Crypto 4: Public Key

Twitter Fight Last Year: Nick Vs Rust Rand_Core Random Number Generators

- Rust (well, the 3rd party library for it) has an interface for "secure" Random Number Generators... But they aren't actually secure!
- EG, "ChaCha8Rng"
  - A ***reduced round*** stream cipher!
  - That has no update() function: no way of adding in entropy after seeding
  - And seed() takes only 32B total (no combining entropy!)
  - Oh, and no rollback resistance either
- ***NONE*** of the "Secure" RNGs are actually cryptographically secure...
  - Because none accept and consume arbitrarily long seeds or have an update to mix in more entropy
- When I say ONLY use HMAC_DRBG, I mean it!
  - Use /dev/urandom and everything else you can think of to shove into HMAC_DRBG

And Vuln of the Day: CVE-2019-16303

- If you wrote an app in JHipster last year or before...
  - You probably want a password reset function...
- Password reset generates "random" URLs
  - But of course, they used a bad RNG!
- So generate a password request for your account
  - You get the RNGs state in the reset URL
- Now you can generate more password resets...
  - And predict what the "random" URL is...
    and take over any account you want!

Reminder Of Our Primitives So-Far: Block Cipher

- Block Cipher: Takes a fixed sized message and fixed-sized key
  - ***E(M, K)***, $E_k(M)$
  - Corresponding inverse/decryption function $D_k(M)$
  - Keyed permutation on an ***N*** bit block:
    If you don't know the key, it should be indistinguishable from a random permutation
  - If you change a single bit of either the input or the key, the output should look totally different
  - E.g. AES: 128b data blocks, keys are 128, 192, 256 (AES-128, AES-192, AES-256)
- Block Cipher Mode
  - A way of repeatedly applying a block cipher on a longer message:
    Goal is to make it independent under chosen plaintext attacks

Reminder Of Primitives So-Far: Hash Function

- Hash takes an arbitrary message M and reduces it to a fixed size
  - Should be indistinguishable from a random number
  - Change a single bit on the input -> Output looks like a completely different random number
  - SHA-256, SHA-384, SHA-512: SHA2 family outputting 256b, 384b, 512b
  - SHA3-256, SHA3-384, SHA3-512: SHA3 family
- Irreversible & resists collisions
  - Intractable given ***H(X)*** to determine ***X***
    (1st Preimage Resistant)
  - Intractable given ***X***, ***H(X)***, find ***X'*** != ***X*** such that ***H(X)*** = ***H(X')***
    (2nd Preimage Resistant)
  - Intractible to find any ***X***, ***X'***, ***X'*** != ***X*** such that ***H(X)*** = ***H(X')***
    (Collision Resistant)

Reminder Of Primitives So-Far: MAC

- MAC takes an arbitrary message M and a key K creating a fixed-length tag
  - ***MAC(M,K)*** -> ***T***
  - Without ***K***, it is infeasible to create ***M'*** such that ***MAC(M', K)*** -> ***T***
  - Without ***K***, it is infeasible to create ***M'***, ***T'*** such that ***MAC(M', K)*** -> ***T'***
  - But with ***K***, of course you can create a valid ***M'***, ***T'*** pair
    - And for some MACs create ***M'*** which MACs to ***T***
- Several alternatives but only One True MAC to use: HMAC
  - Construct using hash functions to create a MAC:
    Has all the previous properties of a hash plus all the properties of a MAC

Reminder Of Primitives So-Far:
pRNG (Pseudo Random Number Generator)

- Three operations:
  - seed(entropy): Set internal state based on arbitrarily long, truly random inputs
  - update(entropy): Add in additional entropy
    Update with 0-entropy should not degrade internal state
  - generate(length): Generate an ***n*** bit string that should be indistinguishable from random
- If you know the internal state it is fully predictable
- If you don't it should be indistinguishable from random
- HMAC_DRBG is the absolute best
  - Also has rollback resistance, if you learned the internal state at time T, you can't predict previous outputs

Public Key...

- All our previous primitives required a "miracle":
  - We somehow have to have Alice and Bob get a shared ***k***.
- Enter Public Key cryptography: the miracle of modern cryptography
  - How starting Friday, but ***what*** today
- Three primitives:
  - Public Key Agreement (previous Ephemeral Diffie/Hellman)
  - Public Key Encryption
  - Public Key Signatures
- Based on some families of magic math...
  - For us, we will use some group-theory based primitives

Public Key Agreement

- Alice and Bob have a channel...
  - There may be an eavesdropper ***but not a manipulator***
- The goal: Alice & Bob agree on a ***random*** value
  - This will be ***k*** for all subsequent communication
- When done, the key is thrown away
  - Designed to prevent an attacker who later recovers Alice or Bob's long lived secrets from finding ***k***.

Reminder of Primitives So Far: Ephemeral Diffie/Hellman Key Exchange

- Public values: prime ***p***, generator ***g***
  - Elliptic curve: different magic math, fewer bits (256b/384b instead of 2048b/3096b for the same security)
- Alice creates random ***a***, ***0<a<p***, computes ***A* = *g*** ^{a} mod ***p***, sends it
- Bob creates random ***b***, ***0<b<p***, computers ***B*** = ***g*** ^{b} mod ***p***, sends it
- Alice computes ***B*** ^{a} mod ***P*** = ***g*** ^{ab} mod ***P*** = ***K***
- Bob computes ***A*** ^{b} mod ***P*** = ***g*** ^{ab} mod ***P*** = ***K***
- Thought to be hard to go backwards (discrete log) to ***a*** given ***A***

Public Key Encryption

- Alice has ***two*** keys:
  - K_{pub}: Her public key, anyone can know
  - K_{priv}: Her private key, a deep dark secret
    - Sometimes written as K_{alice}, K^{-1}_{alice}
- Anyone has access to Alice's public key
- For anyone to send a message to Alice:
  - Create a random session key ***k***
    - Used to encrypt the rest of the message
  - Encrypt ***k*** using Alice's K_{pub}.
- Only Alice can ***decrypt*** the message
  - The decryption function only works with K_{priv}!

Public Key Signatures

- Once again, Alice has ***two*** keys:
  - K_{pub}: Her public key, anyone can know
  - K_{priv}: Her private key, a deep dark secret
- She can sign a message
  - Calculate ***H(M)***
  - $S(K_{priv}, H(M))$: Sign ***H(M)*** with K_{priv}.
- Anyone can now verify
  - Recalculate ***H(M)***
  - $V(K_{pub}, S(K_{priv}, H(M)), H(M))$: Verify that the signature was created with K_{priv}

Things To Remember...

- Public key is ***slow!***
  - Orders of magnitude slower than symmetric key
- Public key is based on delicate magic math
  - Discrete log in a group is the most common
  - RSA
  - Some new "post-quantum" magic...
- Some systems in particular are easy to get wrong
  - We will get to some of the epic crypto-fails later

Our Roadmap For Public Key...

- Public Key:
  - Something ***everyone*** can know
- Private Key:
  - The secret belonging to a specific person
- Diffie/Hellman:
  - Provides key exchange with no pre-shared secret
- ElGamal & RSA:
  - Provide a message to a recipient only knowing the recipient's ***public key***
- DSA & RSA signatures:
  - Provide a message that anyone can prove was generated with a ***private key***

Public Key Cryptography #1: RSA

- Alice generates two ***large*** primes, **p** and **q**
  - They should be generated randomly:
    Generate a large random number and then use a "primality test":
    A ***probabilistic*** algorithm that checks if the number is prime
- Alice then computes **n = p*q** and **ϕ(n) = (p-1)(q-1)**
  - **ϕ(n)** is Euler's totient function, in this case for a composite of two primes
  - ***n*** is big: 2048b to 4096b long!
- Chose random **2 < e < ϕ(n)**
  - **e** also needs to be relatively prime to **ϕ(n)** but it can be small
- Solve for **d =** $\mathbf{e^{-1}}$ **mod ϕ(n)**
  - You can't solve for **d** without knowing **ϕ(n)**, which requires knowing **p** and **q**
- **n**, **e** are public, **d**, **p**, **q**, and **ϕ(n)** are secret

RSA Encryption

- Bob can easily send a message m to Alice:
  - Bob computes $\mathbf{c = m^e \bmod n}$
  - Without knowing **d**, it is believed to be intractable to compute **m** given **c**, **e**, and **n**
    - But if you can get **p** and **q**, you can get **d**:
      It is ***not known*** if there is a way to compute **d** without also being able to factor **n**, but it is known that if you can factor **n**, you can get **d**.
    - And factoring is ***believed*** to be hard to do
- Alice computes $\mathbf{m = c^d \bmod n = m^{ed} \bmod n}$
- Time for some math magic...

RSA Encryption/Decryption, con't

- So we have: $\mathbf{D(C, K_D) = (M^{e \cdot d}) \bmod n}$
- Now recall that **d** is the multiplicative inverse of **e**, modulo **ϕ(n)**, and thus:

  $\mathbf{e \cdot d = 1 \bmod \phi(n)}$ (by definition)

  $\mathbf{e \cdot d - 1 = k \cdot \phi(n)}$ for some **k**
- Therefore $\mathbf{D(C, K_D) = M^{e \cdot d} \bmod n = (M^{e \cdot d-1}) \cdot M \bmod n}$

  $\mathbf{= (M^{k\phi(n)}) \cdot M \bmod n}$

  $\mathbf{= [(M^{\phi(n)})^k] \cdot M \bmod n}$

  $\mathbf{= (1^k) \cdot M \bmod n}$ *by Euler's Theorem: $a^{\phi(n)} \bmod n = 1$*

  $\mathbf{= M \bmod n = M}$

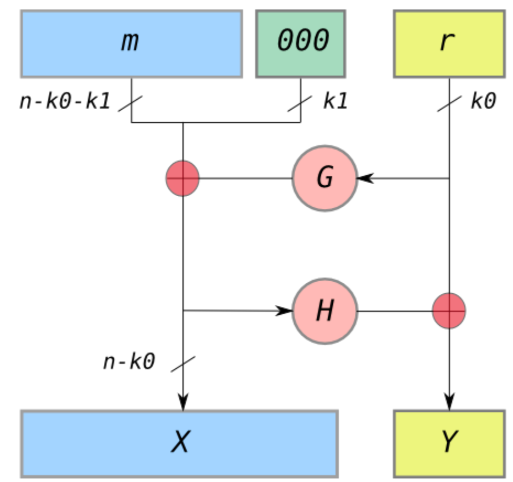
(believed) Eve can recover M from C *iff* Eve can factor n=p·q

But It Is Not That Simple...

- What if Bob wants to send the same message to Alice twice?
  - Sends $\mathbf{m^{e_a} \bmod n_a}$ and then $\mathbf{m^{e_a} \bmod n_a}$
  - Oops, not IND-CPA!
- What if Bob wants to send a message to Alice, Carol, and Dave:
  - $\mathbf{m^{e_a} \bmod n_a}$
    $\mathbf{m^{e_b} \bmod n_b}$
    $\mathbf{m^{e_c} \bmod n_c}$
  - This ends up leaking information an eavesdropper can use ***especially*** if $3 = e_a = e_b = e_c$!
- Oh, and problems if both **e** and **m** are small...
- As a result, you ***can not*** just use plain RSA:
  - You need to use a "padding" scheme that makes the input random but reversible

RSA-OAEP
(Optimal asymmetric encryption padding)

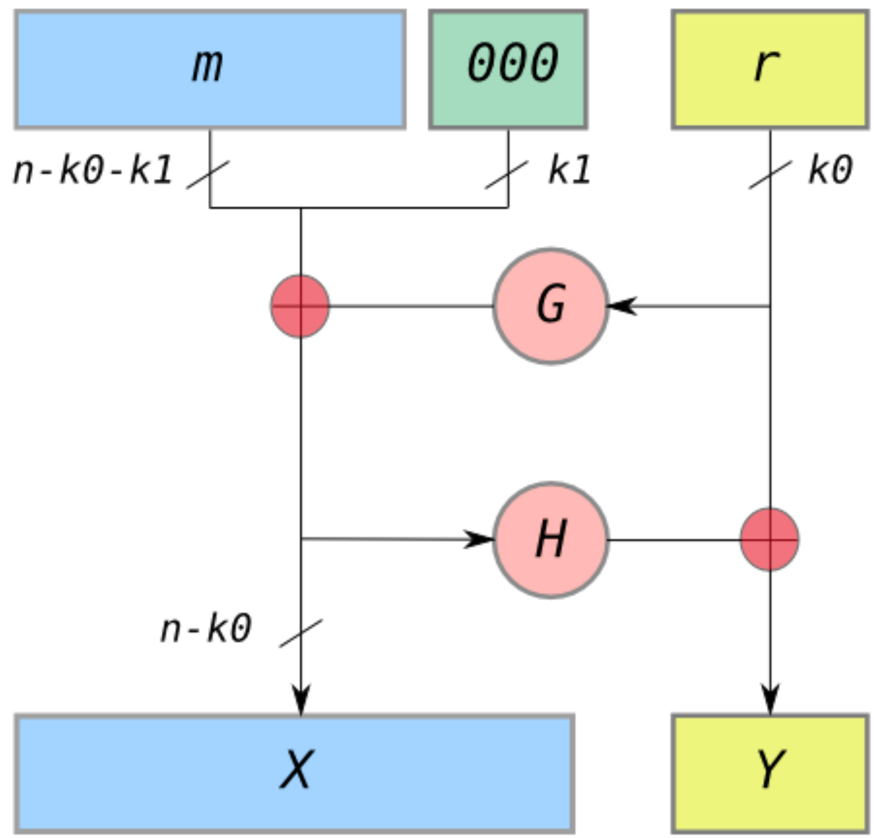
- A way of processing m with a hash function & random bits
  - Effectively "encrypts" **m** replacing it with **X = [m,0...]** $\oplus$ **G(r)**
    - **G** and **H** are hash functions (EG SHA-256)
      k_0 = # of bits of randomness, **len(m) +** k_1 **+** k_0 **= n**
  - Then replaces r with **Y = H(G(r)** $\oplus$ **[m,0...])** $\oplus$ **R**
  - This structure is called a "Feistel network":
    - It is always designed to be reversible.
      Many block ciphers are based on this concept applied multiple times with **G** and **H** being functions of **k** rather than just fixed operations



- This is more than just block-cipher padding (which involves just adding simple patterns)
  - Instead it serves to both pad the bits and make the data to be encrypted "random"

So How Does This Work?

- G and H are not (necessarily) reversible
  - EG, for OAEP it is a hash function: Designed to mix in the randomness and make it uniform
  - Needed for RSA because we want to only ever encrypt "random" values with the public key
  - And since ***r*** is random and ***G*** is a hash, ***m*** is xor'ed with random...
    - Which is then hashed and XOR'ed back into ***r*** to produce ***Y***
- But XOR is!
  - So we do H(***X***) xor ***Y*** to recover ***r***
  - And now G(***r***) xor ***X*** to recover ***m***



But Its Not That Simple... Timing Attacks

- Using normal math, the ***time*** it takes for Alice to decrypt **c** depends on **c** and **d**
  - Ruh roh, this can leak information...
  - More complex RSA implementations take advantage of knowing **p** and **q** directly...
    but also leak timing
- People have used this to guess and then check the bits of **q** on OpenSSL
  - http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf
- And even more subtle things are possible...

```
x = C
for j = 1 to n
    x = mod(x^2, N)
    if d_j == 1 then
        x = mod(xC, N)
    end if
next j
return x
```

So How to Find Bob's Key?

- Lots of stuff later, but for now...
  The Leap of Faith!
- Alice wants to talk to Bob:
  - "Hey, Bob, tell me your public key!"
- Now on all subsequent times...
  - "Hey, Bob, tell me your public key", and check to see if it is different from what Alice remembers
- Works assuming the ***first time*** Alice talks to Bob there isn't a Man-in-the-Middle
  - ssh uses this

RSA Signatures...

- Alice computes a hash of the message **H(m)**
  - Alice then computes $\mathbf{s = (H(m))^d \bmod n}$
- Anyone can then verify
  - $\mathbf{v = s^e \bmod m = ((H(m))^d)^e \bmod n = H(m)}$
- Once again, there are "F-U"s...
  - Have to use a proper encoding scheme to do this properly and all sort of other traps
  - One particular trap: a scenario where the attacker can get Alice to repeatedly sign things (an "oracle")

But Signatures Are Super Valuable...

- They are how we can prevent a MitM!
- If Bob knows Alice's key, and Alice knows Bob's...
- Alice doesn't just send a message to Bob...
  - But creates a random key **k**...
  - Sends $\mathbf{E(M,K_{sess})}$, $\mathbf{E(K_{sess},B_{pub})}$, $\mathbf{S(H(M),A_{priv})}$
- Only Bob can decrypt the message, and Bob can verify the message came from Alice
  - So Mallory is SOL!

RSA Isn't The Only Public Key Algorithm

- Isn't RSA enough?
  - RSA isn't particularly compact or efficient: dealing with 2000b (comfortably secure) or 3000b (NSA-paranoia) bit operations
  - Can we get away with fewer bits?
    - Well, Diffie-Hellman isn't any better...
    - But ***elliptic curve*** Diffie-Hellman is
- RSA also had some patent issues
  - So an attempt to build public key algorithms around the Diffie-Hellman problem

El-Gamal

- Just like Diffie-Hellman...
  - Select **p** and **g**
    - These are public and can be shared:
      Note, they need to be carefully considered how to create p and g...
      Math beyond the level of this class
- Alice choses **x** randomly as her private key
  - And publishes $\mathbf{h = g^x \bmod p}$ as her public key
- Bob, to encrypt m to Alice...
  - Selects a *random* **y**, calculates $\mathbf{c_1 = g^y \bmod p}$, $\mathbf{s = h^y \bmod p = g^{xy} \bmod p}$
    - **s** becomes a shared secret between Alice and Bob
  - Maps message **m** to create **m'**, calculates $\mathbf{c_2 = m' * s \bmod p}$
- Bob then sends $\{\mathbf{c_1, c_2}\}$

El-Gamal Decryption

- Alice first calculates $\mathbf{s = c_1^x \bmod p}$
  - Then Alice calculates $\mathbf{m' = c_2 * s^{-1} \bmod p}$
  - Then Alice calculates the inverse of the mapping to get **m**
- Of course, there are problems...
  - Attacker can always change **m'** to **2m'**
  - What if Bob screws up and reuses y?
  - $\mathbf{c_2 = m_1' * s \bmod p}$
    $\mathbf{c_2' = m_2' * s \bmod p}$
  - Ruh roh, this leaks information:
    $\mathbf{c_2 / c_2' = m_1' / m_2'}$
    - So if you know $\mathbf{m_1}$...

In Practice: Session Keys...

- You use the public key algorithm to encrypt/agree on a session key..
  - And then encrypt the real message with the session key
  - You ***never*** actually encrypt the message itself with the public key algorithm
  - Often a set of keys: encrypt and MAC keys that are separate in each direction
- Why?
  - Public key is ***slow***... Orders of magnitude slower than symmetric key
  - Public key may cause weird effects:
    - EG, El Gamal where an attacker can change the message to ***2m***...
      - If ***m*** had meaning, this would be a problem
      - But if it just changes the encryption and MAC keys, the main message won't decrypt

DSA Signatures...

- Again, based on Diffie-Hellman
  - Two initial parameters, **L** and **N**, and a hash function **H**
    - **L** == key length, eg 2048
      N <= len(H), e.g. 256
    - An N-bit prime **q**, an L-bit prime **p** such that **p - 1** is a multiple of **q**, and
      $g = h^{(p-1)/q} \bmod p$ for some arbitrary **h** ($1 < h < p - 1$)
    - {**p, q, g**} are public parameters
- Alice creates her own random private key **x < q**
  - Public key $y = g^x \bmod p$

Alice's Signature...

- Create a random value **k** < **q**
  - Calculate $\mathbf{r = (g^k \bmod p) \bmod q}$
    - If **r** = 0, start again
  - Calculate $\mathbf{s = k^{-1} (H(m) + xr) \bmod q}$
    - If **s** = 0, start again
  - Signature is {**r**, **s**} (Advantage over an El-Gamal signature variation: Smaller signatures)
- Verification
  - $\mathbf{w = s^{-1} \bmod q}$
  - $\mathbf{u_1 = H(m) * w \bmod q}$
  - $\mathbf{u_2 = r * w \bmod q}$
  - $\mathbf{v = (g^{u_1} y^{u_2} \bmod p) \bmod q}$
  - Validate that **v = r**

But Easy To Screw Up...

- **k** is not just a nonce... It must be random and ***secret***
  - If you know **k**, you can calculate **x**
- And even if you just reuse a random **k**... for two signatures $\mathbf{s_a}$ and $\mathbf{s_b}$
  - A bit of algebra proves that $\mathbf{k = (H_A - H_B) / (s_a - s_b)}$
- A good reference:
  - How knowing k tells you x: https://rdist.root.org/2009/05/17/the-debian-pgp-disaster-that-almost-was/
  - How two signatures tells you k: https://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/

And ***NOT*** theoretical:
Sony Playstation 3 DRM

- The PS3 was designed to only run ***signed*** code
  - They used ECDSA as the signature algorithm
  - This prevents unauthorized code from running
  - They had an ***option*** to run alternate operating systems (Linux) that they then removed
- Of course this was catnip to reverse engineers
  - Best way to get people interested: ***remove*** Linux from a device...
- It turns for out one of the key authentication keys used to sign the firmware...
  - Ended up reusing the same k for multiple signatures!

And ***NOT*** Theoretical: Android RNG Bug + Bitcoin

- OS Vulnerability in 2013
  Android "SecureRandom" wasn't actually secure!
  - Not only was it low entropy, it would occasionally return the ***same value multiple times***
- Multiple Bitcoin wallet apps on Android were affected
  - "Pay B Bitcoin to Bob" is signed by Alice's public key using ECDSA
    - Message is broadcast publicly for all to see
  - So you'd have cases where "Pay B to Bob" and "Pay C to Carol" were signed with the same **k**
- So ***of course*** someone scanned for ***all*** such Bitcoin transactions

And ***Still*** Happens!
Chromebook

- Chromebooks have a built in U2F "Security key"
  - Enables signatures using 256b ECDSA to validate to particular websites
- There was a bug in the secure hardware!
  - Instead of using a random ***k*** that was 256b long, a bug caused it to be 32b long!
  - So an attacker who had a signature could simply try all possible ***k*** values!
- Fortunately in this case the damage was slight: this is for authenticating to a single website: each site used its own private key
- But still...
- https://www.chromium.org/chromium-os/u2f-ecdsa-vulnerability

So What To Use?

- Paranoids like me:
  Good libraries and use the parameters from NSA's CNSA suite
  - Open algorithms approved for Top Secret communication
  - Better yet, libraries that implement full protocols that use these under the hood!
- Symmetric cipher: AES: 256b
  - CFB mode, thankyouverymuch. Counter mode and modes which include counter mode can DIAF...
- Hash function: SHA-384
  - Use HMAC for MAC
- RSA: 3072b
- Diffie/Hellman: 3072b
- ECDH/ECDSA: P-384
- But really, this is extra paranoid:
  2048b RSA/DH, 256b EC, 128b AES, SHA-256 excellent in practice

How Can We Communicate With Someone New?

- Public-key crypto gives us amazing capabilities to achieve confidentiality, integrity & authentication without shared secrets …
- But how do we solve MITM attacks?
- How can we trust we have the true public key for someone we want to communicate with?
- Ideas?

Trusted Authorities

- Suppose there's a party that everyone agrees to trust to confirm each individual's public key
  - Say the Governor of California

- Issues with this approach?
  - How can everyone agree to trust them?
  - Scaling: huge amount of work; single point of failure …
    - ... and thus Denial-of-Service concerns
  - How do you know you're talking to the right authority??

Trust Anchors

- Suppose the trusted party distributes their key so everyone has it …


UNITED STATES OF AMERICA
LIBERTY
IN GOD WE TRUST
D
QUARTER DOLLAR


CALIFORNIA
1850
JOHN MUIR
YOSEMITE VALLEY


NEW CALIFORNIA REPUBLIC


CALIFORNIA
1850
JOHN
MUIR
YOSEMITE
VALLEY
2005
E PLURIBUS UNUM


NEW CALIFORNIA REPUBLIC

Gavin Newsom's Public Key is
0x6a128b3d3dc67edc74d690b19e072f64

Trust Anchors

- Suppose the trusted party distributes their key so everyone has it …
- We can then use this to bootstrap trust
  - As long as we have confidence in the decisions that that party makes

Digital Certificates

- Certificate ("cert") = signed claim about someone's public key
  - More broadly: a signed *attestation* about some claim

- Notation:
  $\{ M \}_K$ = "message M encrypted with public key k"
  $\{ M \}_{K^{-1}}$ = "message M signed w/ private key for K"

- E.g. M = "Nick's public key is K_{Nick} = ***0xF32A99B***..."
  Cert: M,
  {"Nick's public key … ***0xF32A99B***..." $\}_{K^{-1}_{Gavin}}$
  = ***0x923AB95E12...9772F***

Certificate

Gavin Newsom hearby asserts:

Nick's public key is K_{Nick} = **0xF32A99B...**

The signature for this statement using

K^{-1}_{Gavin} is **0x923AB95E12...9772F**

Certificate

Gavin Newsom hearby asserts:

Nick's public key is K_{Nick} = ***0xF32A99B...***

The signature for this statement using

K^{-1} This *is* ***0x923AB95E12...9772F***

Certificate

Gavin Newsom hearby asserts:

Nick's public key is K_{Nick} = **0xF32A99B...**

The signature f is computed over all of this

K^{-1}_{Gavin} *is* ***0x923AB95E12...9772F***

Certificate

Gavin Newsom hearby asserts:

Nick's public key is K_{Nick} = ***0xF32A99B...***

The signature for this statement using

K^{-1}_{Gavin} *is* ***0x923AB95E12...9772F***

and can be *validated* using:


Certificate
Gavin Newsom hearby asserts:
Nick's public key is K_Nick =
The signature for this st
K^-1 Gavin is 0x923AB95
This:
Jerry Brown's Public Key is 0x6a128b3d3dc67edc74d690b19e072f64
CALIFORNIA
1850
JOHN MUIR
YOSEMITE VALLEY
2005
E PLURIBUS UNUM

If We Find This Cert Shoved Under Our Door …

- What can we figure out?
  - If we know Gavin's key, then whether he indeed signed the statement
  - If we trust Gavin's decisions, then we have confidence we really have Nick's key
- Trust = ?
  - Gavin won't willy-nilly sign such statements
  - Gavin won't let his private key be stolen

Analyzing Certs Shoved Under Doors …

- ***How*** we get the cert doesn't affect its utility
- ***Who*** gives us the cert doesn't matter
  - They're not any more or less trustworthy because they did
  - Possessing a cert doesn't establish any identity!
- However: if someone demonstrates they can decrypt data encrypted with K_{nick}, then we have high confidence they possess K^{-1}_{Nick}
  - Same for if they show they can sign "using" K^{-1}_{Nick}

Scaling Digital Certificates

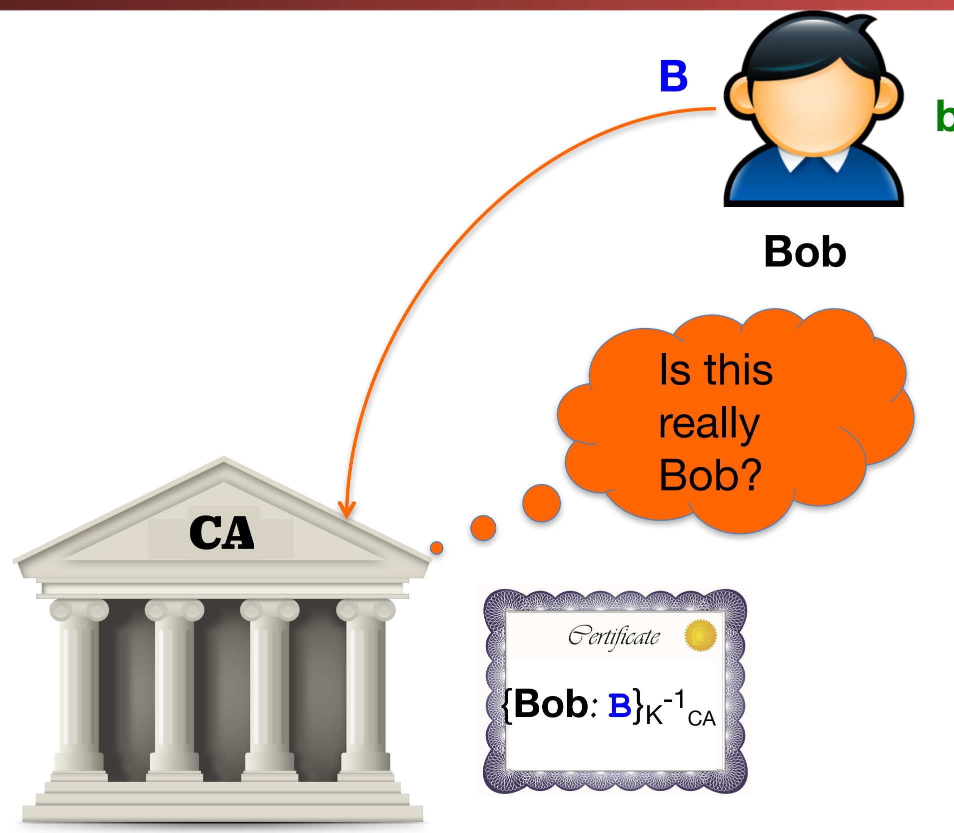
- How can this possibly scale? Surely Gavin can't sign everyone's public key!
- Approach #1: Introduce hierarchy via delegation
  - { "Michael V. Drake's public key is 0x... and I trust him to vouch for UC" }K^{-1}_{Gavin}
  - { "Carol Christ's public key is 0x... and I trust her to vouch for UCB" }K^{-1}_{Mike}
  - { "John Canny's public key is 0x... and I trust him to vouch for CS" }K^{-1}_{Carol}
  - { "Nick Weaver's public key is 0x..." }K^{-1}_{John}

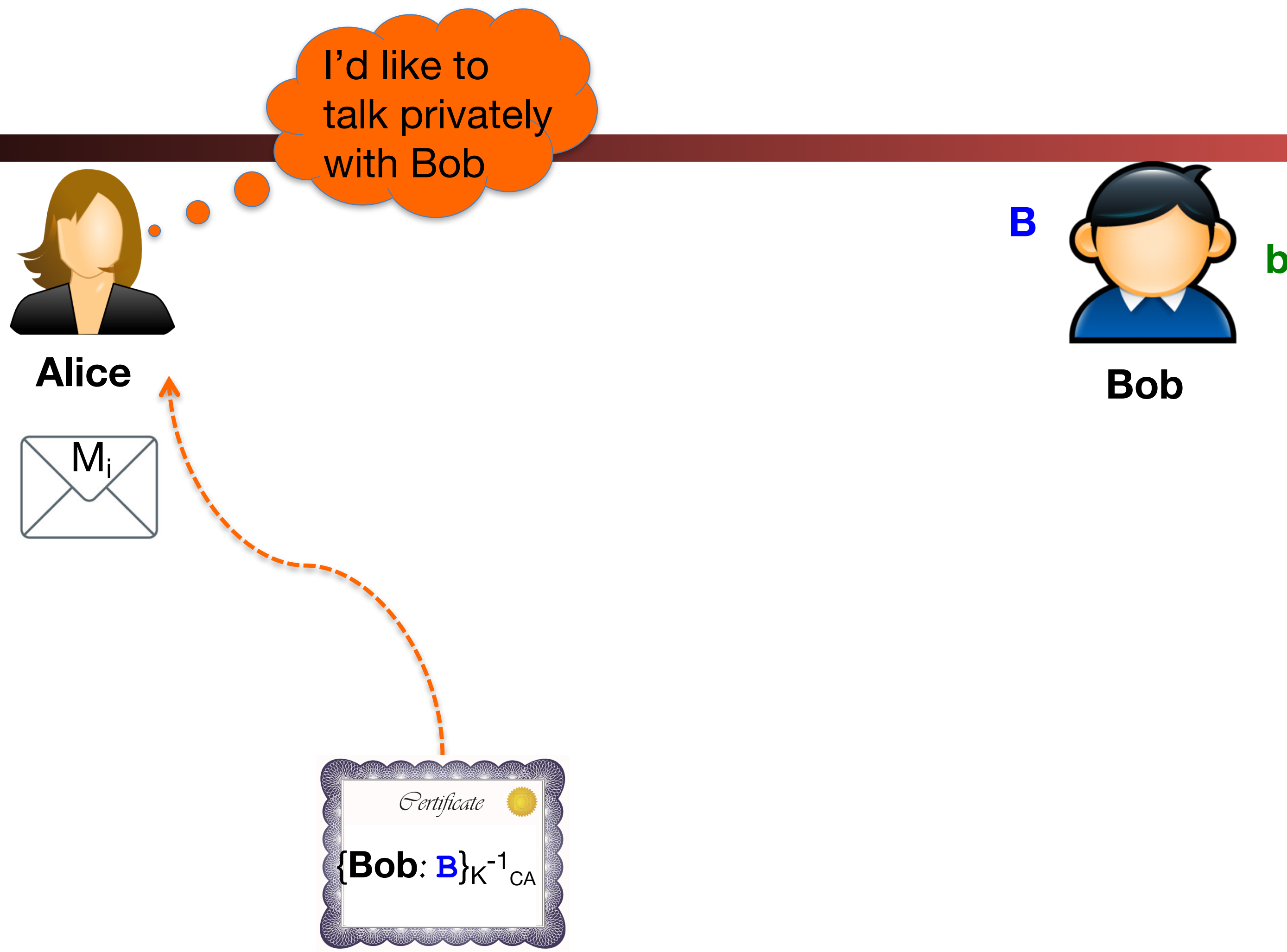
Scaling Digital Certificates, con’t

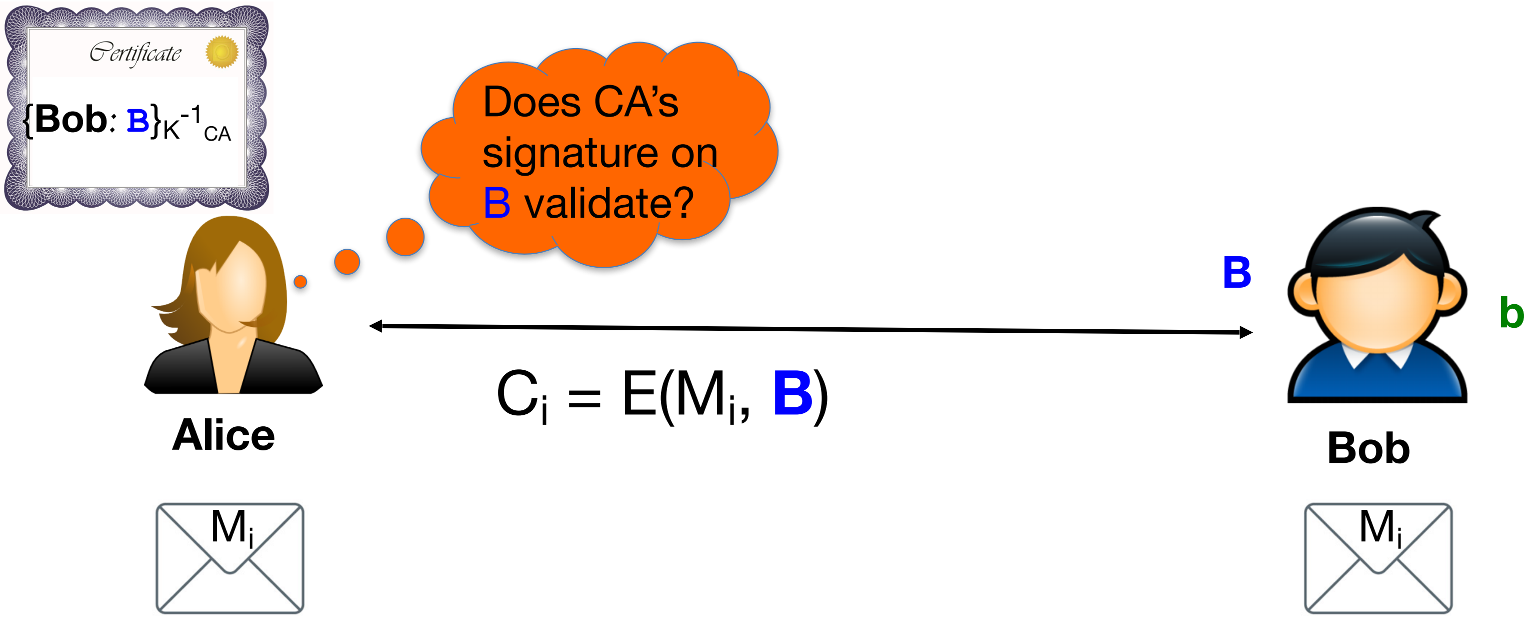
- I put this last certificate on my web page
  - (or shoves it under your door)
- Anyone who can gather the intermediary keys can validate the chain
  - They can get these (other than Gavin’s) from anywhere because they can validate them, too
  - In fact, I may as well just include those certs as well, just to make sure you don't gave to go search for them
- Approach #2: have multiple trusted parties who are in the business of signing certs …
  - (The certs might also be hierarchical, per Approach #1)

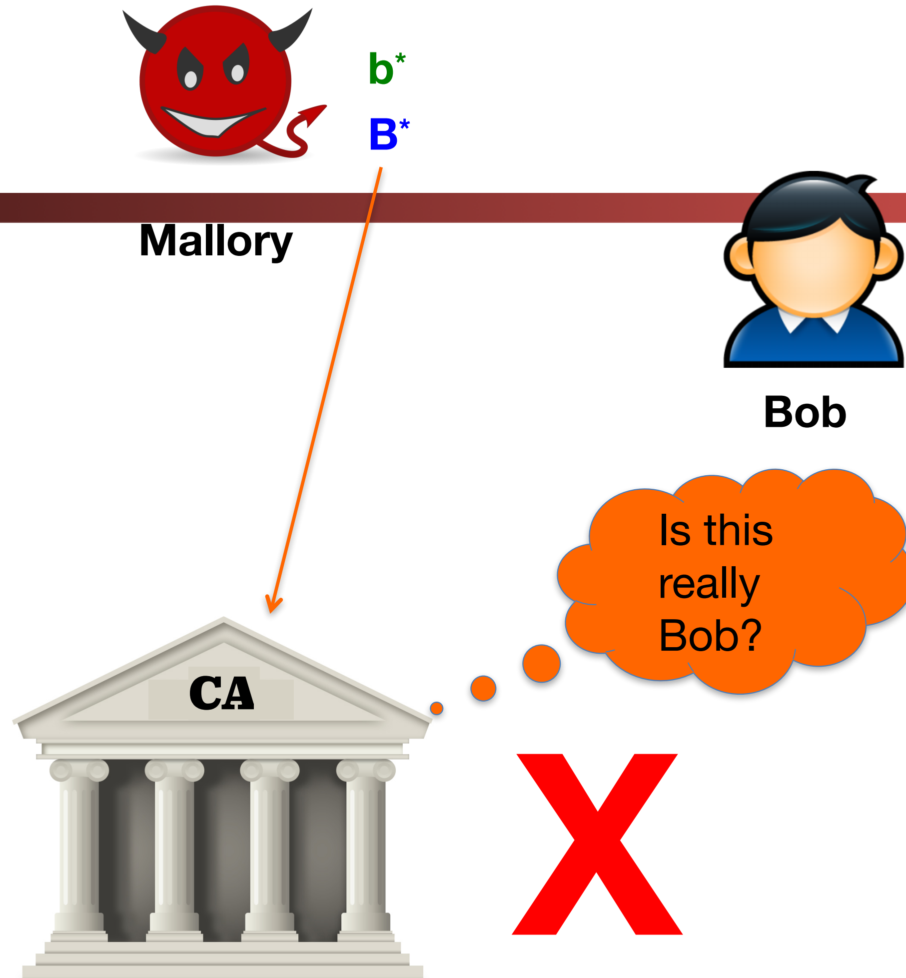
Certificate Authorities

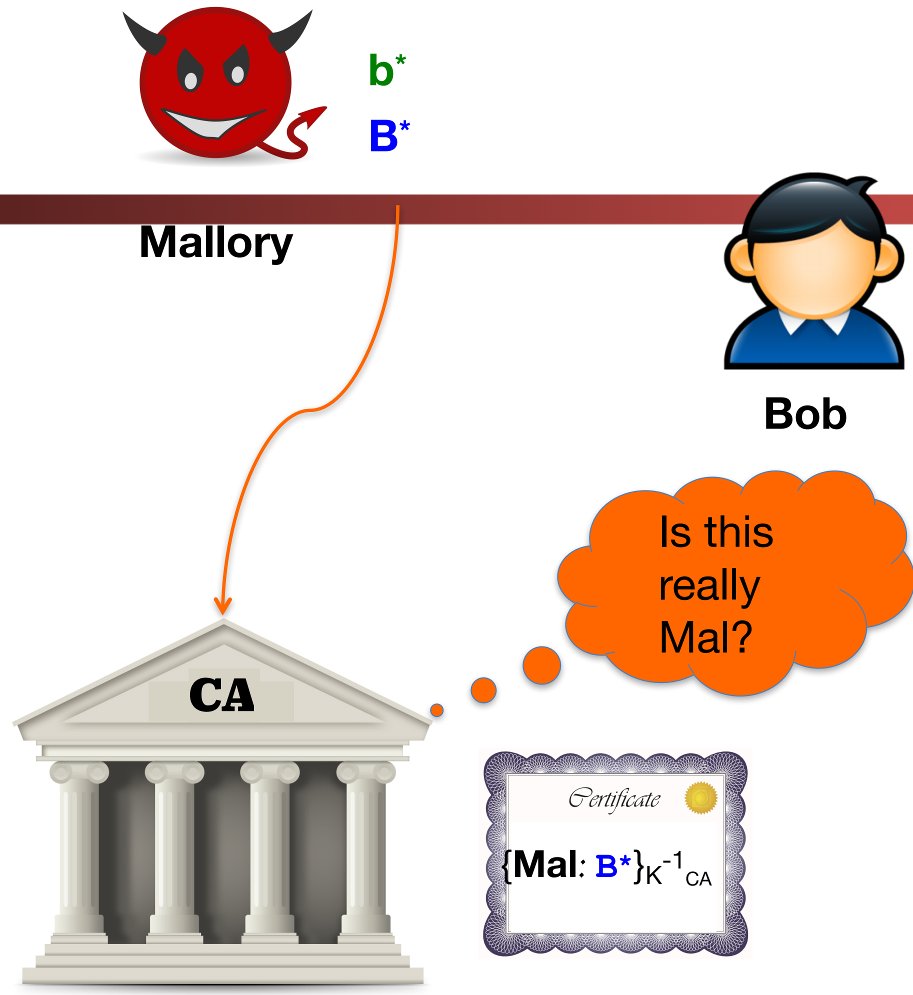
- CAs are trusted parties in a Public Key Infrastructure (PKI)
- They can operate offline
  - They sign ("cut") certs when convenient, not on-the-fly (… though see below ...)
- Suppose Alice wants to communicate confidentially w/ Bob:
  - Bob gets a CA to issue {Bob's public key is B} K^{-1}_{CA}
  - Alice gets Bob's cert any old way
  - Alice uses her known value of K_{CA} to verify cert's signature
  - Alice extracts B, sends $\{M\}K_B$ to Bob

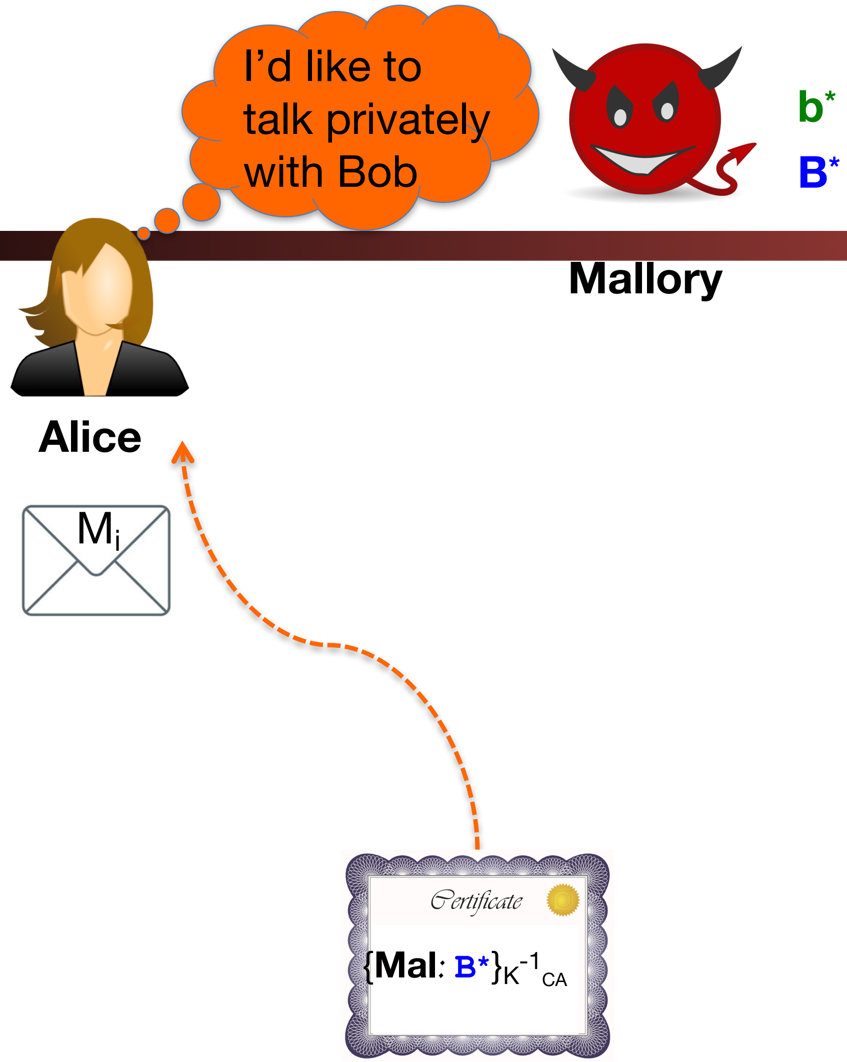

B
b
Bob
Is this really Bob?
CA
Certificate
{Bob: B}K-1CA

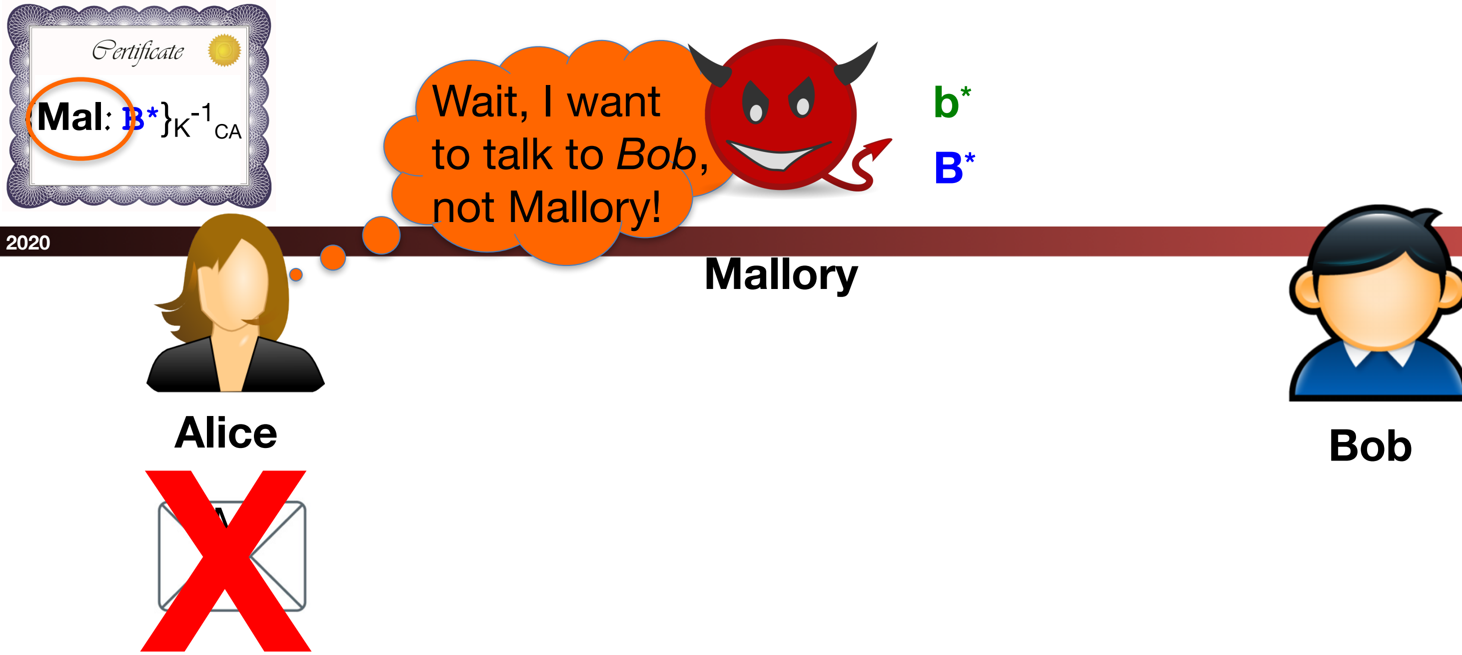

I'd like to talk privately with Bob
Alice
Bob
B
b
M_i
Certificate
$\{Bob: B\}_{K^{-1}_{CA}}$


Certificate
{Bob: B}$_{K^{-1}_{CA}}$
Does CA's signature on B validate?
B
b
Alice
Bob
$C_i = E(M_i, B)$
M_i
M_i


b*
B*
Mallory
Bob
Is this really Bob?
CA

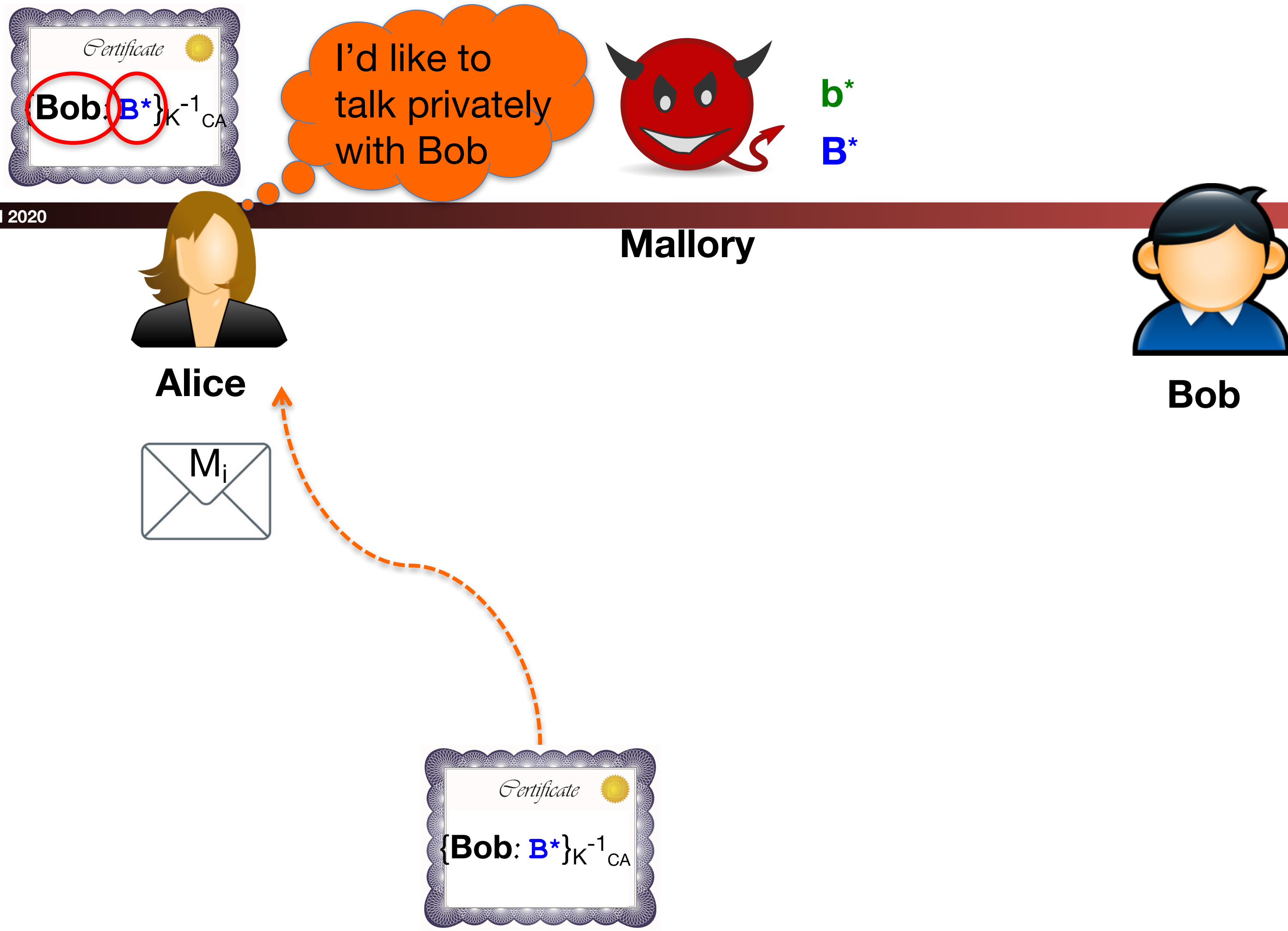

b*
B*
Mallory
Bob
Is this really Mal?
CA
Certificate
{Mal: B*}K-1CA


I'd like to talk privately with Bob
b*
B*
Mallory
Alice
Bob
M_i
Certificate
{Mal: B*}K^-1 CA


Certificate
{Mal: B*}K-1 CA
Wait, I want to talk to Bob, not Mallory!
b*
B*
Mallory
Alice
Bob

Revocation

- What do we do if a CA screws up and issues a cert in Bob's name to Mallory?

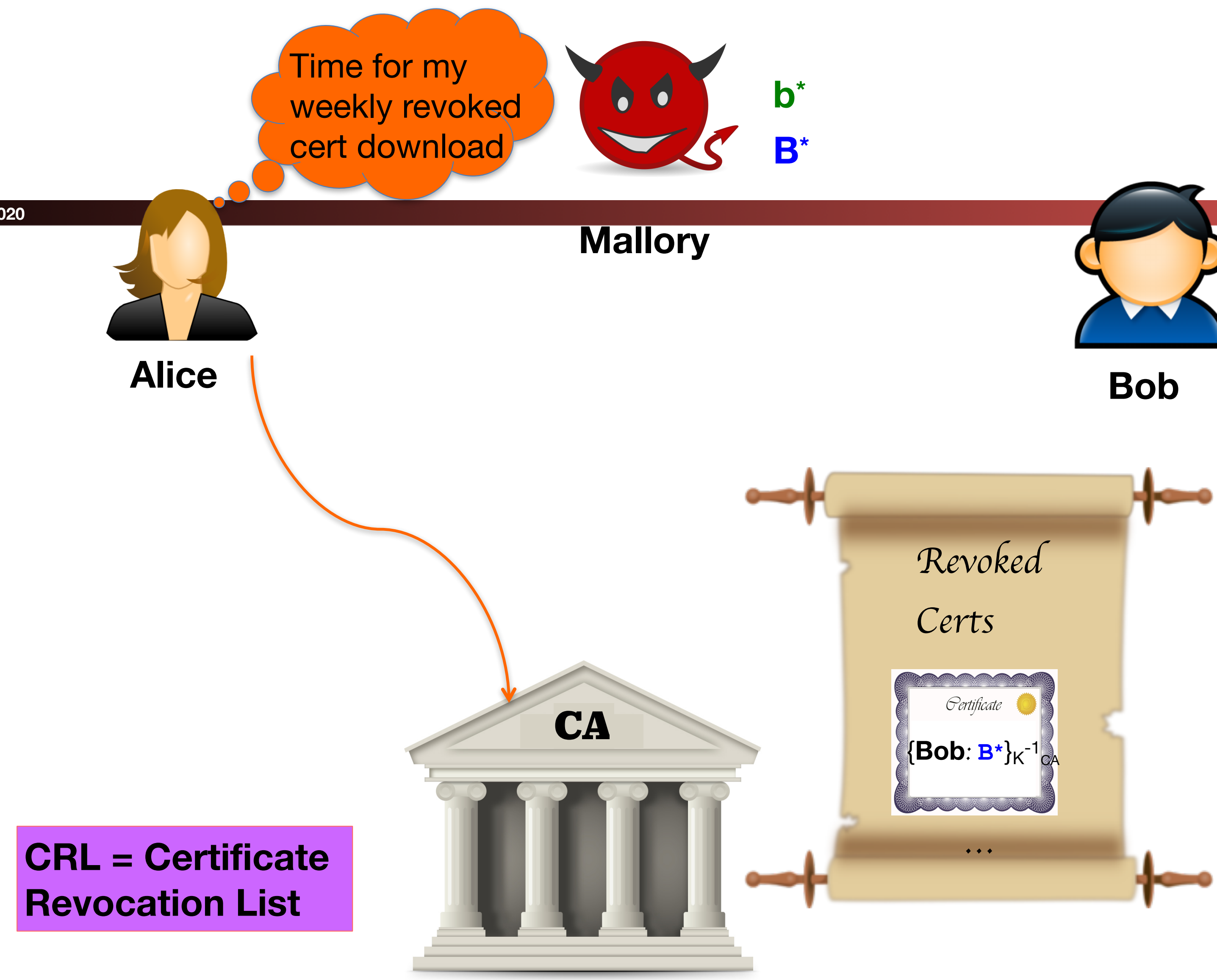

Certificate
{Bob: B*}K-1CA
I'd like to talk privately with Bob
b*
B*
Mallory
Alice
Bob
Mi
Certificate
{Bob: B*}K-1CA

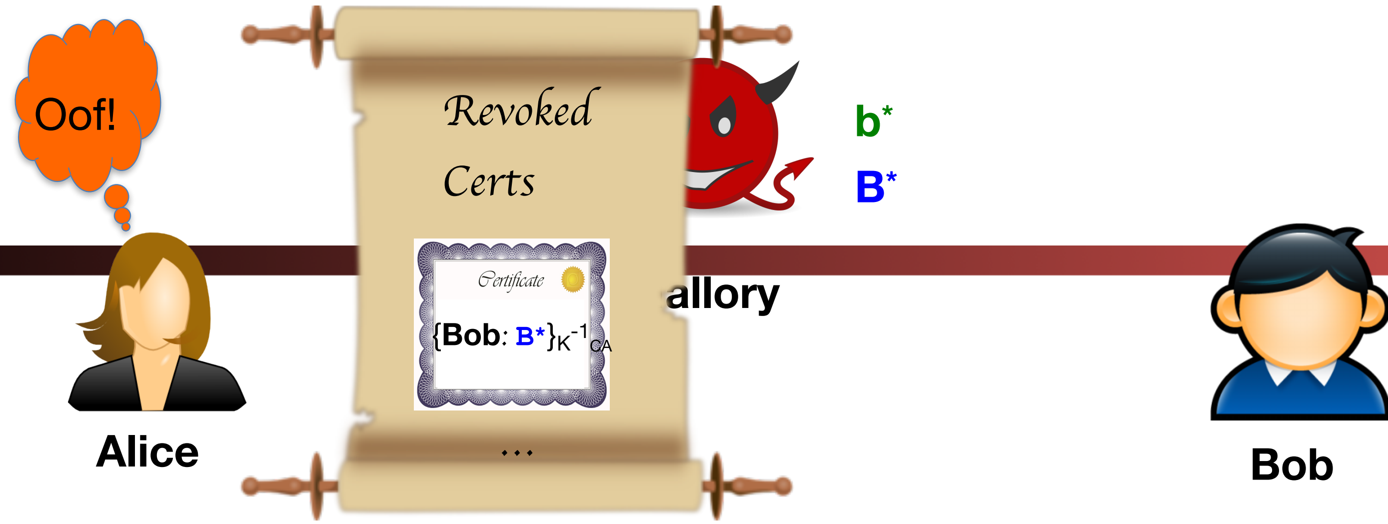
Revocation

- What do we do if a CA screws up and issues a cert in Bob's name to Mallory?
  - E.g. Verisign issued a ***`Microsoft.com`*** cert to a *Random Joe*
  - (Related problem: Bob realizes b has been stolen)
- *How do we recover from the error?*
- Approach #1: expiration dates
  - Mitigates possible damage
  - But adds management burden
    - Benign failures to renew will break normal operation
    - LetsEncrypt decided to make this VERY short to force continual updating

Revocation, con’t

- Approach #2: announce revoked certs
  - Users periodically download cert revocation list (CRL)

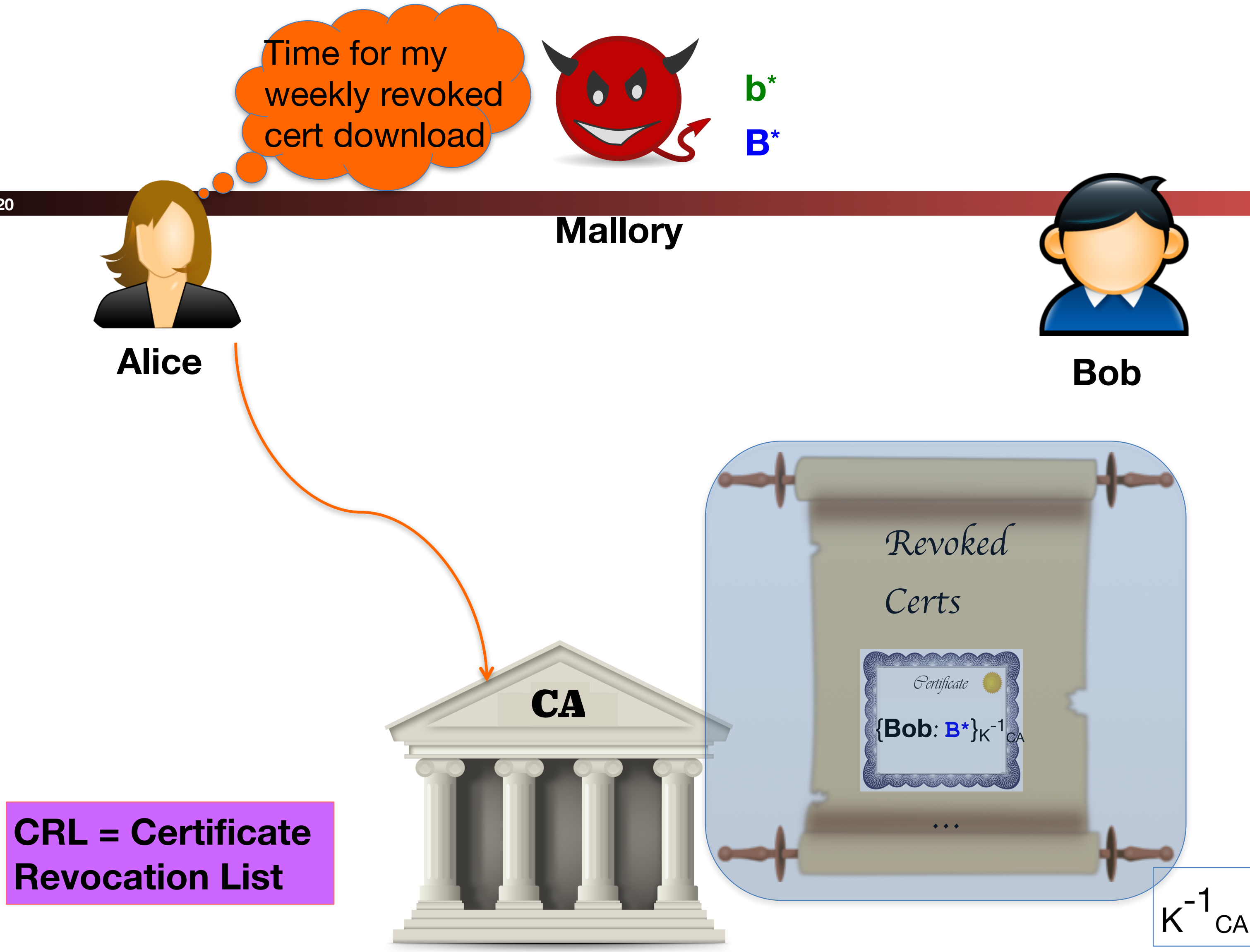

Time for my weekly revoked cert download
b*
B*
Mallory
Alice
Bob
CA
Revoked Certs
Certificate
{Bob: B*}K^-1 CA
...
CRL = Certificate Revocation List



CRL = Certificate Revocation List

Revocation, con’t

- Approach #2: announce revoked certs
  - Users periodically download cert revocation list (CRL)
- Issues?
  - Lists can get large
  - Need to authenticate the list itself – how?


Time for my weekly revoked cert download
b*
B*
Mallory
Alice
Bob
CA
Revoked
Certs
Certificate
{Bob: B*}K-1CA
...
K-1CA
CRL = Certificate Revocation List

Revocation, con’t

- Approach #2: announce revoked certs
  - Users periodically download cert revocation list (CRL)
- Issues?
  - Lists can get large
  - Need to authenticate the list itself – how? Sign it!
  - Mallory can exploit download lag
  - What does Alice do if can’t reach CA for download?
    - Assume all certs are invalid (fail-safe defaults)
      - Wow, what an unhappy failure mode!
    - Use old list: widens exploitation window if Mallory can “DoS” CA (DoS = denial-of-service)

Biggest Problem is Often Complexity

- The X509 "standard" for certificates is incredibly complicated
  - Why? Because it tried to do everything...
- If you want your eyes to bleed...
  - https://tools.ietf.org/html/rfc5280
-

The (Failed) Alternative: The "Web Of Trust"

- Alice signs Bob's Key
  - Bob Sign's Carol's
- So now if Dave has Alice's key, Dave can believe Bob's key and Carol's key…
  - Eventually you get a graph/web of trust…
- PGP started out with this model
  - You would even have PGP key signing parties
  - But it proved to be a disaster:
    Trusting central authorities can make these problems so much simpler!