Cryptography 1 - Stanford - Full

Section 1

- Crypto is everywhere. To secure comms (HTTPS, Bluetooth), to secure files on disk (Truecrypt, EFS), content protection (DVD uses CSS), user authentication... etc
- CSS is very easy to break ;)

Overview comms

- Public key crpyto
- Algorithm has two main parts: handshake protocol, where they establish a shared secret key, then the `record layer`, where they use that shared private key to transmit data.

Overview files on disk

- You want confidentiality and integrity (you want to know if an eavesdropper has modified your files too!)

Symmetric encryption

- First half of the course.

- You have a secret key `k`. You use a `cipher` which has two algorithms, `E` for `encryption` and `D` for `decryption`.

Use cases

- Single use key or a one time key: every key encrypts a single message (used for example to encrypt email)

- Multi use key or many time key: key used to encrypt multiple messages (for example: for encrypted files). This turns out to be more complex, not surprisingly.

What is cryptography?

1. Secret key establishment
2. Secure communication

And also:

- Digital signatures
- Anonymous communication

Example: **digital cash** (I talked about this in Wildlife)!

Theorem (of sorts): anything you can do with a trusted authority, you can do without.

History of crypto

Book: `The codebreakers`

Substitution cipher

Easy to break even though it has `26! = 2**88` key size.

1. Frequency of letters
2. Frequency of pairs of letters

This is called a Ciphertext Only Attack.

Variations are the Caesar Cipher and the Vignere Cipher.

Rotor machine

A motor encodes the substitution table.

Data encryption standard (1974)

- DES

Section 2: Discrete probability

Definition: a **probability distribution** P over U is a function `P: U -> [0,1]` such that `sum(p(x)) = 1, where x exists in U`.

- **Uniform distribution**: for all x in U: P(x) = 1/len(U)

- **Point distribution at x0**: P(x0) =1, for all x not x0, P(x) = 0.

In Cryptography, the universe U is always finite. So we can always just list the output in a **distribution vector**, example: `[P(01), P(00), P(10), P(11)]` would be a distribution vector for an U of two bits.

Definition: an **event** is a subset A included in U; which has a probability `Pr[A] = sum(P(x)) where x in A`.

Example: `U = {0,1}**8`, `A = {all x in U such that the least significant two bits are 11}`, what is the provability of the event A assuming an uniform distribution over U?

```
There are 2**8 = 256 elements in U. How many
of them end with 11?

The Pr[A] in A is 2**6 / 2**8: it is like
choosing a random number of six bits, as the
last two bits are fixed.

But how you express this as a sum of
probabilities?

Well... the probability of getting any number
in U is obviously 1 / 2**8. Now there are
```

```
2**6 elements in A.

Pr[A] = sum(P(x)) where x in A
Pr[A] = sum(1/2**8 * len(A))
Pr[A] = 1/2**8 * 2**6
Pr[A] = 2**6 / 2**8
```

Definition: The **union bound** between A1 and A2 is the probability that any one of A1 and A2 happen. It is fairly reasonable to think that `Pr[A1 + A2] <= Pr[A1] + Pr[A2]`. If the events are disjoint, the probability is exactly the sum.

Example: `U = 2**8, A1=least significant bits is 11, A2=most signifcant bits is 11`

```
The events are not disjoint ;)

So all we can say is that Pr[A1 + A2] <=
Pr[A1] + Pr[A2]
```

Definition: a **random variable** is a `function X: U -> V`.

Example: `X: {0,1}**n; X(y) = lsb(y) (lsb = least significant bit)`

For the uniform distribution on U, `Pr[X=0] = 1/2 ; Pr[X=1] = 1/2`

The important property of a **random variable** is that `X: U -> V` must induce a distribution on V: `Pr[X=v] := Pr[X**-1(v)]`.

```
Note: interestingly, this is not the
intuitive definition of random; as even a
constant X(u) = 1 for all u in U is random.

So formally a random variable is just a map
between U -> V that defined _some_
distribution over V. There's a bit more
formality with real numbers because of
density but who even cares about those.

https://math.stackexchange.com/questions/2701
336/is-a-constant-a-random-variable
```

Definition: `r` is an uniform random variable over U if `Pr[r=a] = 1/len(U) for all a in U; this is written r <- U`.

Exercise : Let r be a uniform random variable on `{0,1}**2`. Define the random variable `X = r1 + r2`.

Then what is `Pr[X=2]`? Hint: `Pr[x=2] = Pr[r=11]`.

```
For X(r) = 2, where r in U; we must have r =
11. That's the hint, and it pretty clear.

Pr[r=11] is exactly 0.25. Too easy due to two
bits universe ;)
```

Definition: a **randomized algorithm** is an algorithm `A, where A(m; r) = y`, the `r` argument is implicit and sampled again every time the algorithm is ran, `r` is of course a **random variable** where `r <- {0,1}**n`.

Interestingly, the **randomized algorithm** is itself a **randomized variable**.

Example: `A(m; k) = E(k, m), y <- A(m)`.

Definition: two events `A, B` are **independent** if `Pr[A AND B] = Pr[A] * Pr[B]`.

The same can be defined over **random variables**: `X,Y` which are random variables are **independent** if `for all a, b in V: Pr[X=a AND Y=b] = Pr[X=a] * Pr[y=b]`.

Definition: `XOR` is an operation over bits where `x ^ y = (x + y) % 2`.

A **very important property of XOR**:

- Y is a random variable over `{0,1}**n`
- X is an indep. random and uniform variable on `{0,1}**n` (independent of Y)

Then `Z := Y ^ X` is *also* an uniform random variable on `{0,1}**n`.

Reading again: if I have a bad random variable Y which is non uniform and I `XOR` it against a nice random variable X, I get a nice random variable Z.

Proof: we'll do the proof for one bit, prolly other by induction?

So say:

- Pr[Y=0] = P0; Pr[Y=1]=P1 (note I can't say anything about these because they are just random, not necessarily uniform)
- Pr[X=0] = 1/2, Pr[X=1]=1/2 (I can do these because they are uniform)

Now, what are Pr[Y AND X]? The variables are independent so I can multiply.

- Pr[Y=0 AND X=0] = P0 * 0.5
- Pr[Y=0 AND X=1] = P0 * 0.5
- Pr[Y=1 AND X=0] = P1 * 0.5
- Pr[Y=1 AND X=1] = P1 * 0.5

Now, what's `Pr[Z=0]`?

```
Pr[Z=0] = Pr[x^y = 0] =

Pr[(x,y) = (0,0) OR (x,y) = (1,1)] =

// note they are disjoint
Pr[(x,y) = (0,0) + (x,y) = (1,1)] =

// i think independence allows you to
// do this...
Pr[(x,y) = (0,0)] + Pr[(x,y) = (1,1)]

// so that leaves
P0/2 + P1/2 =

// an we know P0+P1=1 because it is
// a probability distribution
P0/2 + P1/2 = 1/2

And we now have an uniform distribution :)
```

These theorem is why XOR is so useful in Cryptography!

Definition: the **birthday paradox**. Suppose `r1...rN in U` which are independent and identically distributed random variables.

Then the **birthday paradox** says that if you choose `len(U) ** 1/2` samples then you have more than a half chances that one r equals to other r.

Example: Let `U = {0,1}**128`. Then after sampling `2**64` random messages from U, it is likely two sampled messages will be the same.

Theoretic security

Def: a symmetric cipher is defined over `(K, M, C)` where `(E,D)` is a pair of "efficient" algorithms such that:

- `E: KxM -> C`
- `D: KxC -> M`

And `for every m in M, k in K; D(k, (E(k,m)) = m`

Efficient formally is *run under polynomial time* but practically means *run under concrete time constraints*.

Algorithm `E` is usually randomized. Algorithm `D` must always be deterministic.

One time pad

Designed by Vernam in 1917. It is the first example of a "secure" cipher.

- `M = C = {0,1}**n`

- `K = {0,1}**n` ie: they key is a bit string as long as the message.
- `C:= E(k,m) = k ^ m; D(k,c) = k ^ c`.

Note: you can see see that `E` is a uniform random distribution, assuming k is a uniform random distribution, because of the XOR theorem seen in What is crypto about?

Note: if you have `m` and `E(m, k)` you can easily `m ^ E(m,k) = m ^ k ^ m = k`, so you can recover the key.

Information theoretic security

First person to study this was Shannon in 1949.

The basic idea of Shannon's definition is that the ciphertext should reveal no *information* about the plaintext.

Now what is *information* here?

Def: A cipher `(E,D)` over `(K, M, C)` has **perfect secrecy** if

- `for all m0, m1 in M; len(m0) = len(m1)`
- `for all c in C; Pr[E(k, m0) = c] = Pr[E(k, m1) = c], where k is uniform in K`.

This means that:

- length should not affect encryption
- the probability that a ciphertext C was produced by message m0 or m1 is exactly equal.

You can derive from this, as the attacker learns *nothing* about C, there are no Ciphertext Only Attacks on a perfect cipher (there may be other kind of attacks...).

Lemma: One time pad has perfect secrecy. The proof should be clear from my previous note, the `E` is a uniform distribution.

Another way, longer:

- `for all m, c: Pr[E(k,m) = c] = #len(k in K if E(k,m) = c) / len(k)`

Suppose now that `for all m, c: len(k in K if E(k,m)=c) = T`, where `T` is a constant.

Then if I have that, the `Pr[E(k,m) = c] = T`, because of our supposition.

In the case of the OTP, we have that `len(k in K if E(k,m) = c) = 1`, as every message/ciphertext pair corresponds to exactly one key.

Then `Pr[E(k,m) = c] = 1 / len(k)`. Which is exactly the definition of a random uniform variable :).

Note: perfect secrecy -> a ciphertext/message pair match exactly one key.

Note: easy to see here also that if I want perfect secrecy, I *need* to have a key at least as long as the message, because if not collision will inevitably happen.

The problem with the one time pad and any other perfect cipher: you need a key as long as the message... just pass the message if you have such a secure channel as to pass the key securely.

Stream ciphers and Pseudo Random Generators

Stream ciphers

Idea: replace `random` key by a `pseudorandom` key with a *pseudo-random generator* or PRG for short.

How do we use a PRG to make One time pad practical?

We use the PRG to create bigger keys, basically. So my new improved OTP `C:= E=(k,m) = m ^ G(k); D(k,c) = c ^ G(k)`.

So why is this secure? Of course it does not have perfect secrecy (see Stream Ciphers > Information theoretic security). So let's redefine security, and of course it will depend on the PRG.

The PRG must be not **predictable**.

Definition: a PRG is **predictable** if there exists a `i` such that `G(k)[0..i]`, then I can compute `G(k)[i..n]`. So if given the beginning of a the number produced I can compute the rest.

This would of course be quite bad, as recovering only a part of the key or the ciphertext would leak the whole message!

Because if I have `C[0..i]` and I know that `C[0..i]` is a constant `prefix` (eg: every email begins with a `;`).

Then you have `C[0..i] ^ m[0..i] = G(k)[0..i]` and I can now predict the rest of the `G(k)`, which allows me to compute the rest of the key, which allows me to compute the rest of the message. Even if I can predict a single bit (so from `G(k)[i]` I can predict `G(k)[i+1]`) I can repeat this and decrypt the whole message.

So `G` is **predictable** if there exists an efficient algorithm `A` such that `there exists 1 <= i <- n - 1, such that Pr[A(G(k)[1..i]) = G(k)[i+1] >= 1/2 + epsilon`, where epsilon is non negligible (maybe `epsilon > 1/2**30?`).

So `G` must not be **predictable** if we want to use with OTP.

Example: suppose `G: K -> {0,1}**n` such that for `for all k, XOR(G(K)) = 1` (XOR the bits of the output). Is `G` predictable?

Answer: If for all k `XOR(G(k)) = 1`, that means that I intercept n-1 bits, I can surely predict the next one. But I need n-1...

Negligible and non-negligible values

- In practice: `epsilon` is a scalar and `e >= 1/2**30` is generally considered non-negligible, because it is likely to happen over 1GB of data. `e <= 1/2**80` is quite non-negligible. But these definitions are quite problematic.
- In theory: `epsilon` is a function `e: Z+ -> R+` and `epsilon` is non-neg if `exists d: e(a) >= 1/a**d infinitely often`, and `epsilon` is negligible if `for all d, a>ad: e(a) < 1/a**d`. So basically: if I can find a point at which the `e` function is bound then it is negligible.

OK, this is complicated, but again: `epsilon` is a function which is negligible if my function is smaller than all polynomials: `for all d, a>=ad: e(a) <= 1/a**d`. So for all d ever and for all a after a point, e(a) is smaller than that polynomial. This should remind you a lot of the definition of a limit, and it basically says that `epsilon` is

neg if I can sandwich it between `1/a**d` and zero for all d, which is the same as saying as the limit when a approaches zero exists and is zero.

Examples:

- `e(a) = 1/2**a`: this is clearly negligible, as for any polynomial I can find an A which would be smaller!
- `e(a) = 1/a**1000`: not negligible! because there exists a `d` big enough (say 10000), then `1/a**10000 <= 1/a**1000` for all a!

Attacks on Stream Ciphers

Two time pad is insecure

(!). If you use the same key twice, you are screwed. Easy to see:

- `C1 := m1 ^ PRG(k)`
- `C2 := m2 ^ PRG(k)`
- Then: `C1 ^ C2 = m1 ^ m2`

And then (assuming at least English, ASCII and I suppose most natural languages), if you have the XOR of two messages, you can recover both.

US Intelligence actually screwed up the Russians like this on the cold war between 1941-1946. The project was called Project Verona.

Similarly, MS-PPTP used in Windows NT commited a similar mistakes: the client and server used the same key, so the set of messages from the client and server used the same key. So you could recover the messages. An interesting lesson here: when you have a shared secret key, you actually have one, one for each direction.

Another one: on WEP. Read its page!

Another example, this time with **disk encryption**. Same principle: maybe a file starts with `To Bob` and another one with `To Eve`, but they are identical. Reusing the key will show that that is the only difference. This leaks *where* information changed. And then you can just recover the messages xoring the identical parts of the files.

Take away: **Never use stream cipher keys more than once**. For network traffic, a new key is used for every session. For disk encryption we just don't use stream ciphers ;)

One time pad provides *no integrity* (aka: OTP is malleable)

So you have `m` and `ENC(m, k) = m ^ k`. An attacker now takes the ciphertext and XORs it with a chosen value `p`, such that now `C = m ^ k ^ p`. What will happen when I try to decrypt `C`?

Well you can easily see that `D(C) = m ^ k ^ p ^ k = m ^ p`.

And that is impossible to detect and the modifications are very predictable.

Example: `m = "From: Bob"`. An attacker *knows* the message is from Bob. And he wants to say that the message is from Eve.

So he grabs the `Bob ^ P`, where `P` is whatever results in `Eve`. And that works!

Real world stream ciphers

RC4

An old example from 1987 which should not be used anymore, but nevertheless still is.

It has a variable size seed (let's say 128 bits as example). It expands it to 2048 bits and sets it as the internal state. Then you can run it and it will generate 1 byte per round.

It is used in HTTPS and WEP.

Weaknesses:

- The initial output is biased. Pr[2nd byte = 0] = 2/256 instead of 1/256 (!). Think: this means that it is

twice as likely that the second byte of your plaintext is xored with zero, which means it is not actually encrypted at all! Actually, now we know more bytes are biased like this, so the recommendation is to just discard the first 256 bytes...

- Prob of (0,0) is `1/256**2 + 1/256**3`, instead of `1/256**2`. This bias starts after several gigabytes of data.
- Suffers from related key attacks.

CSS

Badly broken. Used to encrypt DVDs.

CSS is based on a Linear Feedback Shift Register.

The key is 40 bits (5 bytes). Short enough...

Fun fact: This limit is because of encryption exportation restriction on the US, which were limited to algorithms that worked on 5 bytes keys or less. And the creators of DVDs obviously wanted to export those ;)

The design is as follows: there are two Linear Feedback Shift Register, one of 17 bits and another of 25 bits.

The first LFSR is seeded as follows: `1 || first two bytes of the key"` and the second one is `1 || last three bytes of the key`.

Each LSFR runs for 8 cycles, so each generates 8 bits of output. Each output is summed together and to the carry from the previous block, modulo 256. This ouptuts exactly 1 byte per round, which is `XOR`ed against the byte of the movie being encrypted.

This is very easy to break on `2**17` time.

Let's say you have an encrypted movie. But DVDs uses MPEG, you know a prefix (let's say 20 bytes). If you `XOR` this known prefix with your encrypted movie, you now have the first 20 bytes of the output of CSS.

So now, you generate 20 bytes of output with the 17-bit LSFR with the `2**17` possible seeds.

Now you have each of the 20 bytes of outputs possible with the first LSFR *and* you have the first 20 bytes of the output. You subtract them!

The subtraction should be the first twenty byte sequence of the *second* LSFR. If it came from the second LSFR, we have a candidate. Until we hit the right initial state for the 17-bit LSFR and a 25-bit LSFR. And then we learn the correct initial state of both LSFR. And then we can predict the remaining output of the whole system.

Salsa20

Output of the `eStream project`, A project that stopped in 2008 but was dedicated to produce good stream ciphers.

They apparently defined a *nonce*;

A `PRG` such that `PRG: {0,1}**s x R -> {0,1}**n`, where `R` is a nonce.

`E(k, m; r) = m ^ PRG(k, r)`. And `(k,r)` pair is only used once!

And they created **Salsa20**, which is thought both for software and hardware.

`Salsa20: {0,1}**(128 OR 256) x {0,1}**64 -> {0,1}**n`.

`Salsa20(k, r) := H(k, (r, 0)) || H(k, (r, 1)) || ...`

So how does `H` work anyway? You start with a 64 byte state:

```
| T0 | K   | T1 | r  | i  | T2 | K  | T3|
| 4b | 16b | 4b | 8b | 8b | 4b | 16b| 4b|
```

Where:

- `i` is the counter you can see in the definition that you see above
- `4b` and similar mean `4 bytes`
- `T{0,1,2,3}` are constants.

Anyway, you apply a function `h` (little h) to this states, `h: {0,1}**64 -> {0,1}**64`, invertible, designed to be fast on x86.

This `h` function is applied 10 times to the state and then you add 4-bytes at a time the input with the last state. And that's `H` and you get a 64-byte output, and you rotate the `i` as many times as needed.

PRG Security Definitions

Let `G:K -> {0,1}**n` be a PRG.

Goal: define what it means that `[k <- K, output G(K)]` is indistinguishable from `[r <- {0,1}**n, output r]`; remember `<-` means *choosing uniformly over the set*.

And this should be surprising, as `{0,1}**n` is way way bigger than the possible output space for a `G(k)`, which takes as input a really small seed.

Statistical tests

Definition: a **statistical test** on `{0,1}**n` `A` is an algorithm such that `A(x) = 0 OR 1`, `0` means *not random* and `1` means *random*.

For example:

1. `A(x) = 1 iff #0(x) - #1(x) <= 10 * sqrt(n)`, ie: there are roughly the same amount of zeroes and ones.
2. `A(x) = 1 iff #00(x) - n/4 <= 10 * sqrt(n)`, ie: there is roughly the expected amount of double zeroes.
3. `A(x) = 1 iff the max-run-of-0(x) <= 10 * log2(n)`, ie: the longest string of zeroes should be around log2(n).

Def: How do we know if a statistical test is good or not? We use **advantage**: `Adv_PRG[A, G] := | Pr[A(G(k)) = 1] - Pr[A(r) = 1 |`, where `A` is our statistical test, `r` is an actual random number taken, `G(k)` is the output of our `PRG` when given a seed from the specific `PRG`. So what this says is that the advantage is how likely is our test at distinguishing between actual random and pseudo random.

If the advantage is close to zero, the test is pretty bad, as it can't distinguish between random and pseudorandom; while an advantage close to one can.

Example: If my `A` outputs constantly zero, then my advantage is zero.

Less silly example: Let `G: K -> {0,1}**n` which satisfies that `msb(G(k)) = 1 for 2/3 of keys in K` and a statistical test `A` such that `if [msb(x) = 1 ] then 1 else 0`. What is the advantage?

Answer: `Pr[A(G(k))]` is, by definition, `2/3`, as my generator is defined as producing msbs of 1 for 2/3 of keys, while my test is defined as passing when that's true. But for a truly random generator, that probability is 1/2. So `Adv_PRG[A, G] = 2/3 - 1/2 = 4/6 - 3/6 = 1/6`.

Secure PRGs

Definition: We say that `G: K -> {0,1}**n` is a **secure PRG** if for every efficient statistical test A, `Adv_PRG(A, G)` is negligible.

Fun fact: if we leave out the efficient part, then this can't be sattisfied.

Fun fact: It is not known if there are provably secure PRGs. If you can show than `P == NP`, then you can easily prove that there are none. If you can prove that a PRG is secure, you'd be proving that `P != NP`

Show: we can show that if a `PRG` is predictable, then the `PRG` is insecure.

Proof: by contradiction.

- Suppose `A` is an efficient algorithm such that `Pr[A(G(k)[0..i]) = G(k)[i+1] ] = 1/2 + epislon` for a non negligible `epsilon`.
- Then we have an statistical test `B` such that `B(X) = 1 if A[x[0..i] = x[i+1], else 0`.
- Now we will give `B` a truly random string. For truly random, `A` won't be able to do shit, so the probability of B outputing 1 is just `1/2`. If we give `B` a string generated by `G`, then by definition it will be able to guess `1/2 + epsilon`.
- So the advantage is now >= epsilon, which shows that `A` is a good predictor. So `G` is not secure.

The converse is also true. If the PRG `G` is unpredictable at pos. i, whichever the i, then G is a secure PRG.

Computationally indistinguishable

Let `P1` and `P2` distributions over `{0,1}**n`.

Definition: we say `P1` and `P2` are computationally indistinguishable (denoted P1 ~p P2), then if for all efficient stat. tests A, `Pr[A(p1)=1 - Pr(A(p2) = 1| < negligible`.

With this new notation, we can now define a secure PRG more succinctly: a PRG is secure if `{k -> K: G(k)} ~p uniform({0,1}**n)`

Cipher Security Definitions

For now, consider that the attacker's capabilities are to obtain the ciphertext.

Let's think about some possible requirements:

Attacker cannot recover secret key

Wrong. `E(k,m) = m` makes recovering the secret key impossible and is obviously insecure.

Attacker cannot recover all of plaintext

Wrong. `E(k, m0|m1) = m0 | E'(k, m1)`. Not really, because `E'` may be good, and you accomplished your definition, but you leaked half of the plaintext,

Ciphertext should not reveal information about the plaintext

That's a good idea. Is Shannon idea of perfect secrecy again :). But this definition is too strong, as we saw it requires that the key be at least of length message. See Stream Ciphers > Information theoretic security.

Every two ciphertexts should be computationally indistinguishable from each other

Say `(E,D)` has perfect secrecy if `for all m0, m1 in M of equal length, then {E(k, m0) ~p {E(k, m1)}`.

Seems good, but it still too strong. To make it realistic, we need to loosen the `for all` part, and let is say:

Given two ciphertexts they should be computationally indistinguishable from each other

Say `(E,D)` is secure if `given m0, m1 in M of equal length, then {E(k, m0) ~p {E(k, m1)}`.

And this is called **semantic security** for a one-time key.

Semantic security

Definition: experiments `EXP(0)` and `EXP(1)` for `b=0,1`. On experiments, you have a `challenger` and an `adversary`.

1. Challenger picks random key.
2. Adversary picks m0, m1.
3. Challenger outputs `E(k, m0)` *or* `E(k, m1)`
4. Adversary must detect if given the encryption of `m0` or `m1`.

In `EXP(0)` the challenger outputs `m0` and in `EXP(1)` the challenger output `m1`.

Now we have the event `Wb` such that `Wb := [event that EXP(b) = 1]`. So `W0` means adversary's output is `0`, and `W1` means its output is `W1`.

Now we can have the advantage of the adversary: `Adv_Adversary[A, E] : | Pr[W0] - Pr[W1] |`. So if the advantage is big, means the adversary could distinguish clearly when the experiment zero and experiment one where presented to them.

Definition: `E` is **semantically secure** if for all "efficient" `A`, `Adv_Adversary[A, E]` is negligible.

Stream ciphers are semantically secure

We want to prove that if `G: K->{0,1}**n` is a secure RPG, then stream cipher E derived from G is semantically secure.

Strategy: For every sem. sec. adversary A, there exists a PRG adversary B, such that: `Adv_Adversary[A,E] <= 2 * Adv_PRG[B,G]`. This inequality is useful because the right hand side is negligible (per definition, as G is secure). So the left hand side **must** be negligible too. So we proved that `E` is sem. secure.

So let `A` be a sem. sec. adversary (see Stream Ciphers > Semantic security). Now the `challenger`, besides choosing `k`, it will choose `r` a truly random string, and use it as key. The generator is secure, so they are indistinguishable. Now we are encrypting using a truly random string, so this is equivalent to a Stream Ciphers > One time pad.

The advantage of `A`, our adversary, cannot have changed, as they can't distinguish them. But we know that the advantage of `A` with an OTP is zero. So, the advantage of `A` is zero with the PRG too.

What are block ciphers?

- Takes `n` bits of input and outputs `n` bits of output.
- Canonical examples: `3DES` (`n=64 bits, k=168bits`) and AES (`n=128 bits, k=128,192,256 bits`).

Block ciphers are generally built by iteration. The process starts with **key expansion**, which generates a key for every round. A round is a function `R(k_i, m)`, where `k_i` is one of these round keys.

Block ciphers are generally slower than Stream Ciphers, but we can do things that we couldn't with the stream ciphers.

A good abstraction for block ciphers is a Pseudo Random Function. Read that!

A **pseudo random permutation** is such a good abstraction for a block cipher that the terms are many times used interchangeably.

Secure PRFs

Let `F: K x Y -> Y` be a PRF.

Define: `Funs[X,Y]` the set of **all** function from X to Y. The size of this set is `|Y|**|X|`.

Define: `Sf = { F(K, x) / k if fixed and exists in K }`, note this set is inside `Funs[X,Y]`. The size of this set is `|K|`

Intuition: a Pseudo Random Function is secure if:

- a random function in `Funs[X,Y]` is indistinguishable from a random function in `Sf`

Application of PRF: PRG

You can generate a good Pseudo-Random Generator from a good Pseudo Random function.

Let `F: K x {0,1}**n -> {0,1}**n` be a secure PRF. Let us define a `G: K -> {0,1}**nt` (read: `t` blocks of `n` bits). And we can now define **counter mode**:

`G(k) = F(k,0) || F(k,1) || ... || F(k,t)`

Note: this is parallelizable, so we basically have a stream cipher (aka: pseudo-random generator) which is parallelizable.

Why is this secure? **Intuitively**: `F(k, m)` for a fixed K is indistinguishable from a random function `R(m)`.

DES

The `Data Encryption Standard`, standarized in 1973.

Born out of `Luciler`, algorithm by Horst Feistel at IBM, with a key len of 128 bits and a block len of 128 bits. Interestingly, the standard DES downgraded the key size and block dize from `Luciler`: `DES` only has 56-bits keys and 64-bits blocks.

By 1997, `DES` was broken by exhaustive search (you can just brute force the `2**56` keys)

Feistel Network

The core idea behind Block ciphers > DES. AKA Luby-Rackoff Construction

Given functions `f1, ..., fd: {0,1}**n -> {0,1}**n`, the objective is to build an invertible function `F: {0,1}**2n -> {0,1}**2n`.

We will construct `F`. The input of `F` is divided in two: `Ri` and `Li`, where each `i` will be a round, there are `d` rounds.

In each round:

- `Li = Ri-1`
- `Ri = Fi(Ri-1) ^ Li-1`

Amazing claim: no matter what `f1,...,fd` you choose, the Feistel network of `F: {0,1}**2n -> {0,1}2**n` is invertible.

Let's construct the inverse to prove it.

Given `Ri+1, Li+1`, construct `Ri, Li`.

- `Ri = Li+1`
- `Li = Fi+1(Li+1) ^ Ri+1`

Interestingly: the inverse is basically the same circuit, with `f1,...,fd` in reverse order.

Feistel networks are used in many block ciphers, but not in AES.

Theorem: let `f: K x {0,1}**n -> {0,1)**n` be a secure Pseudo Random Function, a 3-round feistel network `F: K3 x {0,1}**2n -> {0,1}**2n` is a secure pseudo random permutation.

How does DES work?

`DES` is a 16-round Feistel network.

- `f1...f16: {0,1}**32 -> {0,1}32`
- `fi(x) = F(ki, x)`, `ki` is derived from `K`, `F` is a function, `fi` is just `F` applied to a different key. See Block ciphers > The F function.

Now, DES algorithms does:

1. Take a 64 bit input
2. Do an `initial permutation`. Not for security reason, it's just there in the standard...
3. Then 16-round Feistel network.
4. Undo the `initial permutation` with a `final permutation`.
5. Spit 64 bit output.

The key expansion expands `K` into the 16 different round keys.

To decrypt, just use the keys in reverse order.

The F function

1. Take `x`, a 32 bit input, and `Ki`, a 48-bit round key.
2. Take `x` through a `expansion box E`, which replicates some of the bits of `x` into 48 bits.

3. Compute the `XOR` of the output of `E(x)` with `Ki`
4. Take the 48-bit `XOR` and separate it into 8 groups of 6 bits, which are called `S Boxes`
5. `S Boxes` map 6 bits into 4 bits, so the output of the boxes is 32 bits.
6. Then perform a permutation on that output; which is the final ouptut of the `F` function.

The `S Boxes` are lookup tables: each 6 bit input is mapped to a 4-bit ouput. Just like that. OK maybe not just like that...

Designing S boxes

To see what a good `S Box` is, let's first think a bad S-box.

Suppose `Si(x1, x2,...,x6) = (x2 ^ x3, x1 ^ x4 ^ x5, x1 ^ x6, x2 ^ x3 ^ x6)`, or, equivalently:

```
Si(x) = Ai . x (mod 2)`, where . is the
vector product

Si(x) = Ai.x (mod 2) =
        0 1 1 0 0 0   x1
        1 0 0 1 1 0 . x2
        1 0 0 0 0 1   x3
```

```
        0 1 1 0 0 1   x4
                  x5
```

Well, then, `Si` is a linear function. It just apply a matrix to an input vector.

Claim: If `S-Box` are linear, `DES` is broken. Because if that's all `S-Box` does is shuffle bits around and `XOR` bits.

As a result, all of `DES` would be a linear function, which would mean I can write it like this:

```
DES(k,m) = B . m   = c
               k1
                          ..
                          k16
```

Where `B` is a matrix of `832 x 64` (64 bit rows and 832 bit columns = 64 bit message + 16 * x 48 keys = 832).

But then: `DES(k, m1) ^ DES(k, m2) ^ DES(k, m3) = DES(k, m1 ^ m2 ^ m3)` (because linear functions!). And I can test this! So I could differentiate easily `DES` from a random function.

If you can recover 832 messages, you could recover the whole secret key.

Fun fact: if you choose the `S-Box` at random, you get something very close to linear, which would also be bad.

Exhaustive Search for block cipher key

Goal: given a few input/output pairs, `(mi, ci = E(k, mi)), i=1..3`, find key K.

We will show that just one pair is enough to constrain a DES key.

Lemma: Suppose DES is an ideal cipher, ie: `2**56` random invertible functions. Then, for every `m, c` there is at most **one** key such that `c = DES(k, m)` with a very high probability (`1-1/256 = 99.5%`)

Proof: What is `Pr[exists k' != k: c := DES(k,m) = DES(k', m)`? Using the `Union Bound.` (see What is crypto about? > Section 2 Discrete probability), the `union bound` is the fact that `Pr[A1 + A2] <= Pr[A1] + Pr[A2]`.

Anyway, using the union bound, you can bound the probability.

```
Pr[exists k' != k: c := DES(k, m) = DES(k', m)` <=
```

```
SUM(Pr[DES(k,m) = DES(k', m)], for every k)
<=
(now note that there are 2**64 possible
outputs, so the likelyhood of DES(k,m) =
DES(k'm) is at mosts 2**64, and we are
summing over 2**56 keys, so...)
2**56 / 2**64 =
1/2**8 =
1/256

So the probability that key is NOT unique is
1/256, and the probability that key is unique
is 255/256 = ~99.%, lemma proven.
```

Now, doing the same analysis with actual `DES` without assuming it is in fact a set of random functions, we will see that given `m1, c1 = DES(k, m1)` and `(m2, c2=DES(k,m2)`, the probability pf `k` being the same is `1/1-2**71` (very high, for AES is `1-1/2**128`, extremely likely!)

DES Challenge

Published by RSA. The challenge is, given:

```
message = "The unknown message is: XXXXX"
CT      = c1 | c2 | c3 | c4 | c5 | c6
```

Now that the first three cipher blocks are gonna be the hardcoded string.

Find `k in {0,1}**56` such that `DES(k, mi) = ci for i=1,2,3`

- 1997: internet search broke it in 3 months
- 1998: EFF machine built special machine, broke it in 3 days.
- 1999: combined the two, broke in 22 hours.
- 2006: with relatively cheap hardware (USD 10K) broke in 7 days.

So... don't use 56 bit keys!

The solution? Iterate the cipher.

Strengthening DES against exhaustive search

Triple-DES

Define `3E: K**3 x M -> M` as `3E((k1,k2,k3), m) = E(k1, E(k2, E(k3, m))`.

This key is `168 bits` now, which is way too big to do exhaustive search. This solution makes it 3x slower than normal DES.

There's still an attack that breaks DES in `2**118` instead of `2**168 bits`, but this is still secure. Anything above

`2**90` is considered secure.

Why not Double DES?

Should have `112` bits of key-len and that should be enough. But...

```
If double DES is this:
2E ((k1,k2), m) = E(k1, E(k2, m))

And given C I want to find (k1, k2)...
E(k1, E(k2, m)) = C (apply D(k1) to both
sides)
E(k2, m) = D(k1, c)

And now I can attack each independent side...
```

This kind of attack is called a Meet in the middle attack, and it runs on a total time of `2**63`, which is way smaller than the theoretical attack of `2**112`.

The same attack is possible against Triple DES! That takes `2**118`.

DESX

A non-standarized method to strengthen DES.

`EX = EX((k1,k2,k3), m) = k1 ^ E(k2, m ^ k3)`

The key length is `64 + 56 + 64 = 184 bits`, but there's an attack which takes `2**120`.

Attacks on block ciphers

On the implementation

1. Measure time do enc/dec or measure power used to enc/dec (Side Channel Attacks).
2. Fault attack: force system to fault, exposing errors in the last round of enc, which leaks the key.

Linear and differential attacks

Given **many** input, output pairs, we can recover the key in time less than `2**56` (applied to DES of course).

Linear cryptoanalysis: let `c = DES(k,m)`, suppose for random `k,m`:

```
Pr[(m[i1] ^ ... m[ir]) ^ (c[jj] ^ ... ^
c[jv]) =
   (k[l1] ^ ... ^ k[lu])] = 1/2 + epsilon
```

Read that as: subsets of the message and the ciphertext are not realted to the key more than the random function would relate them.

For `DES` specifically, `epsilon = 1/2**21`, due to a bug in the fifth `S-Box`.

Quantum attacks

Magic, whatever. In quantum, if you have `f: X -> {0,1}`, large `X` and you want to find the input that produces 1 (most of them are zero).

On classical computers, of course, you have to basically try: `O(len(X))`. But if you build a quantum computer, you have `O(sqrt(len(X))`. The thing is that we don't know if can build these quantum algorithms... This would absolutely break `DES` and `AES-128`.

AES

- In 1997 NIST publishes the request for proposal
- In 2000, Rijndael was chosen.

It has a block size of 128 bits and the key sizes are 128 bits, 192

It **does not** use a Block ciphers > Feistel Network, instead it is built a **Substitution Permutation Network**: every bit is changed in every round.

The algorithm works like this: `xor` the input the first round key, we substitute the `xor`, and then we permutate. And repeat so for `n` rounds.

Note that everything must be reversible to decrypt! And decryption is just that, applying the steps in reverse order.

The functions between the XOR are:

1. ByteSub: substitution box
2. ShiftRow: permutates, cyclic shift on each row
3. MixColumn: a linear transformation is applied to each column

With 10 rounds, although in the last round we don't do MixColumn.

Both Intel and AMD have hardware support for AES. In Intel:

- `aesenc`: one round of AES, 128 need two 128-bit registers, one for the state another for the round key.
- `aesenclast`: the last round of AES.

Only two attacks:

- Best key recovery attack: only four times better than ex. search... so 128 bits AES is still 126 bits...
- Given `2**99` input/output pairs from four related keys in `AES-256` (related keys: similar) can recover keys in time `~2**99`. It is a very good attack (from

`AES-256 -> 99`), but you need `2**99` input/output pairs... but if you repeat keys this may be you.

Block ciphers from a PRG

PRF from PRG

Let `G: K -> K**2` be a secure PRG.

Define a `1-bit PRF F: K x {0,1} -> K`, as `F(k, x) = G(k)[x]`, where `x = 0` or `x = 1`

Can we build a PRF with a larger domain?.

Extending a PRG

Let `G:K -> K**2`: it duplicates its input. Let's see if I can build `G1` that quadruples its input!

Define: `G1: K -> K4` as `G1(k) = G(G(k)[0] || G(G(k)[1])` ... and we now have a two bit PRF.

Proof G1 is secure: is quite intuitive that if `G` was secure, the concatenation of their output is unpredictable or indistinguishable from random.

Now repeat this for ever and you can have `Gn:K -> K**n`. Heck you can even truncate the output and have it output in `Kn`, where `Kn` is a set of size `n`.

This is called the GGM PRF.

This is not used in practice? `F(k, m)` because it is slow.

Review: PRP and PRF

Difference between PRG and PRF

`Pseudorandom functions are not to be confused with [pseudorandom generators](https://en.wikipedia.org/wiki/Pseudorandom_generator "Pseudorandom generator") (PRGs). The guarantee of a PRG is that a _single_ output appears [random](https://en.wikipedia.org/wiki/Random "Random") if the input was chosen at random. On the other hand, the guarantee of a PRF is that _all its outputs_ appear random, regardless of how the corresponding inputs were chosen, as long as the _function_ was drawn at random from the PRF family.`

From Wikipedia, not actually part of the course but I needed the clarification.

PRFs and PRFs

- **PRF** over `(K, X, Y) / F: K x X -> Y, there exists an efficient algorithm to evaluate F(k,x)`

- **PRP** over `(K,X) / E: K x X -> X`, such that (1) exists efficient *deterministic* algorithm to evaluate `E(k, x)`, (2) `E(k, x')` is one to one for a fixed `x'` and all `k`, exists an efficient inversion algorithm `D(k,x)`.

Defining secure PRFs

Remeber `Funs[X,Y]` is all the functions from `X` to `Y`, and `Sf = { F(k, x') for a fixed x' and k in K }` `is in Funs [X, Y]`.

Intuition: a PRF is secure if a random choice of `Funs[X,Y]` cannot distinguish from a random function in `Sf`.

Rigurous: with experiments. For `b=0,1` define `EXP(b)`. `Adv. A` must distinguish between a truly random function in `Funs[X,Y]` and a function in `Sf`; we call the output `b'` and is the output of the experiment, so `EXP(b) = b'` . The adversary sends inputs and the challenger can choose either `b=0` to indicate `f in Sf` or `b=1` to indicate `f in Funs[X,Y]`.

Then we say that `F` is a secure PRF if for all efficient adversaries, `Adv_PRF[A,F] := |Pr[Exp(0)=1] - Pr[Exp(1)=1]|` is negligible.

Defining secure PRP

Very similar to Block ciphers > Defining secure PRFs, but the challenger is gonna choose between a function in `PRP` and a truly random permutation in `X`:

```
Adv_PRP[A,E] = |Pr[EXP(0)=1] - Pr[Exp(1)=1]|
```

Not all secure PRPs are secure PRFs

Example: `E(k,x) = x ^ k` for `X, K={0,1}`.

This is a secure `PRP`, as an attacker cannot distinguish between the two invertible functions (permutations) in this set `(0,1) -> (0,1)` or `(0,1) -> (1,0)`.

But note that this is not a secure `PRF`, because there are four functions in `(0,1)`:

1. `(0, 1) -> (0, 1)` (identity)
2. `(0,1) -> (1, 0)` (invert)
3. `(0,1) -> (1,1)` (constant 1)
4. `(0,1) -> (0,0)` (constant 0)

   Now an attacker can evaluate `(k,0)` and `(k,1)`, and if the output matches we are interacting with the actual random function, as our `PRF` with `XOR`

could not produce matching outputs on this; because:

- If `k=1`, then `1 ^ 0 = 1` and `1 ^ 1 = 0`.
- If `k=1`, then `1 ^ 1 = 0` and `1 ^ 0 = 1`.

This is true only because the set is too small...

PRF Switching lemma

PRF Switching lemma: Any secure PRP is also a secure PRF, if `|X|` is sufficiently large! Let `E` be a `PRP` over `(K,X)`, then for any `q-query` of adversary A, then:

`|Adv_PRF[A,E] - AdvPRP[A,E]| < q**2 / 2*|X|`.

If |X| is sufficiently large, then `q**2 / 2*|X|` is negligible.

Modes of operation: One Time Key

Goal: build a secure encryption from a secure PRP (eg: AES).

Adversary can only see **one** ciphertext encrypted under a one-time key. Their goal is to learn info about the plain text.

ECB

ECB mode is tremendously broken, just skip it.

Deterministic counter mode

Let's have a secure `PRF F` (eg: AES), and we will do:

`Edetctr(k, m) = m[0] XOR F(k, 0) | m[1] ^ F(k,1) | ... | m[L] XOR F(k, L)`

And we have basically built a stream cipher from a block cipher.

This is semantically secure because for every adversary A attacking our `Edetctr`, there exists a `PRF` adversary B for `F` such that: `Adv_SS(A, Edetctr) = 2 * Adv_PRF[B, F]`.

As F is secure, the right hand side is negligible, so the left hand side is negligible.

Note: where does the `2` come from in that equation? The inequality was used in Stream Ciphers > Stream ciphers are semantically secure, but no explanation for the two given there? #question

Semantic security over Chosen Plaintext Attacks

We now allow the adversary for a Chosen Plaintext Attack. They can obtain as many ciphertexts as they want, encrypted with the same key.

Definition of semantic security over CPA

In Chosen Plaintext Attack, we again model semantic security with an adversary and a challenger.

Again, the adversary can submit two messages `m0, m1` and the adversary receives the encryption of either `m0` in experiment `EXP(0)` and `m1` in `EXP(1)`. But now we allow the adversary to iterate over this up to `q` queries! In `EXP(0)` they will always receive the encryption of the left message, and in `EXP(1)` they will always receive the right message. Let's call messages: `(m0,0; m0,1), (m1,0; m1,1), ... (mq,0; mq,1)`.

Note that if adversary wants `c = E(k, m)`, it queries with `mj,0 = mj,1 = m`, so he sends *both* messages in a query `j`. They are sure to receive `E(k,m)`

And as always, `E` is semantiaclly secure if for all efficient adversaries:

`Adv_CPA[A, E] = | Pr[EXP(0)=1] - Pr[EXP(1)=1] |` is negligible.

Note that **all ciphers we've seen are insecure under CPA**. More generally, if `E(k,m)` always outputs the same ciphertext for a message `m`, then this is insecure under CPA.

The attack is simple: `A` send `m0, m0` as both messages in round zero. Then they send `m0, m1` is round 1. If they

get `E(k, m0)` again, the now they are in `EXP(0)`. If they get `E(k, m1)`, a new string, they now they are in `EXP(1)`.

Lesson: if the secret key is gonna encrypt several messages, then the encryption must output different ciphertexts for the same message.

Making cipher secure under CPA

Solution 1: randomized encryption

Make `E(k,m)` a randomized algorithm. `E` itself is gonna choose some random string during the process and now `E(k,m)` can now correspond to several outputs.

Note that the ciphertext now must be long than the plaintext so as to recover these random bits. Roughly speaking: `CT-size = PT-size + #random-bits`.

Solution 2: nonce-based encryption

Make `E(k,m,n)` (add a new `nonce` value). A particular `k,n` must not be used more than once. The `nonce` needs **not** be random.

The simplest option is for the nonce to be a counter. Nice property: if decryptor has same state, I do not need to send nonce with the CT.

We can also just use a random nonce, and this is the same as solution 1.

Important: the system should be secure when an adversary chooses the nonce, but we place a restriction: they must choose distinct nonces. Letting them choose is important because they can generally attack at some arbitrary nonce, but they should not be able (in a correctly implemented system...) to get messages encrypted with the same nonce.

Modes of operation: Cipher Block Chaining (CBC)

Let's build CBC mode, which is secure under CPA when using a random nonce.

Let's see the CBC mode is semantically secure under CPA:

Let's take `L(ength) > 0`, if `E` is a secure PRP over (K,X) then `Ecbc` is semantically secure under CPA over `(K, X**L, X**(L+1))`.

For every `q-query` adversary `A` attacking `Ecbc`, there exists a `PRP` adversary `B` such that:

`Adv_CPA[A, Ecbc] <= 2 * Adv_PRP(B, E] + 2(q**2) (l**2) / |X|`

To argue that `2(q**2)(l**2)` is negligible, then we must make `|X|` be extremely big, way way bigger than `2(q**2)(l**2)`.

In practice: for AES128, we have `|X| = 2**128`. If we want the advantage to be less than `1/2**32`, then we must change the AES key after `2**48` blocks (because, simplifying a bit, we want: `q*L < 2**48`).

Important: CBC is no longer secure when the attacker can **predict** the IV. Rationale:

Suppose given `c = Ecbc(k,m)` an adversary can predict IV for next message. Then, adversary will send on `q=0` the encryption of one block, `0`, they receive `[IV1, E(k, 0 ^ IV1)]`. And now on `q=1`, they can predict (by assumption) `IV2`, so they send `m0=IV2 ^ IV1, m1=whatever`. What they receive will either be `[IV2, E(k, IV2 ^ IV1)]` or `[IV2, E(whatever)]`. They can now easily distinguish between the two as they already had `E(k, 0 ^ IV1) = E(k, IV1)`, and if they are on `EXP(0)`, `c = [IV2, E(IV2 ^ IV1) = [IV2, AES(IV2^IV1 ^ IV2) = AES(IV1)` , note the he had already had `AES(IV1)` from `q=0`!

Nonce-based CBC

You can use nonce on CBC, but you must use two independent keys to avoid the attack described in the

last section. You encrypt the nonce with they extra key! Then the attack cannot predict your next nonce.

What goes wrong if you use the same key to encrypt the nonce?

This was left as an exercise, not difficult but interesting:

`n nonce`, `iv = F(k, n)`, same `k` we will use for encryption.

First round: adversary sends `n=0`, `m=(0, 0)` (two blocks). Receives `(c0, c1)`. Note that `c1=F(k, c0)`, per definition of CBC mode, as `C1=F(k, m1 ^ c0)=F(k, c0)` as `m1 = 0`.

Second round: `n=c0, m1=c0^c1`. Receives `c0'`. Now note: `c0' = F(k, c0 ^ c1 ^ F(k, c0))`. `F(k, c0)` was `c1`. So `c0' = F(k, c0) = c1`. Now I can distinguish between a random permutation and CBC.

Padding

There's a padding oracle attack on CBC. I've exploited this with Cryptopals.

Also note that because of PKCS > 7, if you encrypt a multiple of 16 bytes, you need to add 16 extra bytes to your CP/CT.

Modes of operation: Randomized Counter Mode (CTR)

Superior to CBC mode. See CTR mode.

The `E(k, m) = E(k, IV) ^ m[0] | E(k, IV+1) ^ m1 |... E(k, IV+L) ^ m[L]`. The IV must be chosen at random for every message, and this is parallelizable (unlike CBC).

Semantic security: `Adv_CPA[A, Ectr] <= 2*Adv_PRF[B,F] + 2(q**2)L / |X|`. Again, this is negligible as long as `|X| >> q**2` (`>> = way way bigger than`). See that this is better than CBC! We can encrypt more blocks before loosing security.

Lesson: just use CTR mode instead of CBC mode. CTR...

- Uses a `PRF` instead of a `PRP`: counter mode just uses the encryption capabilities! (see my implementation). This may seem like a technicality but allows it to use more primitives like `Salsa` which are designed as a `PRP`.
- Can be parallelized.
- Is more secure, you can encrypt more messages with the same key while maintaining the same security.
- There's no dummy padding block, actually there's no need to pad with anything but zeroes, as you `E(k,`

`IV+x) ^ m`. As long as `IV+x` is the correct size, you can just `xor` with M correctly.
- Does not expands messages! Suppose 1-byte messages, with CBC you will get a 16-byte block. With CTR, you get a 1 byte CT :)

Test

4 is an interesting exercise revealing a weakness in a two-round Block ciphers > Feistel Network when trying to produce a `PRP` from a secure `PRF`.

- The `xor` of `F(k, 0**64)` and `F(k, 1**32|0**32)` reveals that the first 32 bits of output will always be one! Try it. **Goal**: integrity, not confidentiality.

We generally use Message Authentication Code or MAC for short.

Idea: Alice uses a MAC *signing* algorithm `S(k, m)` which produces a *tag* and appends it to her message `m`. On message reception, Bob runs a *verification* algorithm `V(k, m, tag)` which returns `true` if the tag matches the message.

Integrity requires a secret key: if you use CRC, you are detecting *random* not malicious, errors.

Secure MACs

On MAC, we assume a Chosen Message Attack: for `m1, m2,...,mq` attacker is given `ti <- S(k, mi)`

The attacker **goal** is to do a **existencial forgery**: produce some new valid message/tag pair; not even a `gibberish/tag` pair; not even produce a new tag `t'` for message `m` given `(m,t)`.

The MAC Security game

- `MAC = I(S,V)`, `Adv A` and `Chal. C`.
- Adversary sends `m1...mq` and receives `t1...tq <- S(k, m1) ... S(k, mq)`.
- Adversary sends a `(m', t')` different from the ones received by the challenger.
- Now `b=1` if adversary produced a new `message/tag` pair, otherwise `b=0`.

As usual, a MAC is secure if:

`Adv_MAC[A,I] = Pr[b=1] ~= negligible`

Secure PRF to secure MAC

Take `F: K x X -> Y` a `PRF`, define the MAC `If = (S,V)` such that:

- `S(k,m) := F(k,m)`

- `V(k,m,t) := yes if t = F(k,m)`

Note not all secure PRFs are secure MACs: for example `F: K x X -> Y` where `Y = {0,1}**10`. The output is too small!

But! This is the only scenario where a secure PRF does not produce a secure MAC. This is a theorem.

Theorem: `Adv_MAC[A, If] <= Adv_PRF[B,F] + 1/|Y|`. See that if `|Y|` is big, then negligible. The proof if simple, it is the advantage to distinguish the `PRF` which is negligible plus the luck factor on just hitting it, which is `1/|Y|`.

Easy lemma: you can truncate a secure MAC and make it output just t bits, as long as `1/2**t` is still negligible.

The McDonalds problem

We have AES a 16-byte MAC. Now how to convert this small MAC into a big MAC?

Two main constructions used in practice:

- CBC-MAC: used in banking
- HMAC: used in internet protocols

CBC-MAC

(also known as `ECBC`, `Encrypted CBC`)

Goal: given a PRF for short messages like AES, construct a PRF for long messages.

Notation: `X={0,1}**n` is the block size in bits. Think 128 bits por AES.

Take `F: K x X -> X`.

`F_ECBC: K**2 x X**(<=L) -> X` (note it takes two keys).

Algorithm: go through the CBC algorithm, encrypt each block but don't output anything until encrypting the last block. The encryption of the last block with `k` is then encrypted with `k1` the second key, and that is the tag.

`CBC-MAC` is very commonly used with `AES` and is a standard called `CMAC`.

NMAC

Take a `F: K x K -> K` (note output space is key space)

`F_NMAC: K**2 x K**(<=L) -> K` (note again the two keys)

Algorithm: take our message and break it into blocks, then do `F(k, m[0])`. Use that as key `k'` for `F(k', m[1])` and so on and so on.

The last block will output `t` an element in `K`. If you stop here you have a `cascade function`, which is not a secure MAC.

Then do `t || fpad`, a `fixed pad` (to make the element of `K` an element of `X`). And then do `F(k1, t||fpad)`, where `k1` is the second key.

`NMAC` is not generally used with AES. You need to change the key on every block... too slow. But it is the basis for HMAC!

What is with the last encryption: there exists `RAW-CBC` which does not use the last encryption, but it is **not** a secure MAC. And you can use the `cascade function` also.

But in `cascade` if you ask for `cascade(k, m)` you get `cascade(k, m||n)`! Because `cascade(k,m)` will be the key to `cascade(k||n)`.

In `RAW-CBC`, it is not that simple because you don't know `k`. But! You send `w` the message you want to append, which is only one block long, then do `t = RAW-CBC(k, w)` and now you got the encryption of that! So now you can generate a two block message `(w, t ^ w)`. And `t` is a valid tag for that.

Security analysis of MAC

You can use `MAC` up to `|X|**1/2` or `|K|**1/2` message with the same key.

For `AES`, this means that you have about `2**48` messages.

The security bounds are tight: once passed, you can attack it. After signing `|X|**1/2` message with `ECBC-MAC` or `|K|**1/2` with `NMAC`, then the MACs become insecure.

Attack MAC after lots of key reuse

Suppose the underlying PRF `F` is a PRP (eg, AES). Then both ECBC and NMAC have the following **extension property**:

- For every `(x,y,w); F(k,x) = F(k, y) => F(k, x||w) = F(k, y||w)`.

Now, attack:

1. Issue `|Y|**1/2` message queries for random messages in X.
2. You obtain `(mi, ti)` for `i=1, ..., |Y|**1/2`
3. Now find a collision `t_u = t_v` (one exists because of the Birthday Paradox).
4. Choose some `w` and query for `t := F(k, m_u || w)`.

5. And now I can forge `F(k, m_v | w)`.

MAC Padding

What to do when the message length is not a multiple of the block size?

Note that if you pad with zero you now have `pad(m) = pad(m || 0)`... and this is bad.

Padding must be invertible!: so `m0 != m1 => pad(m0) != pad(m1)`.

ISO pad

Pad with `1000.000`. Add a new dummy block if needed. The `1` indicates beginning of pad.

Remember to add the dummy block. If you don't, `MAC(m[0]...pad) = MAC(m[0]') (where m[0]' ends with the pad)`

CMAC

Goal: avoid prepending dummy block

Variant of a `CBC-MAC` where we use three key `k,k1,k2`. It has no final encryption step (expansion attack thwarted by last keyed xor) and no dummy block (ambiguity resolved by use of `k1` or `k2`).

Now:

1. If message is not a multiple of block length, pad with Message Integrity > ISO pad. Then `XOR` that block with `K1` and finish the process with `F(k, lastBlockXored)`
2. If message *is* multiple of block length, then `XOR` last block with `k2` and finish the process with `F(k2, lastBlockXored)`.

PMAC and the Carter-Wegman MAC

Goal: make the process of generating MACs parallel.

PMAC

You need two keys, then `Fpmac: K**2 x X<=L -> X`. Now do: `F(k1 ,m[0] ^ P(k,0)) ... F(k1, m[l-2] ^ P(k,l-2)), m[l-1] ^ P(k, l-1)`, where `P` is some function. Then we `XOR` all of these values and do a last `F(k1, XOR_OF_ALL)`. (note that you don't apply `F` to the last XOR, this is for technical, boring reasons)

What is P: if you did not have `P`, your MAC is insecure: see that if you swap two blocks, the MAC is the same (because you XOR everything anyway). So `P` is trying to enforce order. So `P(k, i) = k * i` in a finite field.

PMAC is incremental

Suppose `F` is not only a `PRF` but also a `PRP`. Suppose now `m[1]` changes to `m'[1]`, we can quickly update the tag!

Remember you can invert a `PRP`, so go from the tag to the xor of all the values, then XOR the old value `m[1] ^ P(k, 1)` to delete it from our current tag and then `m'[1] ^ P(k,1)` to get the tag for the new message.

One time MAC

`MACs` that are secure against **all** adversaries and are actually faster than `PRF-based` MACs. Keep in mind the key must be used only once!

Let `q` be a large prime (eg: `q = 2**128+51`). Let

- `key = (k, a) in {1...q}**2`,
- `msg = (m1... m[L]) with each block is 128 bit`

`S(key, msg) = Pmsg(k) + a (mod q)` where `Pmsg(x) = m[L]x**L + m[1]x` is a Polynomial of degree L.

Many Time MAC (Carter-Wegman MAC)

Let `(S,V)` be a secure one-time `MAC` over `(k, M, {0,1}**n`.

Carter-Wegman MAC: `CW(k1, k2, m) = (r, F(k1,r) ^ S(k2,m)` where `F` is a secure `PRF`. Note that the slower `PRF` is applied to `r` a small random number and `S` the one-time-mac is applied to an arbitrarily large message.

Collision Resistance

What makes a hash function collision resistance?

Let `H: M -> T` be a hash function (where `|M|` is much bigger than `|T|`). A collision for `H` is a pair `m0, m1` such that `H(m0) = H(m1)` when `m0 != m1`.

A function is Collision resistant if for all efficient algorithms A `Adv_CR(A, H) = Pr[A outputs collisison for H]` is negligible.

MAC from Collision Resistant Hash function

Let `I = (S,V)` be a `MAC` for short messages over `(K, M, T)` (eg: AES). Let `H: BIG > M`.

Now we can expand our `MAC` for large inputs like this: `Sbig(k,m) = S(k, H(m)); Vbig(k,m,t)=V(k, H(m), t)`

The theorem is that if `I` is secure and `H` is collision resistant, then `Ibig` is a secure MAC.

Generic Birthday Attack

An attack like our exhaustive search for the keys on ciphers. The conclusion is the same: collision resistant hash functions must have a certain size for their output, else they are insecure.

Let `H: M -> {0,1}**n` be a hash function ,where `|M|` `way bigger than 2**n`. A generic algorithm to find a collision in time `O(2**(n/2))`:

1. Choose `2**(n/2)` random messages in `M`, `m1, ..., m(2**(n/2))` (which is distinct very likely because `M` is so big)
2. Compute `ti = H(mi)`
3. Look for a collision `(ti = tj)`

If we don't find a collision, then go back to `1`.

How well does this work?: well, you should now about the Birthday Paradox (see that note, there's the uninteresting proof). The important thing to know is that if you choose `sqrt(n)` elements where `n` is the total size of the set, you have a ~50% chance of finding a collision

Merkle-Damgard Paradigm

Given a collision resistant for short message, how can we expand it to arbitrarily long messages?

Have `h` our `hash` function for small messages.

Notations: `h` is sometimes called a *compression function*.

For message `m0... mn || PB`, for:

1. `m0` we do `c0 = h(m0, iv)` (the `iv` is fixed in the standard).
2. `m1` does `c1 = h(m1, c0)`...
3. `m2` does `c2 = h(m2, c1)`...
4. And so on, until `mn`, which does `h(m3 || pb, cn-1)`, where `pb` is a `padding block` and must be there.

The `padding block` is a `1` followed by zeroes, and then 64 bits of the msg len.

Fun fact: from this derives that a hash function using only 64 bits for the length can take *arbitrarily* large input, only `2**64-1` bits. But that's big enough as to be irrelevant.

Theorem: if `h` is collision resistant, then `H` the process of doing a Merkle-Damgard with `h`, is also collision resistant.

Proof: suppose the contrapositive: `collision on H => collision on h`.

Call the intermediate output of each Merkle-Diagram process as `Htn` for the message `H(m)` and `Hrn` for the message `H(m')`.

Suppose `H(m) = H(m')`. We build a collision for `h`. Then: `h(Ht, Mt || PB) = Ht+1 = H'r+1(H'r, M'r || PB')`

Then, supposing `Ht = H'r, Mt = M'r and PB = PB'r` this is fine. But we now that `M != M'`, so this does not hold. So by iterating all of this up until the beginning we prove that if `h` is collision resistant, then `H` is collision resistant.

Building Collision Resistant Compression Functions

Also known as a Hash Function

Can we build compression functions from block ciphers?

Yes. `E: Kx{0,1}**n -> {0,1}**n` a block cipher. The Davies-Meyer compression function `h(H, m) = E(m, H) ^ H`, with `H` the chaining variable

More clear notation from Wikipedia: `Hi = E(mi, Hi-1) ^ Hi-1`. The big `H` in the course notation is the result from the previous hash. It starts with a fixed chaining variable.

Theorem: suppose `E` is an ideal cipher, finding a colision `h(H,m) = h(H', m')` takes `O(2**(n/2))` (and by the Birthday Paradox, this is the best that we can ever do)

Fun fact: all SHA functions use the Davies-Meyer construction.

What about the XOR: If you don't have the `XOR`, so `h(H, m) = E(m, H)`, you can find a collision on `(H, m, m')` by applying `h` to an `H'` such that `H' = D(m', E(m, H))`.

See:

1. `H' = D(m', E(m, H))`
2. `E(m', H') = E(m, H)`
3. And there you... that's the collision.

Other constructions

There are other constructions like Miyaguchi-Preneel, which goes `h(H, m) = E(m, H) ^ H ^ m` (actually 12 similar variants of that)

Many other natural variants are insecure. Example: `h(H, m) = E(m,h) ^ m`.

Case study: SHA-256

- Merkle-Damgard function
- Davies-Meyer compression function
- Block cipher: SHACAL-2

Provable compression functions

Choose a random `2000-bit` prime `p` and choose two random variables `1 <= u, v <= p`. Now, for `m,h in {0,..,p-1}`, define `h(H,m) = u**H * v**m (mod p)` (takes two numbers and outputs one).

Theorem: This equivalent to breaking the Discrete Log problem (see also Number Theory > Discrete Log problem).

Proof: Choose a group `G` where `Dlog` is hard (eg: `(Zp)*` for large `p`).

Let `q = |G|` be a prime. Choose generators `u, v` of `G`. For `x,y` in `G`, define `h(H,m) = u**H * v **m`. Now let's see that this is the same as finding a `Dlog_g` of `v`:

Now suppose we have a collision: `h(x, y) = h(x', y')`. So now:

```
u**x * v**y = u**x' * v**y'
u**x' * v**y' = u**(x-x') = v**(y'-y)
v = g**((x-x')/y'-y) (exp. by 1/(y'-y))
```

So to find `v` you had to solve this discrete log.

The **problem** with this is that it is really slow.

HMAC

Can we construct a Message Integrity > Secure MACs without doing a Secure PRF to secure MAC kind of process?

We have a Merkle-Damgard Paradigm kind of thing going, where `H: X**(<=L) -> T`.

Bad Attempt

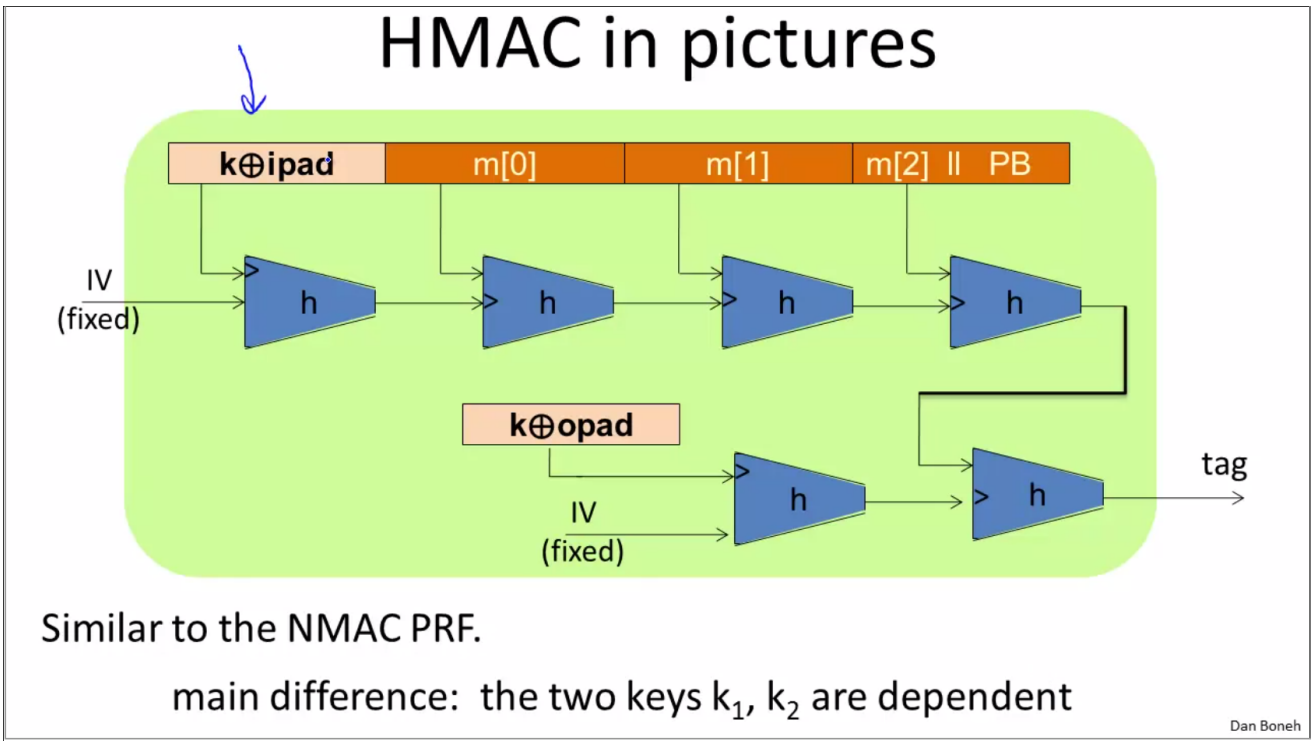
Let's just do `S(k, m) = H(k || m)`

Well, that's insecure because of a Length Extension Attack: given an output from that `H` I can consturct `H(k||m||PB||w)`.

Correct attempt

Use HMAC. You need `H` a secure hash function. Let's use `SHA-256`. Then:

`HMAC: S(k, m) = H(k ^ opad, H(k^ipad||m))`



This is very similar to NMAC! The main difference is that two keys `k1, k2` are just the same key xored with different constants.

Both `opad` and `ipad` are just constants.

Timing attacks on HMAC

Let's look at an example on `Keyczar` crypto library in Python.

```
def verify(key, msg, sig_bytes):
        return HMAC(key, msg) == sig_bytes
```

Problem: `==` is a bite-by-bite comparison, so the comparator returns `false` whenever the first inequality is found.

Solution 1: make string comparator always take the same time

Solution 2: `mac = HMAC(key,msg); return HMAC(key, mac) == HMAC(key, sig_bytes)`: if it is right then true, if it is wrong; false; but the comparison will compare two *very* different outputs!

What does authenticated encryption provide?

We have talked about Confidentiality with security against a Chosen Plaintext Attack with things like Stream Ciphers and Block ciphers.

We have talked about Integrity to provide existential unforgeability under a Chosen Plaintext Attack.

We will now see encryption schemes against tampering, providing both Confidentiality and Integrity at the same time.

Sample tampering attacks

With an attacker on a TCP host

See how the lack of Integrity completely breaks security.

Imagine a TCP/IP connection over a `source machine` and a `server`. The source machines sends a packet `dest=80|data`, the TCP/IP stack of the receiver

receives this packet and forwards this to the process that is listening on port 80.

Now, IPSEC will encrypt this IP packets. Sender and recipient have a shared key `k`, the TCP/IP stack will decrypt the packet before forwarding it to the actual port.

Let's see how without integrity we can't achieve confidentiality: if `Bob` an attacker intercept and change a packet, knowing that it starts with `dest=80`, they will change the first bytes of the message to read `dest=25` so it is forwarded to port 25, where he is listening! And then they can see the data freely.

How do they change the first bytes? They only need to change the IV.

The objective is to go from `IV, dest=80|data` to `IV', dest=25|data`. Now, assume CBC mode, a Chosen Plaintext Attack-secure mode. Remember:

- `m[0] = D(k, c[0]) ^ IV = dest=80`
- `m'[0] = dest=25 = D(k, c[0]) ^ IV'` ... what should `IV'` be?
- `m'0 = dest=25 = D(k, c[0]) ^ IV ^ (..80..) ^ (..25..)` (take the `80` add the `25`)
- Done!

With an attacker only with network access

Imagine a remote terminal application where for every keystroke on the client an encrypted keystroke is sent to the server with CTR mode.

Fact: TCP/IP packets include a checksum.

So our packets looks like this, all is encrypted: `IP header | TCP header | 16-bit checksum | 1-byte Data`

So the attacker wants to know what keystroke was sent.

1. Intercepts packages
2. Stores it
3. Modifies package such that: `IP hdr | TCP hdr | ^ t | ^ s` (tries with several `t` and `s`)
4. Sends the modified packages to the server
5. Only valid tags will be valid
6. If the checksum was valid the the attacker knows that the `data ^ s` corresponds to the `checksum ^ t`

And by knowing that, with some simple checksums, you can derive the actual `data`.

Defining authenticated encryption

An **authenticated encryption** system is a cipher `(e, d)` where as usual `e: k x m x n -> c` but `d: k x c x n -> m u {!}` where `{!}` is called the `bottom`, an unique symbol outside of the message space that indicates the message has been tampered with and should be ignored.

the system must provide:

- semantic security
- and **ciphertext integrity**: even if attacker sees a number of ciphertexts, they cannot produce a new ciphertext that decrypts to something else than `{!}`

definition: a cipher has **ciphertext integrity** if for all efficient adversaries, the `adv_ci[a, e] = pr[challenger outputs 1]` is negligible, where the challenger outputs `1` when presented with a valid ciphertext from an adversary that has seen many valid, different ciphertexts.

bad example: cbc mode with a random iv does not provide ae: every random ciphertext is *valid* as it never outputs `{!}`

implications of ae

1. authenticity: attacker cannot fool bob into thinking a message was sent from alice (although it could be

a replay...)
2. defends against chosen ciphertext attacks...

Chosen ciphertext attacks

see chosen ciphertext attacks.

Chosen ciphertext security

let's define a security model where the adversary's power is both to perform chosen plaintext attacks and chosen ciphertext attacks: can obtain the encryption of arbitrary messages and can decrypt any ciphertext of his choice, other than the challenge ciphertext itself.

the adversary's goal is to break semantic security.

let's do a more **formal definition**, again using a game with an adversary and a challenger. the only difference is that every query `q` can be either a `cpa query` with a `(m0, m1)` in `m` and gets back the ciphertext; or a `cca query` where it sends a `ci` in `c` different from the previous `ci`, and they get the `mi <- d(k, ci)` (the only restriction is that `ci` must not be the result of a `cpa query`).

the objective of the adversary is to know if they are in experiment zero or experiment one: we say that `e` is

`cca secure` if `adv_cca[a, e] = |pr[exp(0)=1 - pr[exp(1)=1]|` is negligible.

example: cbc with rand iv is not cca-secure

1. `a` sends `m0, m1` gets `c` the encryption of one of those.
2. `a` now sends a `cpa query` with `c' = (iv^1, c[0])`
3. `a` will receive `d(k, c') = m[0] ^1`
4. `a` now knows if he got the `c` from `m0` or `m1`

Authenticated Encryption means CCA Security

Theorem: `(E,D)` be a CPA cipher that provides `AE`, then `(E,D)` is `CCA secure`!

For any `q-query` efficient `A` there exists `B1, B2` such that:

```
Adv_CCA[A,E] <= 2q * Adv_CI(B1, E) + Adv_CPA(B2, E)
```

Proof: Note that the `Challenger` will always output `{!}`, as the adversary cannot create a `c` different from the ones provided by the challenger itself. this is because `E` is an authenticated encryption cipher, this is by definition.

As the `CCA Attack` queries always result in `{!}`, we can just ignore it!

So we just have `CPA Attack` queries, and `E` was `CPA` secure... so it is `CCA secure`!

Standard Constructions for Authenticated Encryption

- AE was introduced in the year 2000 by two independent papers
- Crypto APIs before then provided methods to do encryption and mac separately
- But not all combinations of MAC and encryption provide AE...

Let's look at what three projects did:

1. SSL: `msg m => msg m | tag => E(msg m | tag)`
2. IPSEC: `msg m => E(k, m) => E(k,m) | tag(E(k,m))`
3. SSH: `msg m => E(k, m) => E(k,m) | tag(m)`

Now, their problems:

1. SSL: bad, `m||tag` can be modified then, called MAC then encrypt
2. IPSEC: this one is great! called Encrypt then MAC

3. SSH: `tag(m)` is not guaranteed to not leak bits of `m`, this one is called Encrypt and MAC.

Note MAC then encrypt *may* be insecure, not necessarily! Actually, `(E,D)` over random CTR mode or random CBC mode, MAC then encrypt is actually fine.

Standards:

1. GCM Mode: CTR mode then CW-MAC
2. CCM Mode: CBC-MAC then CTR mode (note MAC then encrypt)
3. EAX Mode: CTR mode then CMAC

All support **AEAD**: Authenticated Encryption With Associated Data.

All are nonce-based.

An addendum to Message Integrity's MAC security

When discussing Message Integrity > Secure MACs we said that given `(m,t)` valid message and tag, an attacker can not produce a `(m, t')` a `'t'` which is *also* a valid tag.

Why? This should not really be important.

If we allow for this, then Encrypt then MAC would not have ciphertext integrity!

See the game with an Encrypt then MAC schema:

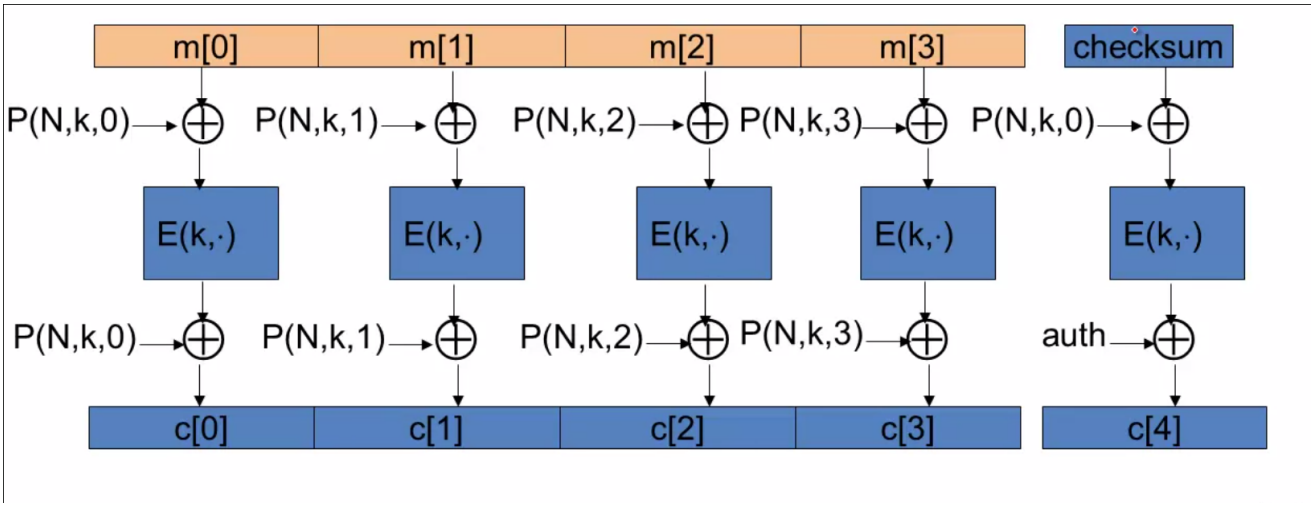
1. Adversaries sends `(m0, m1)` and receives `(c0, t)`.
2. Now adversaries produces `(c0, t')` (he can do this by assumption)
3. Submit a chosen ciphertext query with `(c0, t')`
4. `(c0, t')` is a valid, different ciphertext from the one received, so the challenger must respond with `D(k, c') = mb`
5. Now the adversary knows if they are in `EXP(0)` or `EXP(1)`

OCB: direct AE from a PRP

Once AE was formalized, people started looking for ways to get it in a more efficient way than by first encrypting the MAC-ing the whole thing.

Objective: we want to perform one `E()` operation per block, nothing more.

Turns out we can and that construction is called OCB.



The function `P` is a very simple function so almost no performance penalty apparently.

So why isn't it used?: it is patented :(but is not licensed outside of the US and also they don't care about open source software using it. Actually starting from 2021 it is not patented anymore!

AE Pitfalls

Case study: TLS

The protocol is called the TLS Record Protocol (TLS 1.2).

Packets look like this: `HEADER | TLS RECORD`, where `TLS RECORD` is less than 16 kilobytes.

It uses **uniderectional keys**: one key goes from server->browser and another one goes browser->server. Both server and browser know both keys: `kb->s, ks->b`.

The TLS record uses Stateful Encryption: the encryption uses some state that is maintained by the server and browser, in particular two counters: `ctr_b->s` and `ctr_s->b`. They are zero when session is started. They purpose is to avoid replay attacks.

Let's see encryption with the mandatory algorithms: CBC AES-128 and HMAC-SHA1.

Let's browser -> server data. The `Kb->s = (Kmac, Kenc)`, one for mac and other for encryption. Note that there are four keys! Two for b->s and two for s->b.

Now, a packet looks like this: `TYPE|VER|LEN|DATA|TAG|PAD`. The browser side algorithm for encryption is:

1. tag <- `S(Kmac, [++ctr_b->s, | header | data])`
2. `pad(header|data|tag)` to AES block size
3. `E(Kenc, data|tag)` with a random IV.
4. Prepend the header.

Note: see that the `ctr` value is not actually sent! Because the browser already know what it should be.

Now, to decrypt, the server will:

1. `D(Kenc, packet)`
2. Check the pad: send `bad_record_mac` if invalid

3. Check tag on `[++ctr_b->s | header | data]` send `bad_record_mac` if invalid

Mistakes in previous versions

Prior to TLS 1.1:

- the IV for CBC is predictable: IV for next record is last ciphertext block of the current record, so the scheme is not CPA secure. The attack on this was called BEAST Attack.
- a Padding Oracle Attack was possible, because if the pad was invalid it sent one message and if the mac was invalid it sent another one. I implemented this for Cryptopals.

Case study: WEP

`802.11b WEP` is a great example of how *not* to do stuff. The protocol encrypt messages from a computer to an access key.

There's a shared key `k`. The WEP message is computed like: `m | CRC(m)` where `CRC` is a checksum.

The result is encrypted using a stream cipher: `PRG(IV | k)` and then the `IV` and the ciphertext are transmited.

Then:

1. If the `IV` is repeated, you get a two time pad attack. And it is easy to get repeated IV.
2. The key is always fixed and the only thing that changes is the `IV`, and the `PRG` used in WEP breaks when using related keys.

But even the `CRC` is broken!

Fact: CRC is linear, ie: `CRC(m ^ p) = CRC(m) ^ F(p)`, where `F` is well known public function.

Now, if an attacker wants to change the message, they just do that, which they can express as `m' = m ^ X`. And for the `CRC` checksum, they just do `CRC' = CRC(m) ^ F(X)`!

So Integrity is also broken.

CBC Padding Attacks

CBC mode with a MAC is an authenticated encryption scheme, but the implementation must be correct! Let's see how it was broken in TLS.

If old versions of TLS, if you had a padding error, you got `padding error`. If you had a MAC error, you got `MAC error`.

Fun fact: even after implementations fixed this and started reporting the same error for the both problems,

there was a timing attack there.

Now we can mount a Padding Oracle Attack: suppose attacker has `c = (c[0], c[1], c[2])` and wants `m[1]`.

1. Throw away `c[2]` and keep `c[0], c[1]`.
2. Create `g` a guess for the last byte of `m[1]`
3. `last_byte([c0]) ^ g ^ 0x01`
4. Now, remember in CBC mode `D: D(k, c[i]) ^ c[i-1]`. So, now: `last_byte(D(k, c[1])) ^ g ^ 0x01`
5. If guess was correct, `last_byte(D,c[1]) ^ g` was correct we get `0x01` as the last byte of the plaintext!
6. And this would yield a valid padding.

Now repeat, but `xor` with `g' ^ 0x02`. You can set the last byte of `m[1]` to be `0x02` by setting `last_byte(c[0]) ^ correct ^ 0x02`, and so on and so on.

Problem: TLS renegotiates key when an invalid record is received. But this could actually be exploited on IMAP over TLS.

Note: if TLS had used Encrypt then MAC, this would have been avoided...

Attacking non-atomic decryption

On SSH binary packet protocol.

A packet looks like this:

```
seq. no | packet_len | pad_len | data |
payload | tag
            |              ENCRYPTED
|
```

Fact: packet_len is 32 bits long

1. Decrypt packet length field only (!)
2. Read as many bytes as length specifies
3. Decrypt remaining ciphertext blocks
4. Check MAC tag and send error response if invalid

Note you cannot check the tag before using the length!

Now suppose attacker has `c = AES(k, m)` and it wants `m`.

1. Send a packet to the server `seq no | c`
2. It will interpret the first bytes of `c` to be the length
3. Now attacker feeds server one byte at a time... until the server reads as many bytes as the length field specified.

4. And now it will verify the MAC, which is of course is gonna fail
5. But now we know how many bytes the server accepted before checking the MAC
6. So now we know the first 32 bits of `m`!

Deriving many keys from one

A single `source key` is generally easy to obtain, either via a Key Exchange Protocol or a hardware random number generator.

But how can we derive several keys from this one? It is generally necessary, for example to derive uniderectional keys, or multiple keys for nonce-based CBC mode.

The answer is: we use a Key derivation function. Read that! This works if the source key is uniform in the key space.

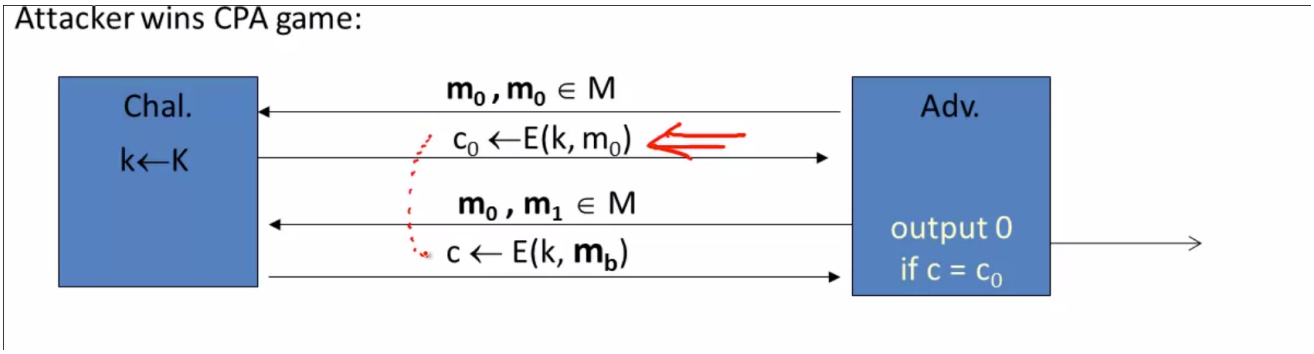
But what if `SK` not be uniform?

Deterministic Encryption

Usecase: searching on encrypted databases

To look for Alice, I just need to send to the Database `E(k1, "Alice")` and I will find it!

This is **not** Chosen Ciphertext Attack! By looking at the ciphertexts, they know that the messages are equal. This is particularly important when the message space `M`, the message space, is small.



To fix this, we can **assert** that the encryptor **never** ever encrypts the same message twice: the pair `(k,m)` never repeats (for example: the messages are random from a large `M` or the message itself ensures uniqueness, for example with unique user ID).

So now we have the concept of Deterministic Chosen Ciphertext Attacks! Read that.

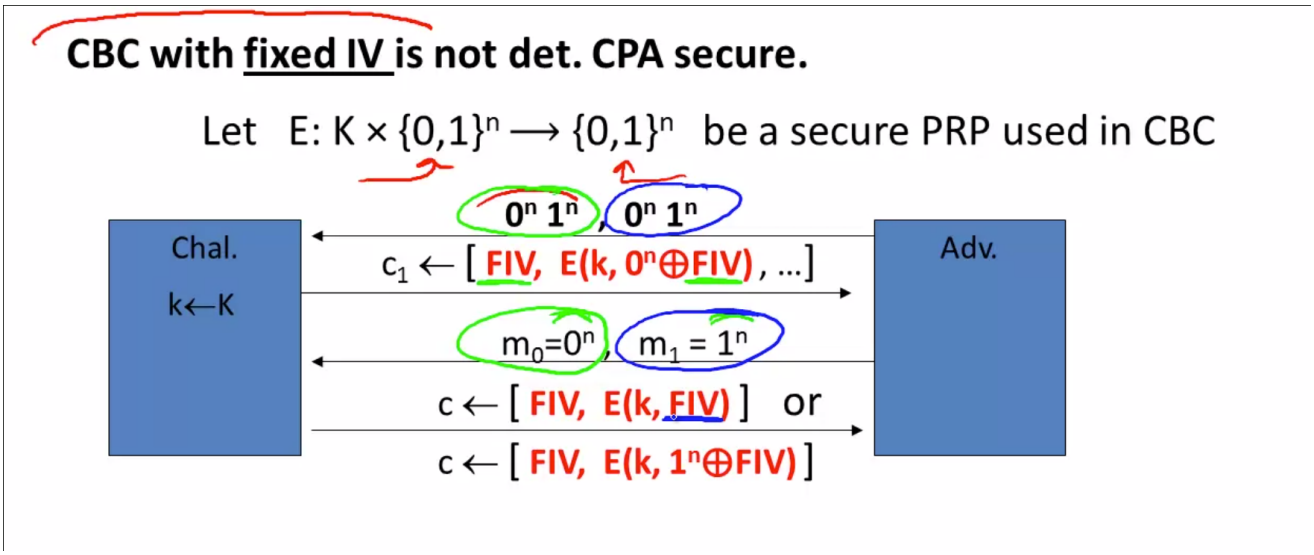
CBC with fixed IV is not deterministic-CPA secure!

Let `E: K x {0,1}**n` be a secure Pseudo Random Permutation used in CBC mode. See how can I attack it:

1. Ask for encryption of the messages `{0**n, 1**n}, {0**n, 1**n}`
2. Receive `[IV, E(k, 0**n ^ IV), ...]`: the first block of the ciphertext is just the encryption of the

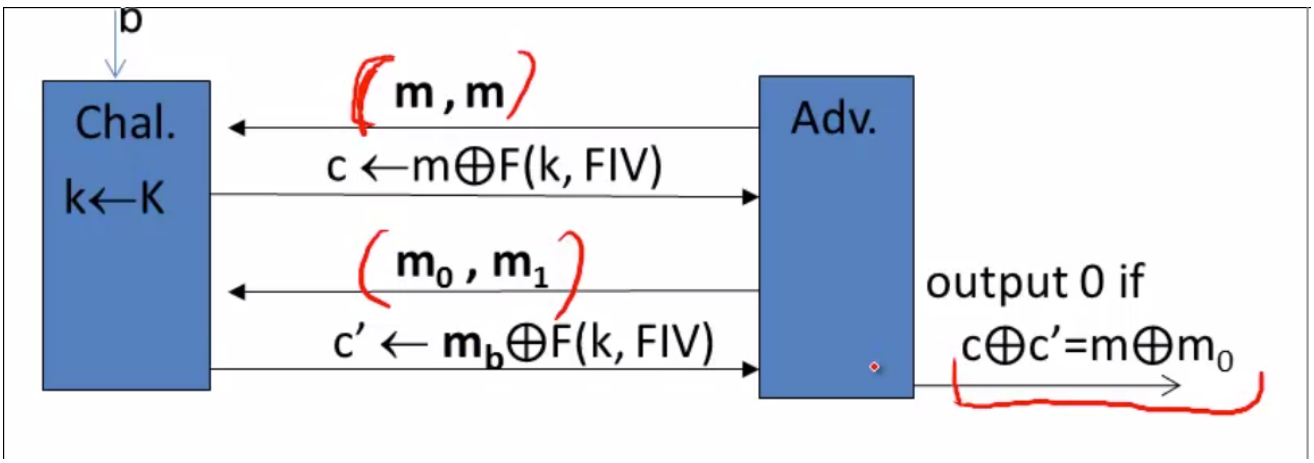
`IV`!

3. Now ask for `0**n, 1**n`. This is a valid deterministic CPA query because they are different from the previous ones.
4. Now I can now which one I received: the challenger is gonna output either `[IV, E(k, IV)]` for the zero message or `[IV, E(k, 1**n ^ IV)]` for the right message!



This apparently leads to significant attacks in practice.

CTR mode with a fixed IV is also not Determinisic CPA Secure...



Secure deterministic encryption: SIV and Wide PRP

Secure under Deterministic Chosen Ciphertext Attack of course.

SIV

`Synthetic IV`. Let `(E,D)` a secure Chosen Ciphertext Attack encryption: `E(k, m ; r) -> c` where `r` is a random variable used by the `E`.

Let `F: K x M -> R` be a secure `PRF` and `R` is the space where `r` lives.

Define: `E_det((k1,k2), m) = E(k2, F(k1,m)) = c`.

Proof/intuition: this is secure `E` because it is encrypting stuff that is indistinguishable from random when providing different messagse.

Cool feature: you get integrity for free: this means SIV is a **DAE**: a Deterministic Authenticated Encryption system.

Consider **SIV-CTR**: SIV where cipher is counter mode with random IV. Use the output of `F(k1, m)` as the IV; then `E = message ^ E'(k2, IV|IV+1...IV+L) = c`. Now, when I decrypt `E'` and apply `F` to it, I should get back to `IV`.

Wide PRP

Let's begin by just using a Pseudo Random Permutation. Let `(E,D)` be a secure PRP `E: K x X -> X`.

Theorem: `(E,D)` is semantically secure under deterministic CPA.

Proof sketch: Remember attacker is not allowed to send two equal messages.

- In `EXP(0)` adversary sees results of `f` an actual invertible function from `X -> X` .
- In `EXP(1)` adversary sees results of `E(m1)...E(mq)`, but we know our PRP is indistinguishable from random.

So attacker can't distinguish!.

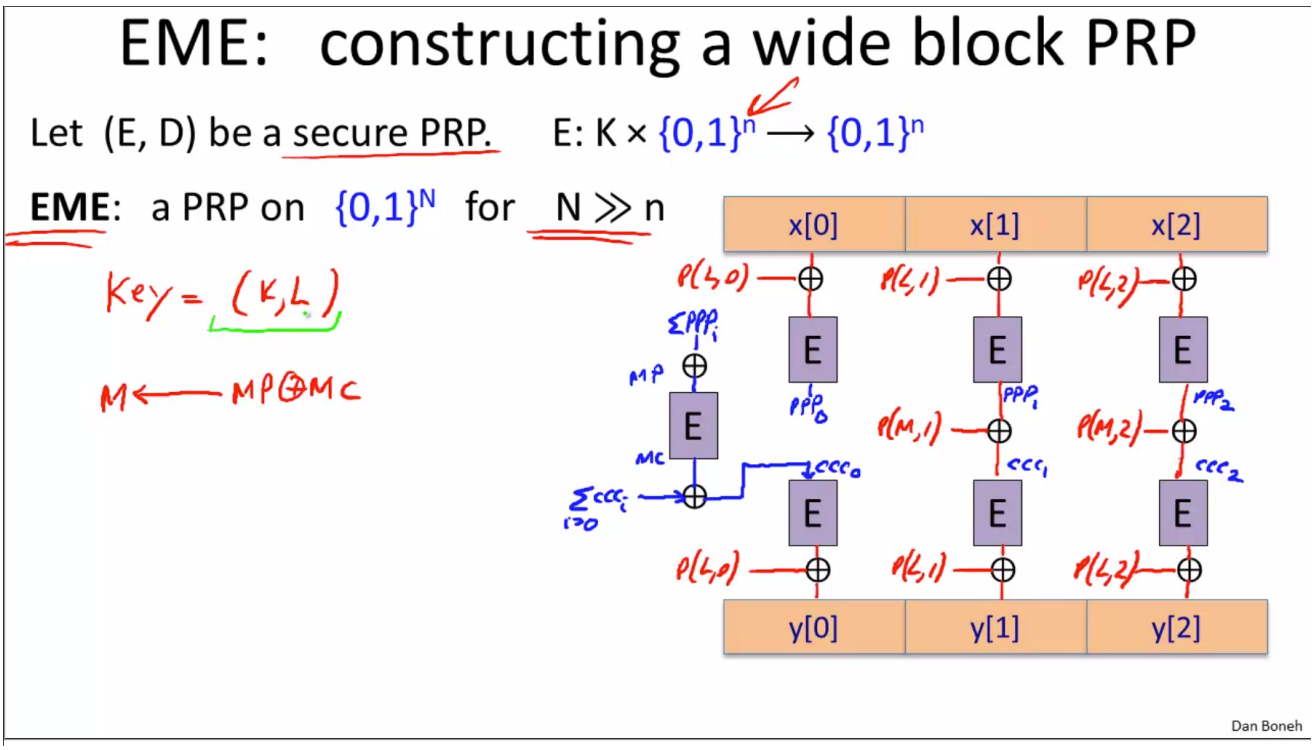
But the problem is that AES only is deterministically CPA secure for 16 byte messages, and notice that this does not provide integrity.

But what if we need a bigger message space?

EME

Let `(E, D)` be a secure PRP `E: K x {0,1}**n -> {0,1}**n`. Now EME is a PRP on `{0,1)**N` where `N >> n`.

It uses two keys: `(k,l)` (in reality `l` is derived from `k`, we can ignore that).



What a complicated mess.

The problem is that this is 2x slower than SIV.

Integrity to PRP-based deterministic encryption

Very simple: append zeroes during decryption and check that they are there when it is decrypted check that the zeroes are still there.

The only thing to look at is that you should add enough zeroes!

The idea is that when encrypting you are doing `E(m || 0)`. But this is as indishtinguisable from random as `E(m)`. You can't produce a ciphertext that would decrypt to the specific amount of zeroes!

Tweakable Encryption

We will explain it using Disk Encryption as an application.

Problem of disk encryption: no size to expand! If the disk sector are a fixed size (eg: 4KB), then encryption cannot expand. And we must use deterministic encryption (where would we store the randomness?). And we cannot add integrity, as we cannot add integrity bits.

Lemma: if `(E,D)` is a det. CPA where secure cipher with `MESSAGE SPACE = CIPHERTEX SPACE` then `(E,D)` is a `PRP` (remember Deterministic Encryption > Wide PRP?)

But we don't want to leak information about two sectors being the same. How can we do better?

Yes, let's see ideas.

Idea 1: Use different keys for different sectors

- Avoids leak when sectors are the same
- But attacker can tell if a sector is changed and then reverted

To manage keys: `Kt = PRF(k,t), t=1...L`. So no need to store the keys.

But we can do even better...

Idea 2: Tweakable block cipher

Goal: Construct many PRPs from a key. **Syntax**: `E, D: K x T x X -> X`, the `T` is called the `tweak space`

For every `t in T` and `k in K`, `E(k, t, .)` is a invertible function ox `X` indistinguishable from random!

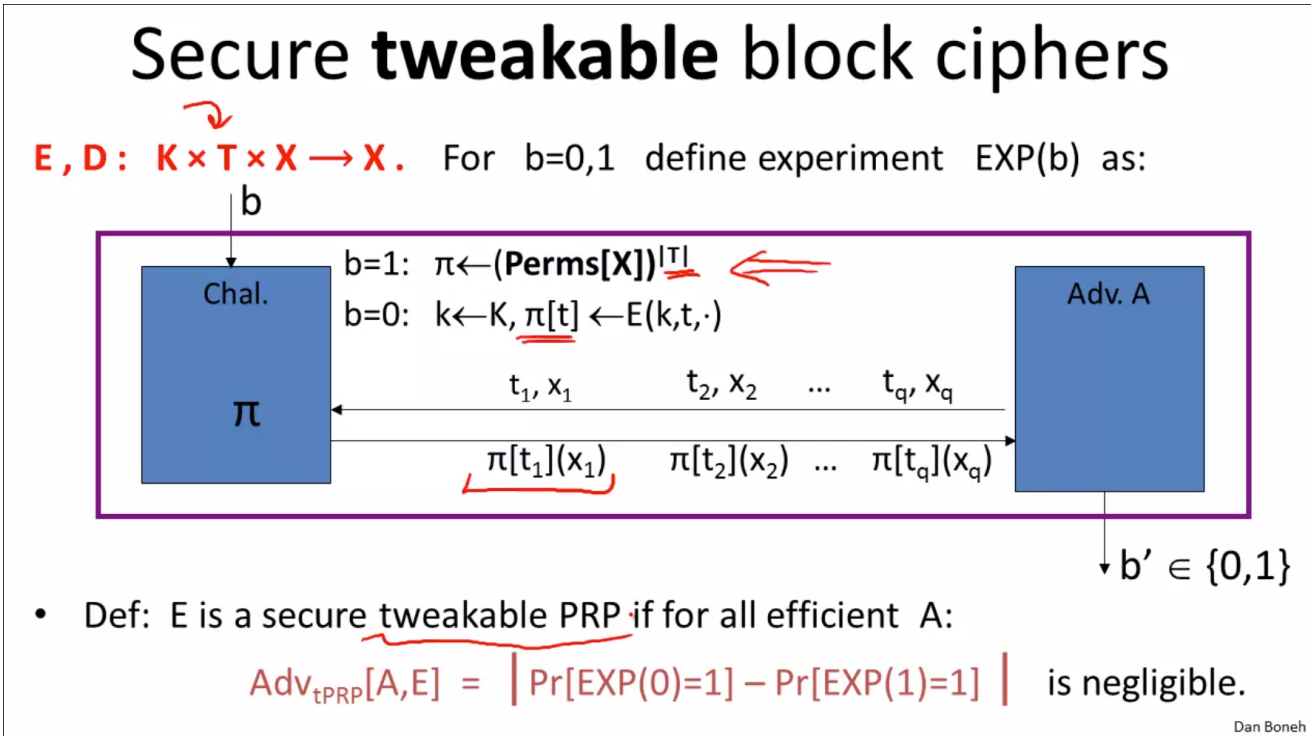
Application: use the sector number as the tweak, and then every sector gets its own independent PRP! We avoid having to compute the `PRF` for every sector.

Secure tweakable block ciphers

`E,D: K x T x X -> X`. We define a new security game:

- `EXP(1)` we choose `|T|` random permutations of `X`
- `EXP(0)` we choose a `k` and choose `|T|` evaluations from `E(k,t,.)`

Now the adversary submits `t1, x1` and they can see the value of the permutation `permutation(t, x)`. They must now distinguish from the truly random permutations from `X` or from the ones given by our `E`.



The trivial tweak

`E(k,t,x) = E((k,t), x)`

Note we must do `2n` calls to `E`! We can do better.

XTS tweakable block ciphers

Have `E: K x {0,1}**n -> {0,1}**n` a regular block cipher.

Now `XTS: E_tweak((k1,k2), (t,i), x)`. The way this works is:

1. `N <- E(k2, t)`
2. `m' <- m ^ P(N, i)`, `P` a fast padding function
3. `e1 <- E(k1, m')`
4. `e1' <- e1 ^ P(N, i)`
5. `e1' = c`

Note that to encrypt `N` blocks we need `N+1` applications of `E`.

Using XTS from block encryption

Each block is encrypted with `(t,i)` where `t` is the sector number and `i` is just the block number. So each block is encrypted with its own PRP.

This is used in Mac OS X-Lion, TrueCrypt...

Format Preserving Encryption

Very common to encrypt credit card numbers.

Credit card format

`bbbb bbnn nnnn nnnc`.

- `b`: the bit number, the ID of the issuer.
- `n`: the account number
- `c`: checksum

Around ~42 bits of information (`2**42` possible card numbers)

Our **goal** is to have End to end encryption between our POS Terminal and acquiring bank. But the processors in the middle expect something that *looks* like a credit card number

Constructing FPE

Given `0 < s < 2**n`, build a Pseudo Random Permutation on `{0,....,s-1}` from a secure PRF `F: Kx{0,1}**n -> {0,1}**n`.

Having that, we can use it to encrypt credit cards.

1. Map given CC number to `{0,...,s-1}`
2. Apply PRP to get an output in `{0,...,s-1}`
3. Map output back to `CC#`

Step 1: `{0,1}**n -> {0,1}**t`

We want PRP on `{0,...,s-1}`. Let `t` be such that `2**(t-1) < s <= 2**t` (ie: the closest power of 2 that is bigger than `s`)

Method: Luby-Rackoff Construction with `F': Kx{0,1}**(t/2) -> {0,1}**t/2` (truncate F). See Block ciphers > DES

- Truncating the output of a PRP like this is simple, but it is not the best way. But for our methods, that works.
- We plug it into the Luby-Rackoff Construction so as to convert our truncation into an invertible function, aka a Pseudo Random Permutation. The best way is actually to use a 7-round Feistel Network

Now we have Pseudo Random Permutation that operates on `{0,1}**t`

Step 2: `{0,1}**t -> {0,...,s-1}`

Define `E'(k,x)`, which for an `x` in `{0,...,s-1}`, will `y <- x; do {y <- E(k,y)} until y in {0,...,s-1}; output y`

So basically encrypt until we hit our target set.

Is this secure?

Step 2 is pretty easy: Yes. We are encrypting several times on a secure thing! Believe it.

Step 1: Same as analysis as a Feistel Network

Note that there's no integrity.We will start seeing Assymetric Cryptography to understand how to establish a shared secret key.

Trusted 3rd Parties

N users, all store mutual secret keys.

Problem: Every user has to store `N-1` keys.

Solution: Establish a trusted 3rd party. Every user shares a secret key with the TTP.

What happens when Alice wants to talk to Bob? They must somehow establish a shared secret key.

Toy Protocol for Generating Keys

`Alice <-> Bob`, and they want only eavesdropping security only.

```
Alice -> TTP: Alice wants key w/ Bob
TTP: choose random Kab,
TTP -> Alice: E(Ka, "Key for A,B"||Kab)
TTP -> Alice: ticket = E(Kb, "Key for
A,B"||Kab)
Alice: Decrypts her Kab
Alice -> Bob: The ticket
```

Why is this secure?: If `(E,D)` is secure, the eavesdropper leans nothing about `Kab`

Note: The TTP knows all session keys! :scream:

Note: This is basis of the Kerberos system.

Note: This is completely insecure against an active attacker. Imagine a replay attack!

Key question: can we do this without an **online** trusted 3rd party? And of course this is possible, and this is the starting point of Assymetric Cryptography. Merkle, Diffie-

Hellman, RSA... and more recently ID-based Encryption (2001) and Functional Encryption (2011)

Merkle Puzzles

Our first key exchange without a Trusted 3rd Parties.

For now: security against eavesdropping only.

Question: can we discard the trusted 3rd party using *only* Symmetric Encryption and Hash Functions and the like?

Answer: Yes! But very inefficient, and they are not used in practice. Let's study them anyway.

Main tool: puzzle.

What is a puzzle? A problem that can be solved with some effort

Example: `E(k,m)` a symm. cipher with `k` in `{0,1}**128`. Now `puzzle(P) = E(P, "message")` where `P = 0**96 || b1..b32`. The goal of the puzzle is to find `P` by trying all `2**32` possibilities.

So the algorithm works like this.

1. Alice prepare `2**32` puzzles. For `i=1..2**32` choose random `Pi` of 32 bits, and `xi, ki` are 128

bits. Set `PUZZLEi <- E(0**96 || Pi, "Puzzle #xi" || ki)`

2. Bob chooses a random puzzle `j` and solves it. Obtains `(xj, kj)`.
3. Bob sends `xj` to Alice
4. Alice looks up puzzle with number `xj` and now uses `kj` as a shared secret.

Each puzzle takes `n` to solve and `1` to create. So:

- Alice's work: `O(n)` (prepare `n` puzzles)
- Bob's work `O(n)` (solve 1 puzzle)
- Attacker work: `O(n**2)` (solve `n` puzzles)

This is pretty bad. Transmitting `2**32` puzzles gives only `2*64` security!

Can we achieve a better gap using general symmetric cipher?: Unknown... but roughly speaking, quadratic gap is the best we can get.

Diffie-Hellman

Goal: exchange key without a Trusted 3rd Parties and also achieve an *exponential gap* between the work done by Bob / Alice and that done by an attacker trying to eavesdrop (unlike Merkle Puzzles, which achieves a *linear gap*)

Algorithm:

1. Fix a large prime `p` (around 600 digits or `2000` bits or `2Kb` of data)
2. Fix an integer `g` in `{1, ..., p}`

This is done *once* and then those are fixed. Then...

1. Alice chooses a random `a` in `{1, ..., p-1}`.
2. Alice computes `A = g**a (mod p)` and sends that to Bob
3. Bob chooses random `b` in `{1, ..., p-1}`
4. Bob computes `B = g**b (mod p)` and sends that to Alice
5. Now the shared key is that `Kab = g**(a*b) (mod p)`

Note that both parties can compute `Kab`!

1. Alice does `B**a (mod p) = (g**b)**a (mod p) = g**(a*b) (mod p)`
2. Bob does `A**b (mod p) = (g**a)**b (mod p) = g**(b*a) (mod p)`

Note: we will see why computing `g**a (mod p)` and `g**b (mod p)` is very efficient later.

Security of Diffie-Hellman

We will see this in depth later. But let's use our intuition:

- Eavesdropper sees `p, g, A=g**a (mod p), B=g**b(mod p)`
- Can the eavesdropper compute `g**(a*b)`?

More specifically, we define `DHg(g**a, g**b) = g**(a*b) (mod p)`. How hard is this DH function?

Suppose prime `p` is n bits long. The best known algorithm to compute this function is called General Number Field Sieve with a run time of `exp(O(n**(1/3)) = e**(O(n**(1/3))` (simplified)

This is not really exponential, because we take the cubic root of `n`! So the problem is difficult, but not *that* hard.

Examples:

- For a 1024 bits prime, we get the solution in `e**(1024**(1/3)) ~= e**10`! Very small (again, simplified... it ends up being `e**80`... but the point still stands)

For comparison, compare the key size of a cipher with the prime size (note again this is very simplified).

cipher key size	modulus size
80 bits	1024 bits
128 bits	3072 bits
256 bits (AES)	**15360** bits

So if you want to use Diffie-Hellman to exchange keys of a certain size, you should take this into account, so that both are comparable.

Note: using `15360 bits` is quite problematic! We can't use that in practice.

Can we do better? Yes. This is the approach taken originally, using arithmetic modulus of primes. We can translate this approach to another algebraic object... Elliptic Curves! And in that world, we get that the problem is much much harder.

cipher key size	modulus size	Elliptic Curve size
80 bits	1024 bits	160 bits
128 bits	3072 bits	256 bits
256 bits (AES)	**15360** bits	512 bits

Note: all of this is insecure against an active attack! We will see how to make it secure next week, for now, let's see how it is vulnerable.

Suppose you have a man in the middle. The MiTM is gonna intercept `A = g**a` and will change it to `A' = g**a'`. Now Bob receives `A'`, computes `B = g**b` and again the MiTM sends to Alice `B' = g**b'`.

Now Alice compute `g**ab'` as a shared key and Bob computes `g**ba'`, and the MiTM actually knows both. So now the MiTM just acts as a relayer, reading / modifying every message sent.

Property: non-interactivity

Diffie-Hellman can be viewed as a **non-interactive protocol**. Everyone can post their `g**a, g**b, g**c` and such... and then whenever `A` wants to connect with `C`, they just go to the post and establish it! No further communication is needed.

Open problem: can we achieve **non-interactivity** for more than two persons? So I want `A` to read only from `B, C, D` and compute a shared key for all four of them!

For `n=3`, we can do it with a protocol called Protocol du Joux

For `n=4` and bigger, we just don't know.

Public Key Encryption

Again: we consider only security against eavesdropping.

Now we will see an approach based on **public key encryption**:

- The encryption algorithm `E` is given a public key `Pk_bob`
- The decryption algorithm `D` is given a secret key `Sk_bob`

Now, more formally: public-key encryption is a system of three algorithms: `G, E, D`

- `G()` a randomized algorithm outputs a key pair `(pk, sk)`
- `E(pk, m)` a randomized algorithm that takes `m` and outputs `c`.
- `D(sk, c)` a deterministic algorithm that takes `c` and outputs `m` or `{!}`

Semantic Security of Public Key Encryption

The game now is a little different.

1. Challenger creates `pk` and gives it to adversary
2. Adversary sends two messages `m0, m1`
3. Challenger sends `c <- E(pk, mb)`
4. Adversary must guess if `mb` is `m0` or `m1`

As always: `G,E,D` is secure if attacker cannot distinguish between `EXP(0)` and `EXP(1)`

Note that we don't need to explicitly give the attacker power to do a Chosen Plaintext Attack: this is inherent to the system, the adversary can do as many encryptions as they want using `pk` , which is given to them!

Establishing a shared secret

1. Alice `(pk, sk) <- G()`
2. Alice sends to Bob the `pk`
3. Bob chooses a random `x` between `{0,1} ** 128`
4. Bob sends `c = E(pk, x)`
5. Alice decrypts `c`
6. Alice and Bob use `x` as a shared secret

Note: this is very different from Diffie-Hellman, because here Bob and Alice must take turns! Bob cannot send `E(pk, x)` before Alice generated her `pk` and sent! In Diffie-Hellman, the messages could be sent at arbitrary times.

This gives a nice property: we can just post our public keys and everyone can sends us encrypted messages!

Man in the middle attack

Again, this scheme is secure only against passive eavesdroppers.

1. Alice generates `(pk, sk) <- G()`

2. Eve changes that for `(pk', sk')` and sends that to Bob.
3. Bob picks `x` in `{0,1}**128`
4. Bob sends `E(pk', x)`
5. Eve again acts as a relayer and sends correct message `E(pk, x)` to Alice to exchange goes correctly (she decrypts with `pk'` and encrypts again with `pk`).

We will review some number theory so as to understand correctly public key encryption.

Notation

- `N` is a positive integer
- `p` is a prime number
- `Zn` is `{0, 1, ... n-1}`

Note we can do addition and multiplication modulo of `N`.

- `gcd(x,y)` is the greatest common divisor of `x,y`

Fact: for all `x,y`, there exists `a,b` such that `a*x + b*y = gcd(x,y)`. Read: `gcd(x,y)` is a linear combination of `x,y` with `a,b`.

Example: `gcd(12,18) = 2 * 12 - 1 * 18 = 6`

These can be found using the Euclidian Algorithm. The time of this is roughly quadratic in respect to `log(N)`.

- `gcd(x,y) = 1` then `x,y` are coprime

Modular inversion: over `R`, the inverse of `2` is `1/2`. In `Zn`, the inverse of `x` is `y` such that `x*y = 1 (mod n)`. `y` is denoted `x**-1`.

Example: let `N` be an odd integer, what is the inverse of `2` in `Zn`? The answer is `(N+1)/2` (`2 * (N+1)/2 = N+1` so this is the inverse)

Lemma: `x` in `Zn` has an inverse if and only if `gcd(x,N) = 1`

- `Zn*` is the set of invertible elements in `Zn`.

Note: for `Zp`, `Zp - {0} = Zp*`. Also: `|Zp*| = p -1`

So we know how to solve **modular linear equations**:

For `a * x + b = 0` in `Zn`, the solution is: `x = -b * a**-1`.

Use the Euclidian Algorithm to find `a**-1`

Fermat & Euler's Theorem

See Fermat Theorem and Euler Theorem

Modular e'th Roots

We know how to solve modular linear equations: `a * x + b = 0` in `Zn`. This is done be calculating `x = -b * a**-1`, using Euclid's Algorithm to get `a**-1`

Now we want to solve higher degree polynomials: `x**2 - c = 0` or `z**37 - c = 0` all in `Zp`

Let `p` be a prime and `c` exists in `Zp` o

Definition: `x in Zp / x**e = c in Zp is called an eth root of c`

Examples:

- `7**(1/3) = 6 in Z(11)` because `6**3 = 7` in `Z(11)`
- `3**(1/2) = 5 in Z(11)` because `5**2 = 25 = 3` in `Z(11)`
- `1**(1/3) = 1 in Z(11)` because `1**3 = 1` in `Z(11)`

Not all roots exists: `2**(1/2)` does not exists modulo 11.

When does it exists?

The easy case : `gcd(e, p-1) = 1`, then for all c in `(Zp)*: c**(1/e) exists` and is easy to find.

Algorithm: Call `d` the inverse modulo `p-1`. Then: `d = e**-1` in `Z(p-1)`. We can construct `c**(1/e) = c**d`

`in Zp`.

Proof: As `d * e = 1 in Zp-1` (because it is the inverse), then there exists `k` such that `d * e = k(p-1) + 1 (mod Zp)` (note this is basically the definition).

So we now do `(c**d)**e`, which should be `c` if our algorithm holds, because `d = e**-1`.

Now check that `(c**d)**e = c**(d*e) = c**(k(p-1)+1) = ((c**(p-1))**k).c`. Now note that `c**p-1 = 1` by Fermat Theorem (`n**p-1 = 1 (mod p)`). So now we have `(1**k) * c (mod Zp) = c (mod Zp)`. We got `c` back! That means `c**d` was our root.

The hard case: with `gcd(e, p-1) != 1`. Example, `e=2`. Note that for all `p`, `gcd(2, p-1) != 1`.

Fact: in `(Zp)*`, `x -> x**2` is a 2-to-1 function. See an example in `Z(11)*`, `2**2` and `9**2` both map to `4`. This is valid for all elements in `Z(11)*`.

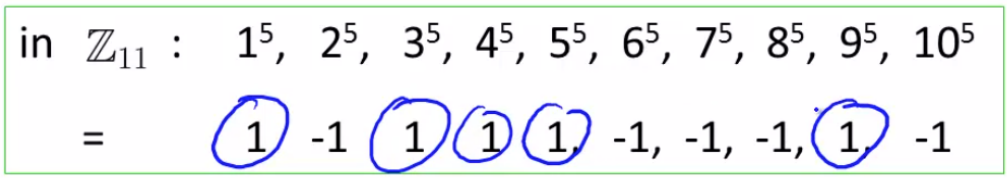
Definition: `x` in `Zp` is a quadratic residue if it has a square root in `Zp`.

Note: if `p` is an odd prime, the number of quadratic residues is `(p-1)/2 + 1`. Exactly half of the elements in `p-1` are gonna have a quadratic residue, because `x**2` maps 2 to only 1, and then we add `zero`!

Euler's Theorem (another one: deciding if an element has a quadratic residue)

Theorem: `x` in `(Zp)*` is a quadratic residue iff `x**((p-1)/2) = 1 (mod Zp)`, with `p` an odd prime.

Example:



`x**((p-1)/2)` is called the `Legendrey Symbol` of `x` over `p`.

Note that this is not a constructive argument, we need to compute our roots anyway...

Computing square roots modulo p

Suppose `p = 3 (mod 4)`.

Lemma: if `c` in `Zp` is a quadratic residue, then `c**(1/2) = c**((p+1)/4) in Zp`.

Proof: `c**((p+1)/4)**2)` (note we should get `c` back...). Now this is `c**((p+1)/2) = c**((p-1)/2) * c`. And now we know that because `c` is a quadratic residue, so `c**((p-1)/2) = 1`, so we get that this is equivalent to `c`!

What we p = 1 (mod 4)? Then we can't do it like that, because our formula gets a rational fraction on the exponent. But we can do it... although it is quite harder and there's a randomized algorithm to do it (related: Extended Riemman Hypothesis)

The run time of all this is `O(log3 p)`.

Solving quadratic equations mod p

So now we can find the solutions for things like `a * x**2 + b * x + c = 0`. Solution: `x = (-b += sqrt(b**2 - 4ac) /2a in Zp`.

We can find `(2a)**-1` using extended Euclid. Find `b**2 - 4ac` using the square root algorithm.

Computing eth roots mod N

Let `N` be a composite number and `e > 1`. Computing this is **as hard** as factoring `N` (as far as we know, this is the best algorithm we have...).

Arithmetic Algorithms

Representing bignums

You put them into buckets. If you have a 64-bit machine, you use 32-bit buckets for `A*B` and `A+B` still fit into a word of your machine.

Now, given two n-bit integers:

- **Subtraction**: `O(n)`
- **Multiplication**: naively `O(n**2)`. But there's a great algorithm (Karatsuba's Algorithm) that achieves `O(n**1.585)`. And there's an even better one that achieves `O(n * log(n))`, but this is used only for *enormous* algorithms.
- **Division with remainder**: `O(n**2)`
- **Exponentiation**: `O( (log x) ** n**2) <= O(n**3)` . See Complexity of exponentiation.

Note: for simplicity, we will assume multiplication takes `O(n**2)`.

Complexity of exponentiation

Take a finite Cyclic Group `G` (for example, `G=Zp*)`).
Goal: given `g` and `x` compute `g**x`.

Naively, you may want to just multiply `g` by `g` `x` times. But our `x` are enormous!

So what we do is *repeatedly square*. Suppose `x=53 = 0b110101 = 32+16+4+1`. Then: `g**53 = g** (32+16+4+1) = g**32 * g**16 ** g**4 ** g`. And now we start just computing the squares of `g`: `{g, g**2, g**4, g**8, g**16, g**32}` ... and using those

results we can get `g**53`. This took us only 5 squares and 4 multiplications.

```
def exp(g: bits, x: bits):
        y = g
        z = 1
        for i in (0, n):
                if x[i] == 1:
                        z = z*y
                y = y**2
        return z
```

The complexity of this algorithm is `log2(x)`

Hard problems

There are many easy problems...

- Given `N` and `x` in `Zn`, find `x**-1` in `Zn` (use Euclidian Algorithm
- Given `p` and polynomial `f(x)` in `Zp[X]`, find `x` st `f(x) = 0` in `Zp` (if one exists). This takes linear time with respect to the degree of `f`.

But many problems in modular arithmetic are difficult, and they form the basis for Assymetric Cryptography.

Hard problems with primes

Discrete Log problem

Fix a prime `p>2` and `g` in `Z(p)*` of order `q`. Consider `x -> g**x (Zp)`. Note that using the algorithm described in Complexity of exponentiation this is very easy.

But what about the inverse function? Given `g**x`, I want `x`. This is a Logarithm, in particular the Discrete Log problem: `Dlog_g(g**x) = x`, where `x` in `{0, ..., q-2}`.

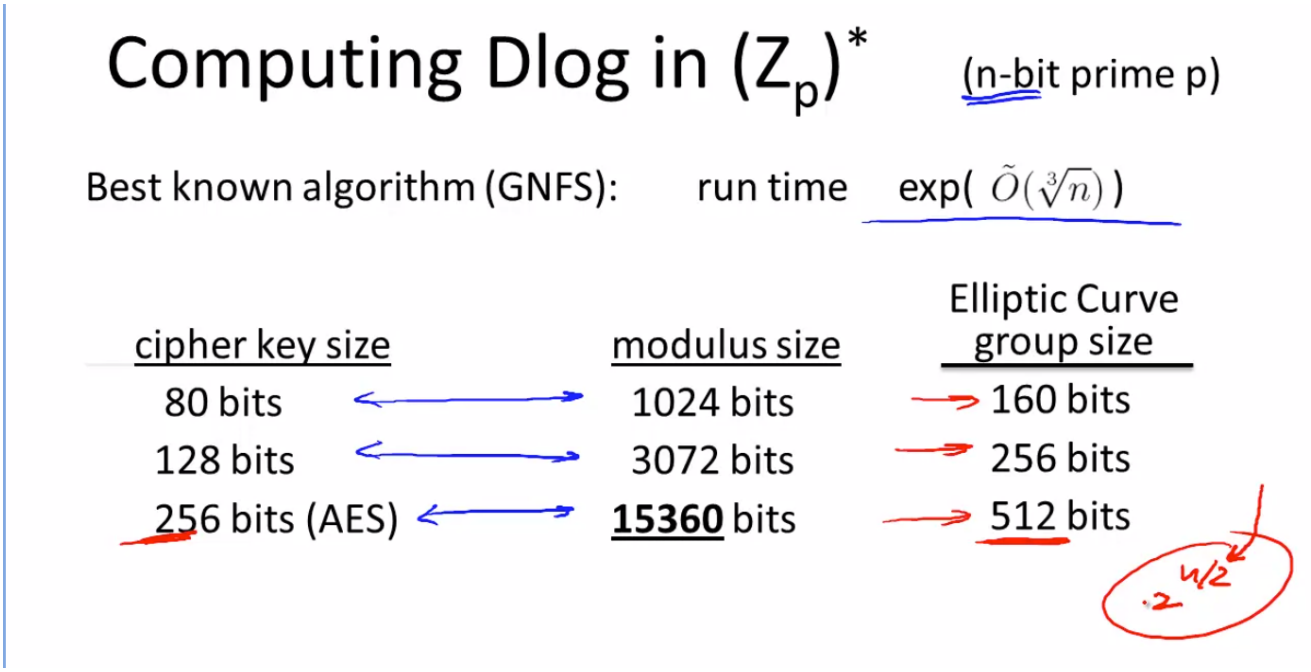
Example: for `Z(11)`.

in Z11	Dlog2
1	0
2	1
3	8
4	2
5	4
6	9
7	7
8	3
9	6
10	5

Example, with `5`: take `2**4 = 16 = 5 (mod 11)`. So `2**4 = 5`, so `log2(5) = 4`.

See the Discrete Log problem page for more details.

Note: for a given prime `p`, the problem is way harder in Elliptic Curves modulo `p` than in `(Zp)*`.



We mentioned that this is useful to prove that hash functions are collision resistant. See Message Integrity > Provable compression functions.

Hard problems with composites

Consider `n=1024` and `Z_2(n) = {N = p*q, wuere p*q are n-bit primes}`.

Problem 1: Factor a random `N` in `Z_2(n)`. For `n=1024` this is hard to do but not hard enough nowadays, use `n=2048`.

Problem 2: given polynomial `f(x)` where `deg(f) > 1`, and a random `N` in `Z(2)(n)`, find `x` in `Zn` such that

`f(x)=0`. We know how to do this for `deg(f) = 1`, but as soon as the degree is bigger it is really hard. RSA uses this.

The factoring problem

Old as the world. The Greeks already were aware of it, and Gauss worked on it to.

How to distinguish a prime from a composite, and how to factor the composite into their primes?

Distinguishing has been solved! But factoring is still hard :(

The best algorithm is the Number Field Sieve, with `exp(O(n**1/3))`.

The current world record is factoring a 232 digits, and it took two years of hundred of machines. Factoring a 2014-bit integer would take ten years longer!

Test

1. Alice to TTP. TTP generated K_ABC. Tickets are encrypted.
2. `(g**(3xy), g**(2xy)`, `g**(x(y+1))`
3. `g=a/b, Alice computs B**a, Bob A**(1/b)`
4. Not safe
5. Have to do euclidian algorithm

6. `3x+2=7 (mod 19) = 8`
7. Elements in `Z(35)* = 24 = fi(7 * 5) = (7-1) * (5-1)`
8. `2**10001 (mod 11) = 2`
9. `2**245 (35) = 32`
10. `ord_35(2) = 12`
11. Generators: only `7` and `6`.
12. Have to use cuadratic formula
13. `11 roth of 2 in Z(19) = 13`
14. `Dlog2(5)` in `Z(13) = 5`
15. `fi(p-1)`

What is public key encryption?

In a public key encryption scheme, there's a pair of keys: pk (*public key*) and sk (*secret key*) and $E(m, pk) = c$, $D(c, sk) = m$; that is: encryption takes the public key and decryption the secret key

Applications

- **Session setup**: as we saw in Key Exchange.

Lot's of non interactive applications, like:

- Email: Bob sends mail to Alice encrypting with Alice's public key.

- Disk Encryption: Bob encrypts his disk using a symmetric key, encrypts the symmetric key with his secret key. If Bob wants to give Alice access, he encrypts the symmetric key with Alice's public key.
- Key Escrow: data recovery without Bob's key by encrypting everything Bob encrypts with Bob's public key and an additional escrow key, which the escrow always can give the company to recover Bob's data.
- Note here that Bob needs Alice's public key, so here enters Public Key Management.

Formal Definition

A public key encryption system is a triple of algorithms:

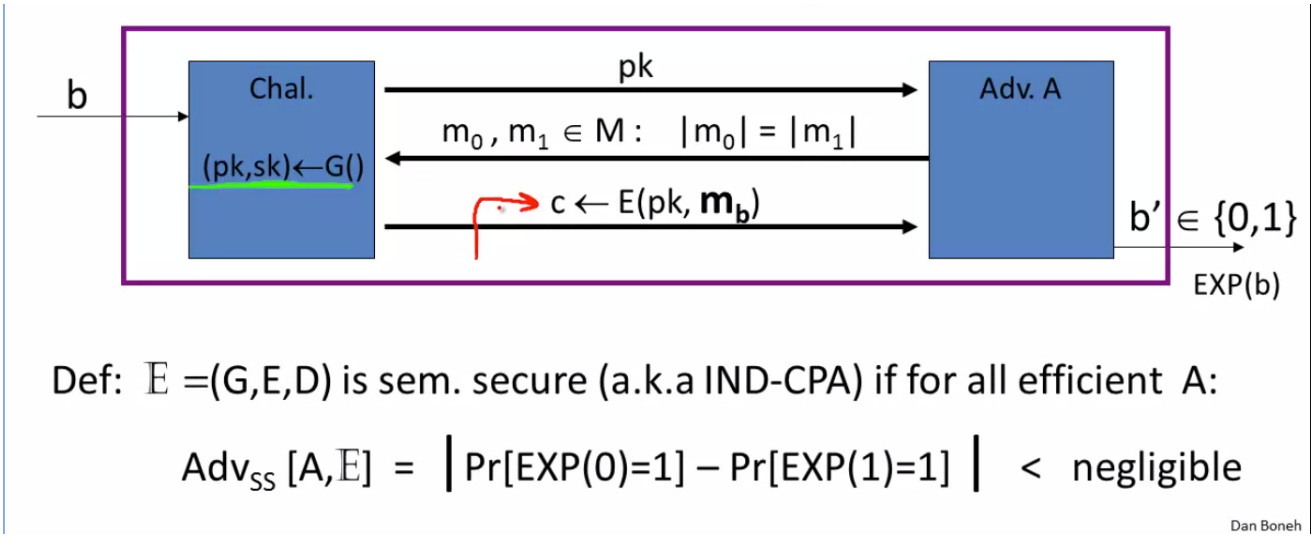
- $G()$: a randomized algorithm that outputs a key pair. It is defined as having no parameters, but in reality would take the size of the key pair.
- $E(pk, m)$: randomized algorithm that takes m and outputs c
- $D(sk, m)$: deterministic algorithm that takes c and output m or $\perp$

And of course:
$\forall (sk, pk) \leftarrow G(), \forall m \in M, D(sk, E(pk, m)) = m$

Security

Security against eavesdropping

For $b = 0, 1$, we define the experiments $EXP(0)$ and $EXP(1)$.



Relation to symmetric cipher security

For symmetric ciphers, we had:

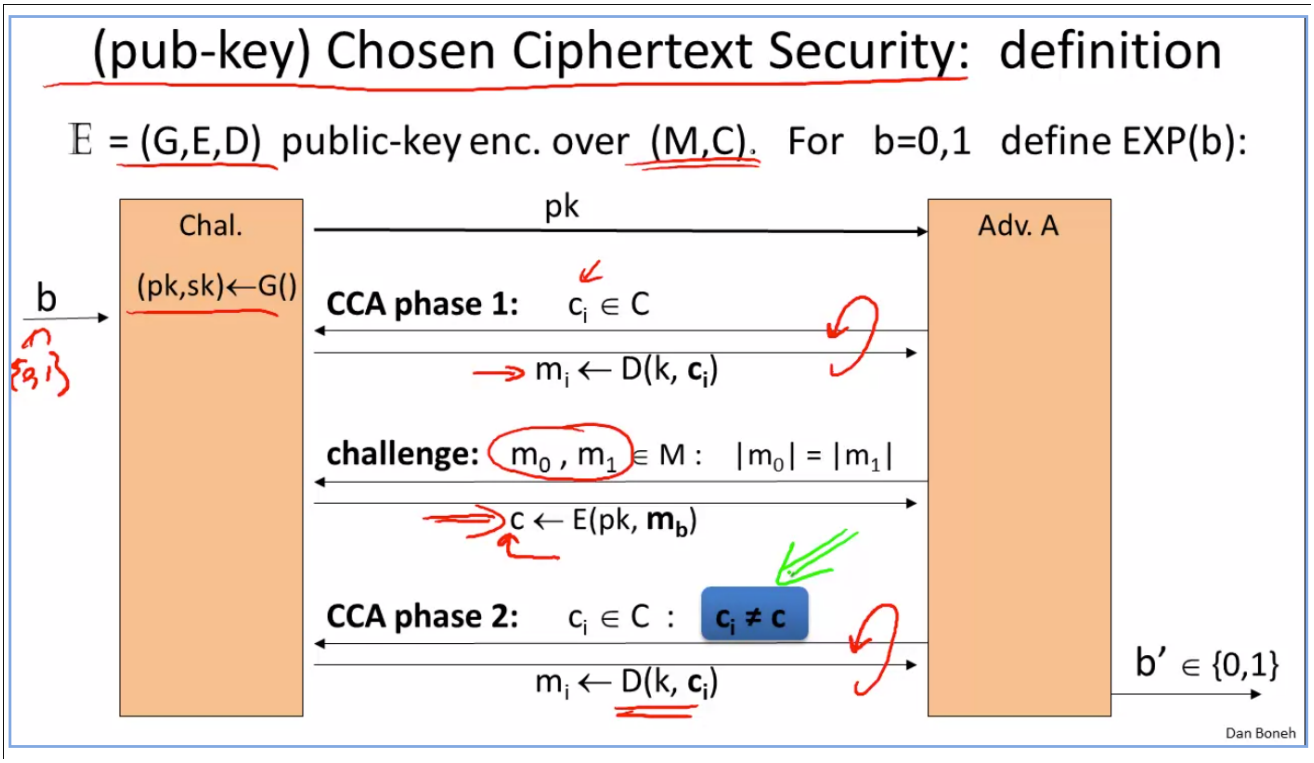
- **one time security** and **many-time security** (Authenticated Encryption > Chosen ciphertext security)

For public key encryption, **one time security** is more or less the same than **many-time security**. This follows from the fact that E takes a public key as an input, so the attacker can encrypt as many messages as they want.

Security against active attacks

To define this we define Chosen Ciphertext Security. Take $E = (G, E, D)$ a public encryption over (M, C). For $b = 0, 1$ define $EXP(b)$:

1. The challenger $Chall$ generates $(pk, sk) \leftarrow G()$ and gives that to adversary A.
2. Now A sends $c_i \in C$ to challenger, and $Chall$ replies with $m_i \leftarrow D(k, c_i)$. A can do this as long as he pleases. This is called *phase 1*.
3. Now comes the challenge: A sends $m_0, m_1 \in M : |m_0| = |m_1|$ and receives $c \leftarrow E(pk, m_b)$
4. Now, A can continue their queries: they can send $c_i \in C, c_i \neq c$ and receive $m_i \leftarrow D(k, c_i)$. This is called *phase 2*.
5. Now A should not be able to tell whether he was given then encryption of m_0 or m_1, even though they were able to query for decryption of an arbitrary number of c!



So, E is **CCA Secure** (also referred to as IND-CCA, indistinguishable chosen ciphertext security) if for all efficient A:

$Adv_{CCA} = |Pr[EXP(0) = 1 - Pr[EXP(1) = 1]|$ is negligible.

Public Key Encryption from Trapdoor Permutations

Introduction to Trapdoor Functions

Also known as **TDF**.

Definition: a trapdoor function $X \rightarrow Y$ is a triple of efficient algorithms (G, F, F^{-1}).

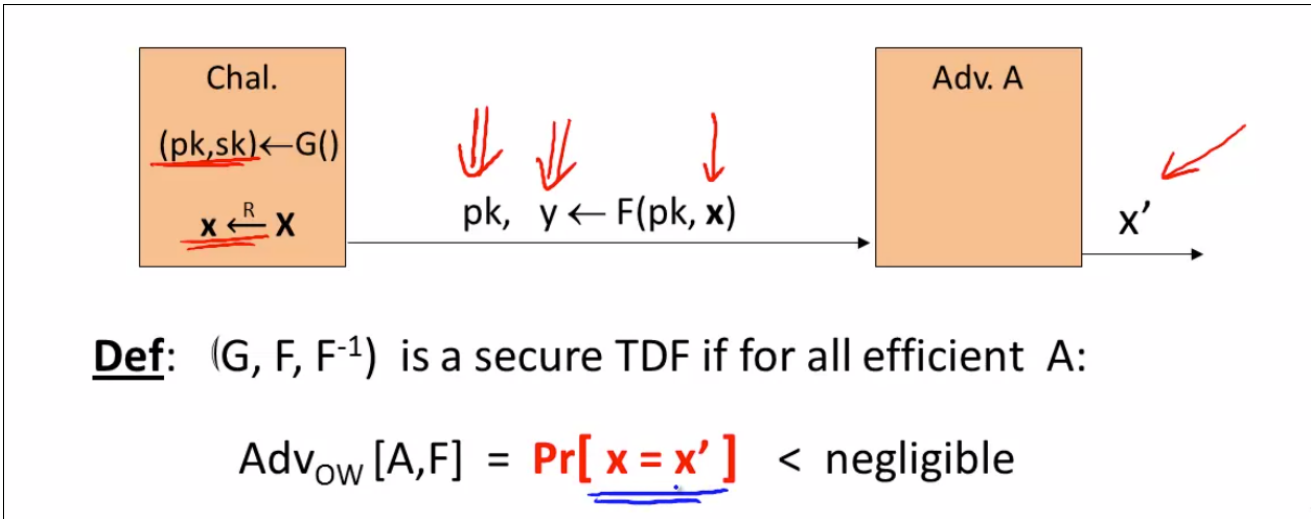
- $G()$ a randomized alg. which outputs a keypair (sk, pk)

- $F(pk, \cdot)$ a deterministic algorithm that defines a function $X \to Y$
- $F^{-1}(sk, \cdot)$ defines a function $Y \to X$ that inverts $F(pk, \cdot)$

More precisely,
$\forall (pk, sk) \leftarrow G, \forall x \in X : F^{-1}(sk, F(pk, x)) = x$

Security of Trapdoor Functions

(G, F, F^{-1}) is secure if $F(pk, \cdot)$ is a *one-way* function: it can be evaluated, but cannot be inverted without sk.



Public Key Encryption from Trapdoor Functions

Take:

- (G, F, F^{-1}) a secure trapdoor function $X \to Y$
- (E_s, D_s) a symmetric authenticated encryption defined over (K, M, C)
- $H : X \to K$ a Hash Function.

Now, for **encryption**:

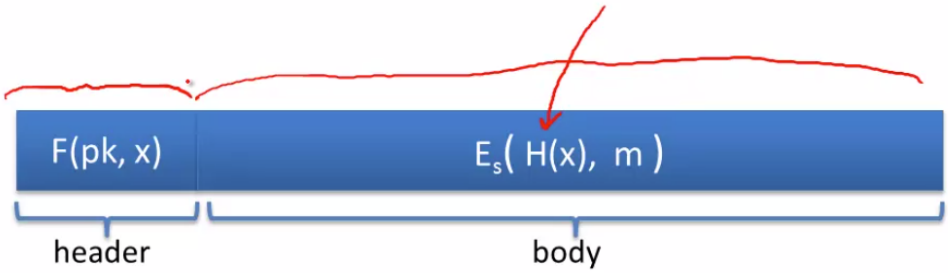
1. Take random $x \leftarrow X$
2. Apply F the trapdoor to x: $y \leftarrow F(pk, x)$
3. Apply H our hash to x: $k \leftarrow H(x)$
4. Get the ciphertext by encrypting with our symm. encryption: $c \leftarrow E_s(k, m)$
5. Output (y, c)

Note that the trapdoor function is only applied to x. m itself is only encrypted with the symmetric cypher.

For **decryption**:

1. Calculate x by doing $F^{-1}(sk, y)$
2. Get $k \leftarrow H(x)$
3. Get $m \leftarrow D_s(k, c)$
4. Output m

In pictures:



Theorem: if (G, F, F^{-1}) is a secure trapdoor, (E_s, D_s) provide authenticated encryption and H behaves as a random oracle, then (G, E, D) is Chosen Ciphertext Attack-secure.

This mode of encryption is an ISO Standard.

Warning: incorrect use of a trapdoor function

Never encrypt by applying F directly to plaintext!

Problems:

1. It is totally deterministic, so it cannot have Stream Ciphers > Semantic security.
2. Many other attacks exists.

RSA

As with Trapdoor Functions, we just have (G, F, F^{-1}).
We will adjust F here and specify that it is a *permutation*: so $F(pk, \cdot) : X \to X$.

RSA History and uses

First published in Scientific American in 1997. RSA stands for Rivest, Shamir and Adleman.

Very widely used:

- SSL: certificates and key exchange
- Secure email and file systems
- etc, etc

Review: Number Theory

Let $N = pq$ where p, q are prime.

- $Z_n = \{0, 1, 2, 3 \ldots N-1\}$
- $(Z_n)^*$ are the *invertible elements* in Z_n

Fact: $x \in Z_n$ is invertible $\iff$ $gcd(x, N) = 1$ **Fact**: $|(Z_n)^*| = \phi(N) = (p-q)(q-q) = N - p - q + 1$ **Euler Theorem**: $\forall x \in (Z_n)^* : x^{\phi(N)} = 1$ **Fact**: p, q are in the order of $\sqrt{N}$, and because $\phi(N) = N - p - q + 1$, $\phi(N) \simeq N - 2\sqrt{n} \simeq N$. This means that if we choose a random element in Z_n, it is *very likely* that it will be in $\phi(N)$, so it is very likely that it will be invertible.

RSA Permutation definition per the ISO Standard

Now, G is defined as follows:

- choose two random primes $p, q \simeq 1024\ bits$.
- Set $N = pq$.
- Choose $e, d : ed \equiv 1 \pmod{\phi(N)}$.
- Output $(pk = (N, e), sk = (N, d))$. e is sometimes called the *encryption exponent*, and d the *decryption exponent*

Now, F is called RSA and is defined as follows:

- $RSA(pk, x) : (Z_n)^* \to (Z_n)^*; RSA(x) = x^e \pmod{Z_n}$
- $RSA^{-1}(sk, y) = y^d$

Check that the inverse actually behaves as the inverse:

$y^d = RSA(x)^d$ $RSA(x)^d = (x^e)^d = x^{ed}$ $x^{ed} = x^{k\phi(N)+1}$ (
$ed \equiv 1 \pmod{\phi(N)} \implies \exists k : k.\phi(N) + 1 = ed$)
$x^{k\phi(N)+1} = (x^{\phi(N)})^k.x$
$x^{\phi(N)} = 1 (Euler\ Theorem) \implies (x^{\phi(N)})^k.x = 1^k.x = x$

As x was our original input to RSA, RSA^{-1} is indeed the inverse.

RSA Security

The claim is that for all efficient algorithms A:

$$Pr[A(N, e, y) = y^{1/e}] < negligible$$

where $p, q \leftarrow n - bit\ primes$, $N \leftarrow pq$ and $y \leftarrow Z_n*$

This is an *assumption* and is called the *RSA Assumption*.

RSA Public Key Encryption

Take (E_d, D_s) symm encryption scheme with authenticated encryption and $H : Z_n \to K$ where K is the key space of (E_d, D_s)

- $G()$ generates the RSA params
  $pk = (N, e), sk = (N, d)$
- $E(pk, m)$: Choose random x in Z_n,
  $y \leftarrow RSA(x) = x^e, k \leftarrow H(x)$, output $(y, E_s(k, m))$
- $D(sk, (y, c)) = D_s(H(RSA^{-1}(y)), c)$

How Not to Use RSA: Textbook RSA

The *textbook* implementation of RSA is insecure: encrypting directly with the trapdoor function is insecure. As we have seen in Warning incorrect use of a trapdoor function, this produces a deterministic cipher, which can't be semantically secure.

But even more, the trapdoor is just a function! Is not an encryption scheme. Let's see what goes wrong if we use RSA directly using SSL as an implementation.

Quick SSL review:

1. Browser sends `CLIENT HELLO`
2. Server responds `SERVER HELLO (e, N)`
3. Browser sends `c = RSA(k)`

If we do this and directly use RSA to encrypt k, what happens?

Suppose k is 64-bits. Eve sees $c = k^e \pmod{Z_n}$ If k factors into a product of roughly equal size numbers: $k = k_1 . k_2$, where $k1, k2 < 2^{34}$. This happens roughly 20% of the times.

But then:

$$c = k_1^e . k_2^e \implies \frac{c}{k_1^e} = k_2^e \pmod{Z_n}$$

And now I can do a Meet in the middle attack on this.

1. Build a table $c/1^e, c/2^e \ldots c/2^{34e}$ (time: 2^{34})
2. For $k_2 = 0..2^{34}$, test if k_2^e is in table (time: 2^{34})

Output the matching (k_1, k_2). Total attack time: $\sim 2^{40} << 2^{64}$.

RSA in Practice: PKCS1 v1.5

The ISO Standard for RSA is not really used in practice, but was described in RSA Permutation definition per the ISO Standard.

Of course we **don't** use the textbook implementation described in How Not to Use RSA Textbook RSA.

In practice, we use PKSC1 v1.5. The difference is that we don't generate a random x as a symmetric key, but instead the symmetric key is *given* to the RSA system, the RSA system performs some *preprocessing* to expand this key into the modulo size (for example, $128bits \rightarrow 2048bits$).

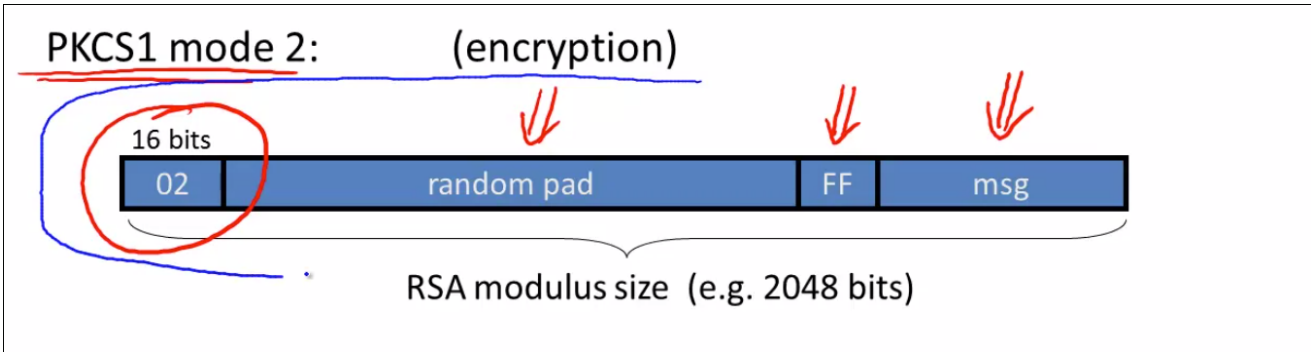
Questions:

- What is this preprocessing about?
- How do we know the system is secure?

For this analysis, we will analyze PKSC1 v1.5 **mode 2** (mode 1 is for signatures, mode 2 is for encryption).

So, in this schema, we will create a value x which will be encrypted with RSA:

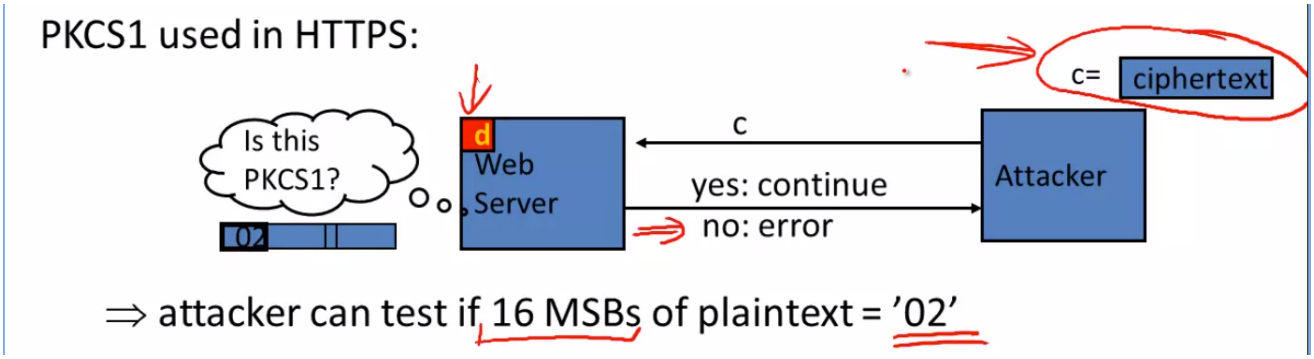
1. Take your msg (which most likely is a symmetric key) and put it as the least significant bits of the x you are creating.
2. Then you prepend $0xFF$.
3. You prepend a random pad without $0xFF$
4. You prepend $0x02$



That is the x you plug into RSA.

This was developed in the eighties and with no security proofs at the time! So of course there's a good attack on it.

Bleichenbacher Attack on PKCS1 v1.5



It should be clear that this scheme lets an attacker know whether a given ciphertext c starts with $0x02$ or not.

This is enough to completely decrypt a given ciphertext:

1. Choose $r \in Z_n$.
2. Compute $c' \leftarrow r^e.c = (r.PKCS1(m))^e$. The equality comes from just plugin r into $PKCS1$ and RSA. Remember $E_{RSA}(x) = x^e$.
3. Send c' to web server.

And by sending enough of these queries to the server (about 1M), you get $PKCS1(m)$!

Because this seems too magical for mortals, let's understand more deeply a toy alternative: enter **Baby Bleichenbacher**.

1. Attacker sends c
2. Web server decrypts c but instead of checking for $0x02$, the web server responds only whether $MSB(c) = 1$.

Now suppose N is a power of two (ie: $N = 2^n$). Note this is an invalid N in RSA! But let's use it.

So:

- Sending c reveals $MSB(c)$
- Sending $2^e.c = (2x)^e \pmod{Z_n}$. Note that $(2x)^e$ is shifting one left because we are working modulo 2^N. So we can now reveal the $MSB_2(c)$!

Continue doing this and you can reveal all of c...

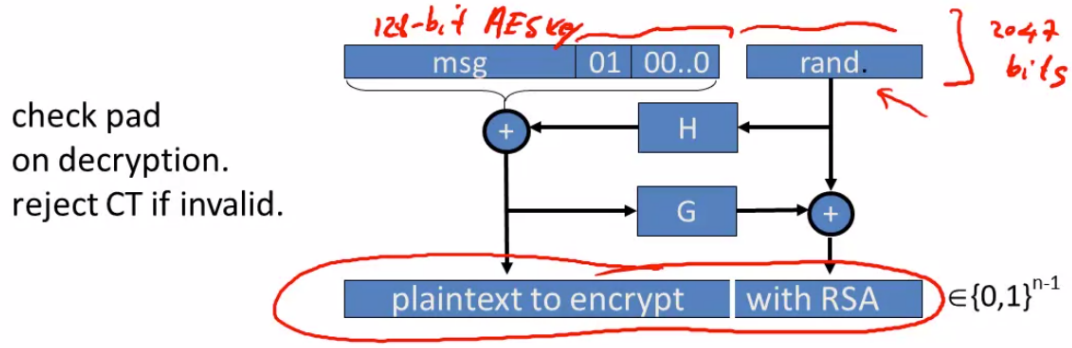
All SSL implementations were vulnerable to this...

The HTTPS Defense

This is described in **RFC 5246**, and it basically says that if a ciphertext does not start with $0x02$, then just set a *master secret* as a random 46 byte string.

RSA in Practice: PKCS1 v2.0

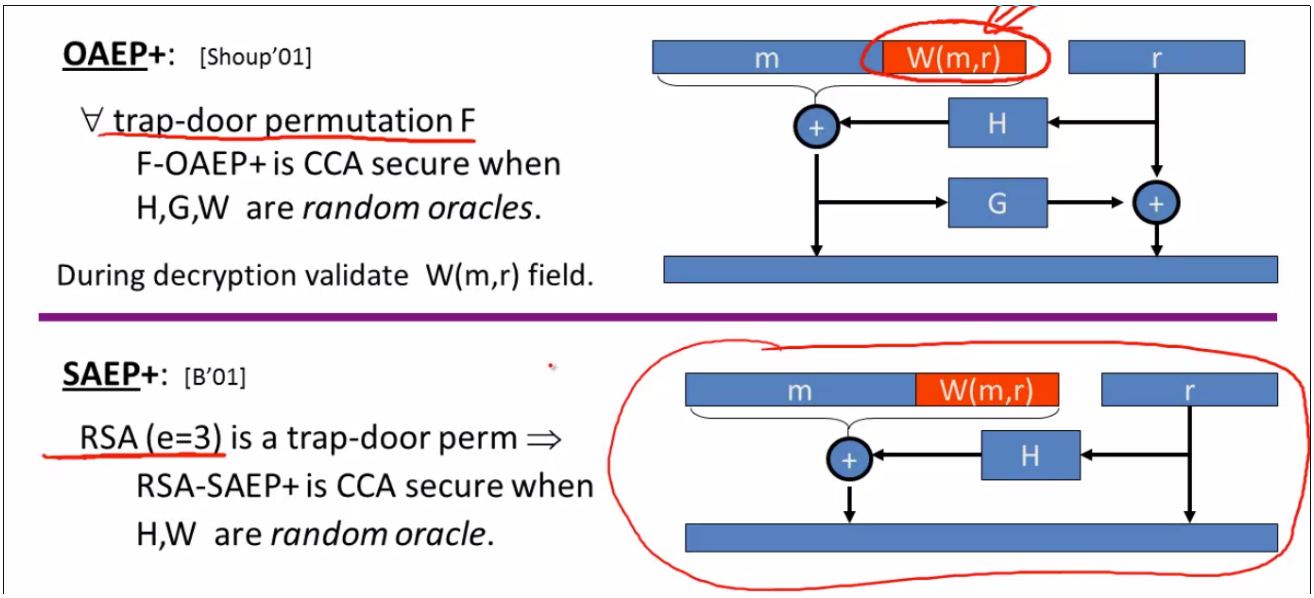
Also called **Optimal Asymmetric Encryption Padding** (OAEP), and it is an effort to improve PKSC v1.5 so it can be provable secure.



G and H are hash functions. The $0100..0$ at the right of msg is just a padding.

Theorem: RSA is a trapdoor permutation, then $RSA-OAEP$ is Chosen Ciphertext Attack-secure when H, G are random oracles.

Interestingly, the theorem is **only valid** of RSA! But there are improvements to make it work with any trapdor function: **OAEP+** and **SAEP+**, although these are not really used in practice.



RSA in Practice: Final tips

RSA With a Low Public exponent

You can choose a small e : $c = m^e \pmod N$

- $e = 3$ is the minimum value you can use
- The recommended value is $e = 65537 = 2^{16} + 1$ which takes only 17 multiplications

Asymmetry of RSA: is is very fast to encrypt, but slow to decrypt. Although there exists RSA-CRT which speeds up the decryption about 4x. In ElGamal this does not happen.

Key Length

Cipher key-size	RSA Modulus size
80 bits	1024 bits
128 bits ←	→ 3072 bits (2024)
256 bits (AES)	**15360** bits

Implementations attacks

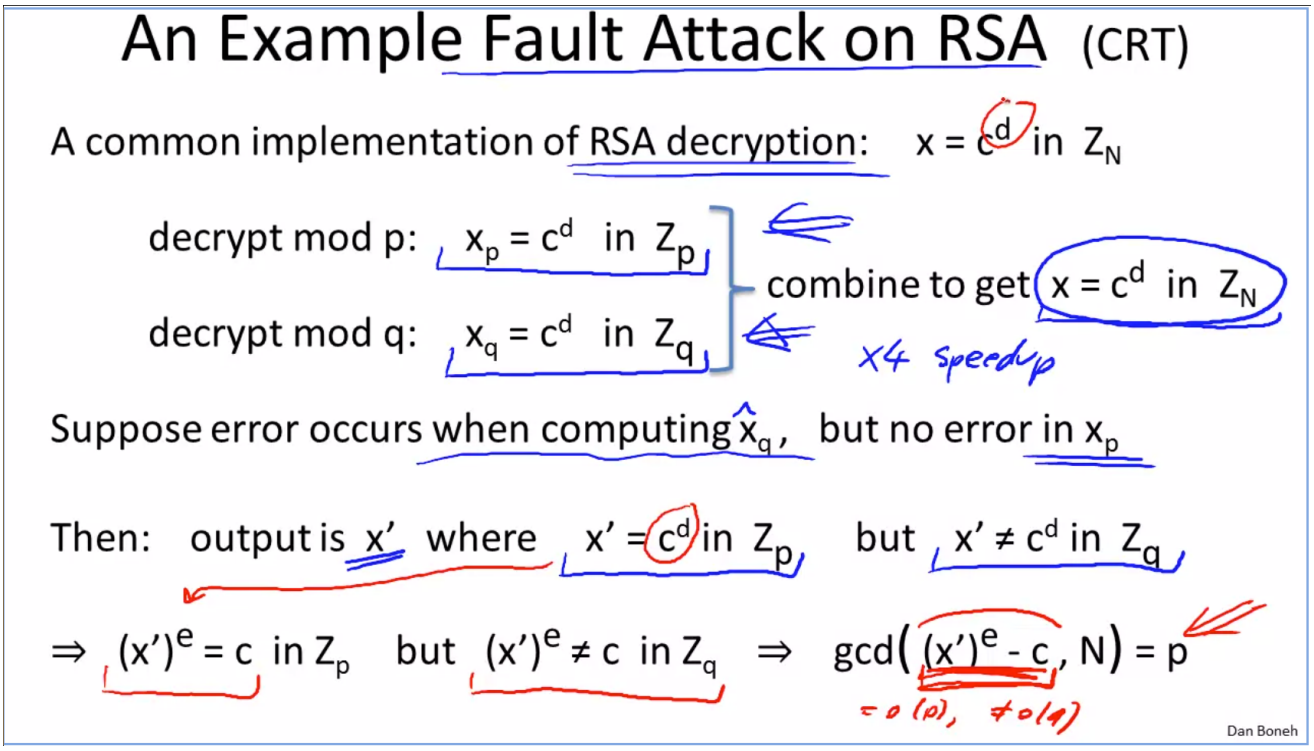
Attacks against mathematically correct RSA implementations.

- **Timing attack** (Kocher '97): the time it takes to compute $c^d \pmod N$ can expose d
- **Power attack** (Kocher '99): the power consumption of a smartcard while it is computing $c^d \pmod N$

can expose d

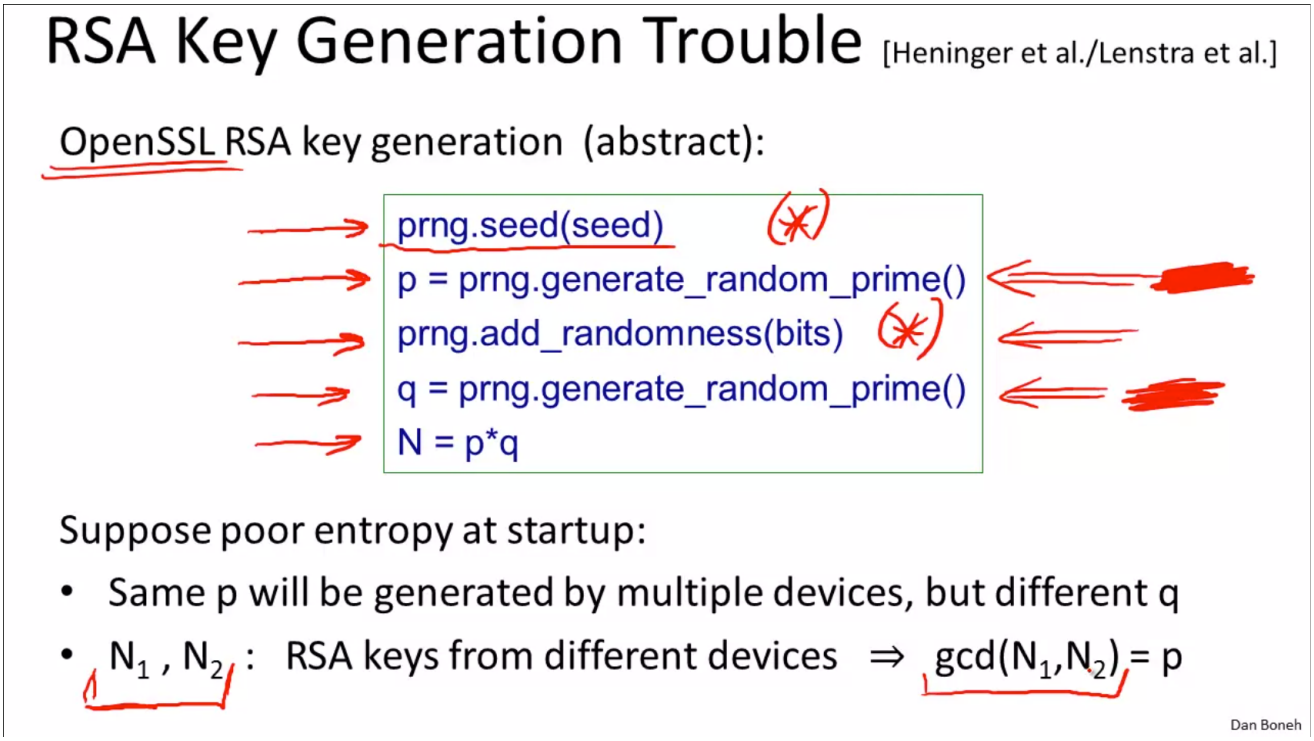
- **Faults attacks** (BDL '97): A computer error during $c^d \pmod N$ can expose d. This is significant enough as to crypto libraries double check the decrypted plaintext by encrypting it again and checking against the original. This introduces a 10% slowdown.
- **Key generation trouble**: poor entropy at key generation will lead to the same p be generated by multiple devices, but q will be different.

Example Fault Attack on RSA-CRT



Once you get p, you can derive d by computing $\phi(N)$, because you got the factorization of N.

Key generation trouble



If several devices generate the same p but different q, then N_1, N_2 from different devices will have a $gcd(N_1, N_2) = p$.

In practice, the researchers (Heninger et al. / Lenstra et al.), they were able to get 0.4% of public HTTPS keys!

Is RSA a one-way permutation?

If an attacker has $c = x^e \pmod N$, how difficult is to actually retrieve x?

The attacker knows the public key, which is basically (N, e).

So at the end of the day, we are asking how hard it is computing eth roots modulo a composite.

Our best known algorithm:

1. Factor N (this is very hard)
2. Compute e'th roots modulo p and q (easy)

Are there any shortcuts? **Must** we factor N? To prove that we can't, we should prove a reduction: $\exists$ algorithm for e'th root mod $N \implies$ efficient algorithm for factoring N.

This is not known. It is one of the oldest problem in public key cryptography. This is not known even for $e = 3$! But it is known for $e = 2$.

There *some* **very** weak evidence that no such reduction exists...

How not to improve RSA's performance

To speed up RSA decryption use a small private key d ($d \sim 2^{128}$). It is not possible to do a Brute Search on 128 bits, and this would be good because the exponentiation algorithm runs on linear time in relation to the size of d: $c^d = m \pmod N$

Terrible idea. There's an attacker by Wiener (published in '87), if $d < N^{0.25}$, then RSA is insecure in the worst sense: we can recover the private key.

So what if we set $d \sim 2^{512}$... no, there's an extension to Wiener attacker published in '98 that shows that if $d < N^{0.292}$, then RSA is insecure.

The conjecture is that this is true up to $N^{0.5}$, but no one has been able to prove this.

Wiener's Attack on RSA

Attacker is given: (N, e) and we want to recover d. We know that $d < \sqrt[4]{n}/3$

Recall:
$e.d = 1 \pmod{\phi(N)} \implies \exists k \in Z : e.d = k.\phi(N) + 1$

So now, we will manipulate that equation:

$$e.d = k.\phi(N) + 1$$

$$\frac{e}{\phi(N)} = \frac{k}{d} + \frac{1}{d.\phi(N)} (\text{divide by } d.\phi(N))$$

$$\frac{e}{\phi(N)} - \frac{k}{d} = \frac{1}{d.\phi(N)}$$

$$|\frac{e}{\phi(N)} - \frac{k}{d}| = \frac{1}{d.\phi(N)} (\text{add absolute value})$$

Because $\phi(N) \simeq N$, we know that $\frac{1}{d.\phi(N)} < \frac{1}{\sqrt{N}}$.

Our objective is to get $\frac{k}{d}$! Once we get that, we can get d. To do that, we should try to figure out $\frac{e}{\phi(N)}$.

Now, we don't know $\frac{e}{\phi(N)}$ because we don't know $\phi(N)$. But we can approximate it: we know $\phi(N) \simeq N$. Then, $\frac{e}{\phi(N)} \simeq \frac{e}{N}$.

So let's see our error if we replace $\frac{e}{\phi(N)}$ for $\frac{e}{N}$:

$$\phi(N) = N - p - q + 1 \implies |N - \phi(N)| < p + q + 1$$

Because p, q are roughly half the size of N, we assert that they both are in the order of $sqrt(N)$, so we can assert that:

$$p + q \leq 3\sqrt{(N)}$$

Now, we will use the fact that we know that d is small:

$$d \leq \frac{N^{0.25}}{3}$$

Then, if we square everything, invert it and then multiply one side by half:

$$\frac{1}{2d^2} - \frac{1}{\sqrt{(N)}} \geq \frac{3}{\sqrt{(N)}}$$

So this implies:

$$|\frac{e}{N} - \frac{k}{d}| \leq |\frac{e}{n} - \frac{e}{\phi(N)}| + |\frac{e}{\phi(N)} - \frac{k}{d}|(\text{triangular inequalit}$$

On our first argument, we actually asserted that the last term is less than $\frac{1}{\sqrt{(N)}}$.

And for the other term, we can do common denominator $(n.\phi(N))$: so it ends up: $\frac{e3\sqrt{(N)}}{n.\phi(N)}$. And we know that

$3 < \phi(N)$, so if we get rid of e and $\phi(N)$, we get a bigger fraction: $\frac{e3\sqrt{(N)}}{n.\phi(N)} < \frac{3}{sqrt(N)}$.

But we also know that:

$$\frac{3}{\sqrt{(N)}} \leq \frac{1}{2d^2} - \frac{1}{\sqrt{(N)}}$$

Therefore:

$$|\frac{e}{n} - \frac{k}{d}| \leq |\frac{e}{n} - \frac{e}{\phi(N)}| + |\frac{e}{\phi(N)} - \frac{k}{d}| \leq \frac{1}{2d^2}$$

$$|\frac{e}{n} - \frac{k}{d}| \leq \frac{1}{2d^2}$$

There are **very** few fractions that match this inequality. You just have to find them, and try to find d from k/d.

Public Key Encryption from Diffie-Hellman Key Exchange

Also known as ElGamal Encryption . Our goal is Chosen Ciphertext Security.

Review: Diffie-Hellman Protocol.

Fix a finite cyclic group G (eg: $G = (Z_p)^*$) of order N and a generator $g \in G$ (ie: $G = \{1, g, g^2, \ldots, g^{n-1}\}$).

1. Alice chooses random a in $\{1, \ldots, n\}$
2. Alice sends $A = g^a$ to Bob

3. Bob chooses random b in $\{1, \ldots, n\}$
4. Bob sends $B = g^b$ to Alice
5. Now the shared secret
   $k_{ab} = A^b = (g^a)^b = g^{ab} = (g^b)^a = B^a$

The attacker can only see g, A, B. The attacker can't extract from this a, b because that's equivalent to solving the Discrete Log problem. And from A, B, the attacker can't compute g^{ab}: we believe this is hard and is called the Diffie-Hellman#

Now, what happens if Alice and Bob are separated through time? That is, they can't exchange A, B quickly. Then:

1. Alice chooses random a in $\{1, \ldots, n\}$
2. $A = g^a$ is Alice's public key PK_a
3. Bob chooses b at random in $\{1, \ldots, n\}$
4. Bob retrieves A and computes $g^{ab} = A^b$.
5. Derives symmetric key k from $g^{ab}.
6. Encrypts message with k
7. Sends $B = g^b$ and $E(k, m)$
8. Alice retrieves Bob ciphertext
9. Alice computes g^{ab} with Bob's B
10. Alice derives key k from g^{ab}
11. Alice decrypts message m with k

ElGamal

Note: this is not how ElGamal originally described the system, but it is a modern view which is equivalent to the original.

Fix:

- G our finite cyclic group.
- (E_d, D_s): our symmetric authenticated encryption defined over (K, M, C)
- $H : G^2 \to K$ a hash function

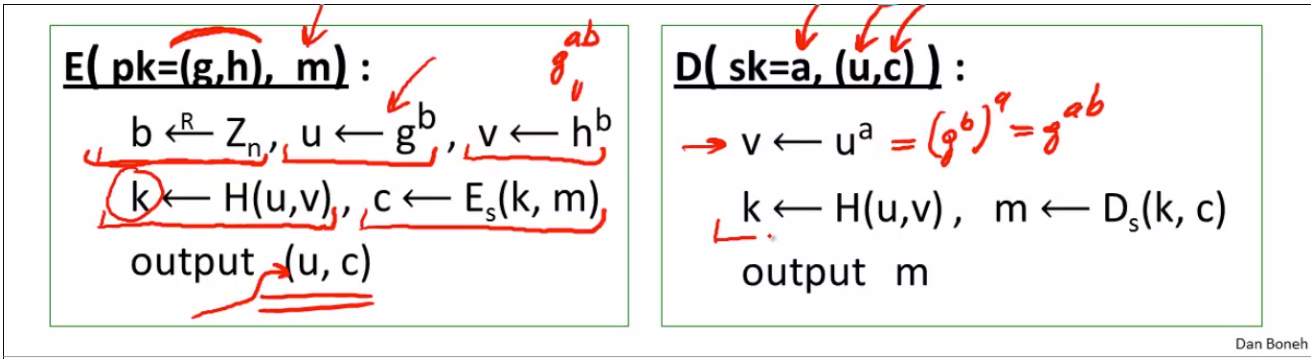
Then, construct a Public Key Encryption system:

Key Generation

- Choose random generator $g \in G$ and random $a \in Z_n$
- Output $sk = a, pk = (g, h = g^a)$

Note: the use of a random generator is needed to prove some security guarantees of the system. Getting one is pretty easy: take g and raise it to some power which is relative prime to N.

Encryption And decryption



ElGamal Performance

- Encryption takes 2 exponentiation, which is fairly slow.
- Decryption takes 1 exponentiation.

Surprisingly, this does not mean encryption takes longer than decryption: the values in encryption are fixed for the same public key: g^b and h^b. We can precompute: $[g^{2^i}, h^{2^i} for\ i = 1 ... \log_2 n]$. This leads to a 3x speedup! Although it requires the precomputation of large tables.

With this change, encryption is faster than decryption. Most libraries don't implement this! But anyway, if encryption is the bottleneck for you, you can just use RSA.

ElGamal Security

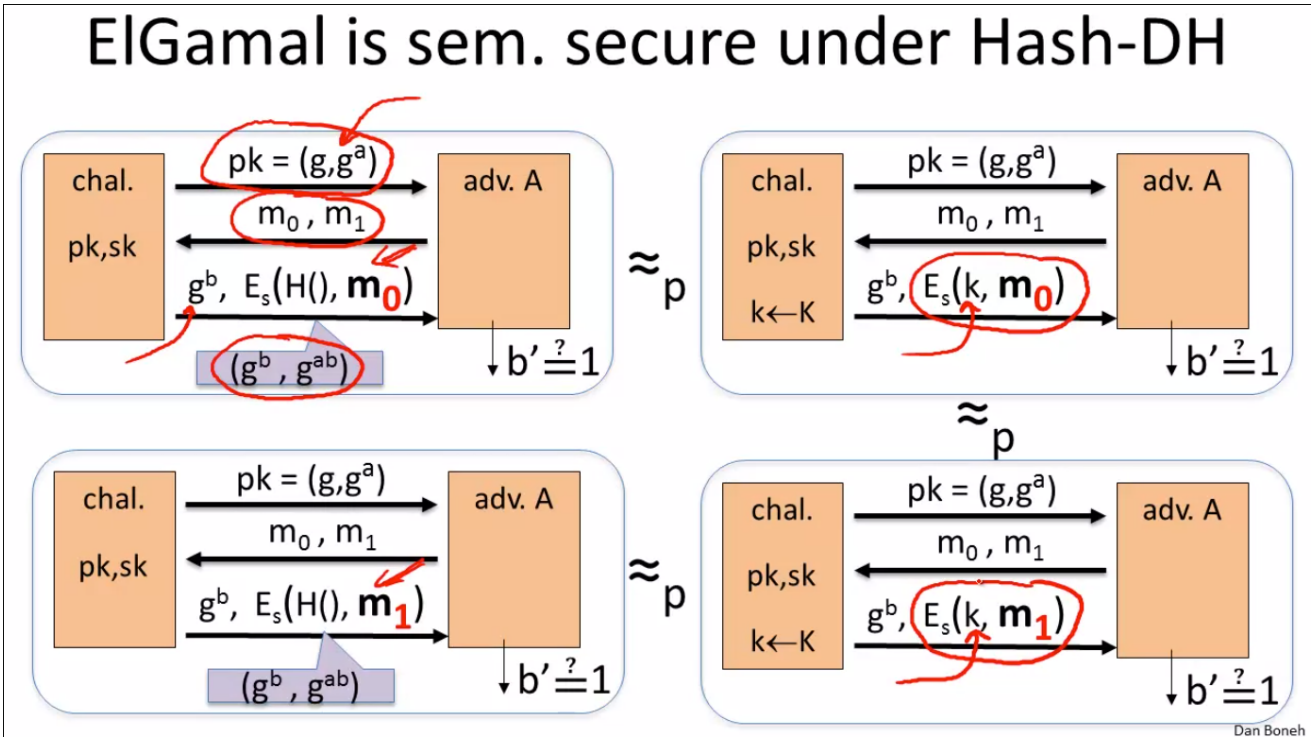
G is a finite cyclic group of order N.

The security of Diffie-Hellman is based on the assumption that $g, g^a, g^b \Rightarrow\!\!\!\!\!/\ \ g^ab$, where $\Rightarrow\!\!\!\!\!/$ here

means "it is difficult". This is called the Computational Diffie-Hellman Assumption.

This assumption is not very useful when analyzing ElGamal security. To analyze the security, we use the Hash Diffie-Hellman Assumption.

So let's see the ElGamal is semantically secure under Hash Diffie-Hellman. As usual, we do this with a game. The key is that under Hash Diffie Hellman, $H()$ is not distinguishable from $k \xleftarrow{r} K$.



But we want Chosen Ciphertext Security! To prove this, we need an even stronger assumption: the Diffie-Hellman Key Exchange > Interactive Diffie-Hellman Assumption. With that assumption, we can prove that ElGamal is chosen ciphertext secure.

We will see versions of ElGamal which only need the CDH assumption to hold to get Chosen Ciphertext Security and also don't need random oracles.

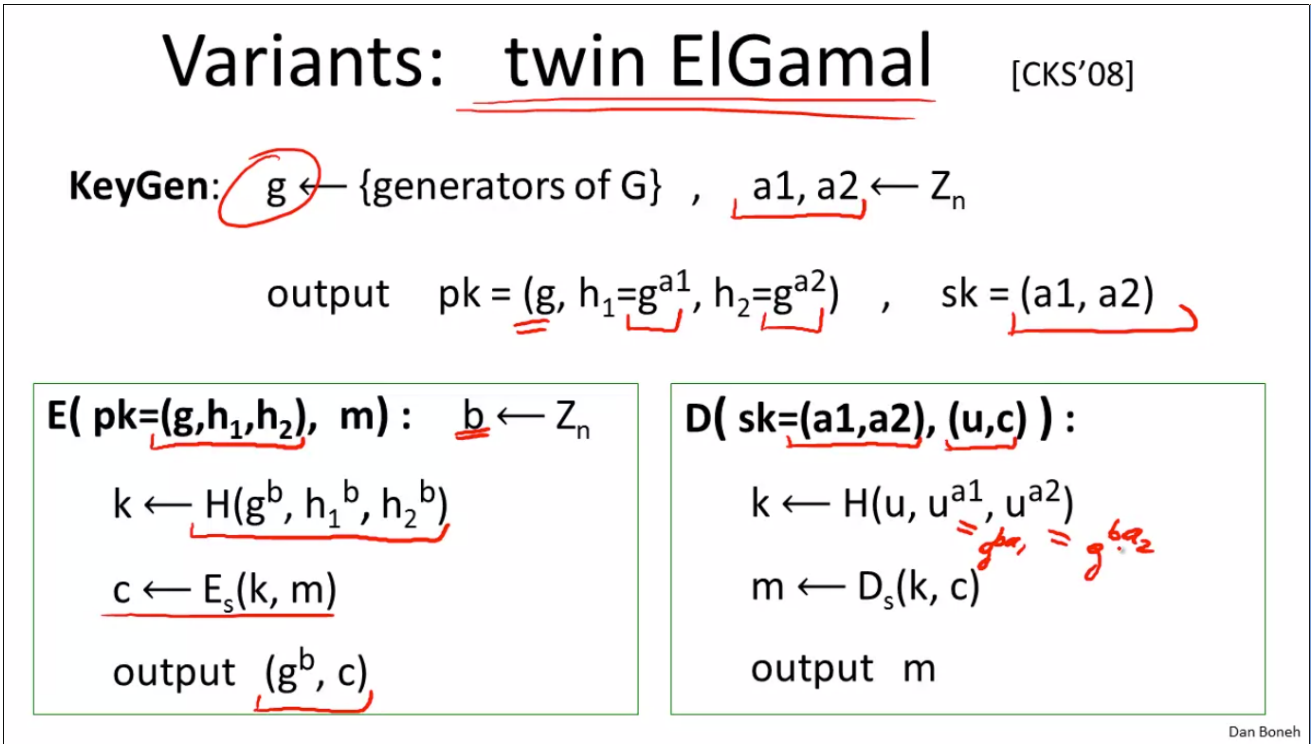
ElGamal Variants with Better Security

CCA-Secure Elgamal Without Interactive Diffie-Hellman

We want systems that do not depend on DHI.

One option is to use a group G where CDH is equivalent to the IDH. There are known as *bilinear groups*, not hard to construct. These are constructed from special Elliptic Curves.

If you want to avoid using Elliptic Curves, another option is to actually change the ElGamal system to Twin Elgamal.



Note that in regular ElGamal, we get $pk = (g, h = g^a)$. In Twin Elgamal, we put one more element for the public key and add it to the hash.

This simple modification allows us to prove Chosen Ciphertext Security with Elgamal. The cost of this is that you need one more exponentiation in encryption and decryption.

Is this worth it? Official answer: who knows... at the end, you just change the assumption. But we don't know of much of these assumptions.

CCA-Secure ElGamal without Random Oracles

One option again is to use bilinear groups.

Another one is to use groups where **Decision Diffie-Hellman**, which are subsets of $(Z_p)^*$. We won't cover this assumption in this course, but it exists.

One Way Functions

This is the *unifying theme* of Public Key Encryption. We will describe them informally:

A function $f : X \to Y$ is one-way if:

- There is an efficient algorithm to evaluate $f(\cdot)$
- Inverting f is hard:
  $\forall$ efficient A and $x \leftarrow X : Pr[f(A(f(x)) = f(x)] < negli$

Let's review some examples of one-way functions.

Generic one-way functions

Let $f : X \to Y$ be a secure PRG (where $|Y| >> |X|$). This could be built using deterministic counter mode.

Lemma: f a secure $PRG \implies f$ is one way.

This should be easy to see: $PRG(\cdot)$ should not be distinguishable from random.

Proof sketch: if A inverts F, then:

$$B(y) = \{ \begin{matrix} F(A(y)) = y & \text{output } 0 \\ \text{otherwise} & \text{output } 1 \end{matrix}$$

And we have a distinguisher for our secure PRG. This is a contradiction, so A cannot invert F if F is a secure PRG.

The problem with this one-way function is that we the output of F has no special properties, so we cannot use it for more than Key Exchange > Merkle Puzzles.

DLOG one-way function

Fix a finite Cyclic Group G (ie: $G = (Z_p)^*$) of order n and g a random generator in G.

Define: $f : Z_n \to G$ as $f(x) = g^x \in G$

Lemma: As per the Discrete Log problem, f is one-way.

Properties: $f(x), f(y) \implies f(x+y) = (g^x)^y = f(x).f(y)$. This is easy to compute even without knowing x, y, only $f(x), f(y)$. This is the basis for Key Exchange and Public Key Encryption.

RSA one-way function

- Choose random primes $p, q \approx 1024bits, N = p.q$
- Choose integers $e, d : e.d \equiv 1 \pmod{\phi(N)}$

Define: $f : Z^*_{\mathbb{N}} \to Z^*_{\mathbb{N}}, f(x) = x^e \in \mathbb{Z}_N$

Lemma: f is a one way function

Properties: $f(x.y) = f(x).f(y)$ and most importantly f has a trapdoor! This makes it specially useful for Digital Signatures.