversary from learning the contents of your wireless traffic
  - Access Control
    - Prevent an adversary from using your wireless infrastructure
  - Data Integrity
- Unfortunately, none of these is actually enforced!

WEP is super shit. Its basically plaintext. We want to use WPA2

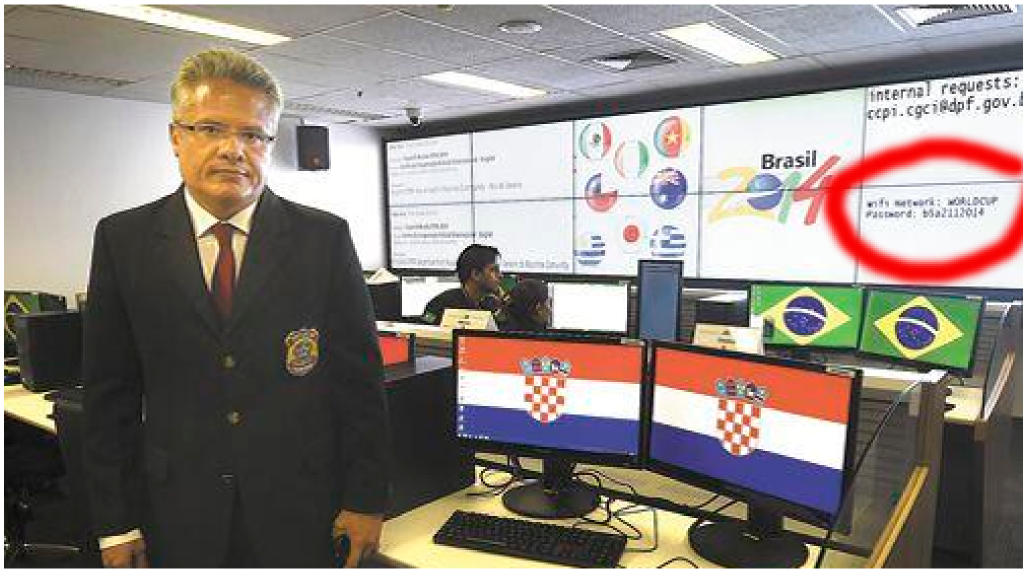
WEP description

- Brief description:
- The sender and receiver share a secret k
    - The secret k is either 40 or 104 bits long
- In order to transmit a message M:
    - Compute a checksum $c(M)$
        - this does not depend on k
    - Pick an IV (a random number) v and generate a keystream $RC4(v, k)$
    - XOR $\langle M, c(M) \rangle$ with the keystream to get the ciphertext
    - Transmit v and the ciphertext over the wireless link

A key that is 64 or 128 bits long and it uses the RC4 stream cipher where we concatonate v and k to form the keystream K. You get the cipher text through the message concated with the cipher of the message otimesed with the key stream.

One problem is that for integrity they are using a checksum where people can compute them easily.

Problem number 0: Widely shared "secrets"



http://www.theregister.co.uk/2014/06/25/brace_yourselves_brazil_dill_in_world_cup_wifi_spill/

In this example you can see the wifi password in the background of this news broadcast.

WEP description

- Upon receipt of v and the ciphertext:
  - Use the received v and the shared k to generate the keystream $RC4(v, k)$
  - XOR the ciphertext with $RC4(v, k)$ to get $\langle M', c' \rangle$
  - Check to see if $c' = c(M')$
  - If it is, accept M' as the message transmitted

- Problem number 1: v is 24 bits long
  - Why is this a problem?

Bob gets the keystream

$$\begin{aligned} K' &= RC4(v \parallel k) \\ M' \parallel C' &= C \otimes K' \\ accept &= c' == c(M') \end{aligned}$$

Assume that v is a counter (so 1, 2, 3, ...) with some max probably equal to max int which loops back to 1 when it overflows. But v is random so we can apply the birthday paradox so the number of samples required to cause a collusion is going to be the square root of int max.

WEP data integrity

- Problem 2: the checksum used in WEP is CRC-32
  - Quite a poor choice; there's already a CRC in the protocol to detect random errors, and a CRC can't help you protect against malicious errors.
- The CRC has two important properties:
  - It is independent of k and v
  - It is linear: $c(M \text{ XOR } D) = c(M) \text{ XOR } c(D)$
- Why is linearity a pessimal property for your integrity mechanism to have when used in conjunction with a stream cipher?

WEP uses the cyclic redundancey check to calculate the checksum which kinda sucks. Checksums are not meant to protect against attackers.

An adversary picks some $M' = M \otimes \delta$ and gets the cipher text $c(M') = c(M' \otimes \delta) = c(M') \otimes c(\delta)$. The attacker wants some $C' = (M' \parallel c(M')) \otimes K$ but they do not know what K is.

Do some math:

$$C' = (M' \parallel c(M')) \otimes K$$
$$C' = (M \otimes \delta \parallel c(M) \otimes x(\delta)) \otimes KC' \qquad = K \otimes (M \otimes \delta \parallel c(M) \otimes x(\delta))C' = K \otimes ((M \parallel c(M)) \otimes (\delta \parallel c(\delta)))$$

We see that $K \otimes ((M \parallel c(M))$ is just the key stream so an attacker can just take the first chunk of a message sent and build her message out of that in such a way that the integrity check still passes.

WEP access control

- What if the adversary wants to inject a new message F onto a WEP-protected network?

- All he needs is a single plaintext/ciphertext pair
- This gives him a value of v and the corresponding keystream $RC4(v, k)$
- Then $C' = \langle F, c(F) \rangle$ XOR $RC4(v, k)$, and he transmits v, C'
- C' is in fact a correct encryption of F, so the message must be accepted

Say the attacker gets the cipher text and the plain text pair, they can use it to calculate the key string.

$$RC4 = C \otimes (M \parallel c(M)) = K$$

Not to send some made up message F we can calculate the checksum because we now have some K to use. WEP allows us to reuse key streams since collisions are common. They do not check for reuse. So once we get a keystream for any message we can use it to encrypt all messages.

WEP authentication protocol

- How did the adversary get that single plaintext/ciphertext pair required for the attack on the previous slide?
    - Problem 3: It turns out the authentication protocol gives it to the adversary for free!

- This is a major disaster in the design!
- The authentication protocol (described on the next slide) is supposed to prove that a certain client knows the shared secret k
- But if I watch you prove it, I can turn around and execute the protocol myself!
    - “What’s the password?”

When you first connect to an access point it sends you a plaintext value M which is a random challenge. You encrypt it and send it back along with some initialization vector v. The attacker just has to get v and they can send as many encrypted messages as they want.

WEP authentication protocol

- Here's the authentication protocol:
  - The access point sends a challenge string to the client
  - The client sends back the challenge, WEP-encrypted with the shared secret k
  - The wireless access point checks if the challenge is correctly encrypted, and if so, accepts the client

- So the adversary has just seen both the plaintext and the ciphertext of the challenge
- Problem number 4: this is enough not only to inject packets (as in the previous attack), but also to execute the authentication protocol himself!

WEP decryption

- Somewhat surprisingly, the ability to modify and inject packets also leads to ways the adversary can decrypt packets!
  - The access point knows k; it turns out the adversary can trick it into decrypting the packet for him and telling him the result.
- Note that none of the attacks so far:
  - Used the fact that the stream cipher was RC4 specifically
  - Recovered k

Recovering a WEP key

- Since 2002, there have been a series of analyses of RC4 in particular
  - Problem number 5: it turns out that when RC4 is used with similar keys, the output keystream has a subtle weakness
    - And this is (often) how WEP uses RC4!
- These observations have led to programs that can recover either a 104-bit or 40-bit WEP key in under 60 seconds, most of the time
  - See the optional reading for more information on this