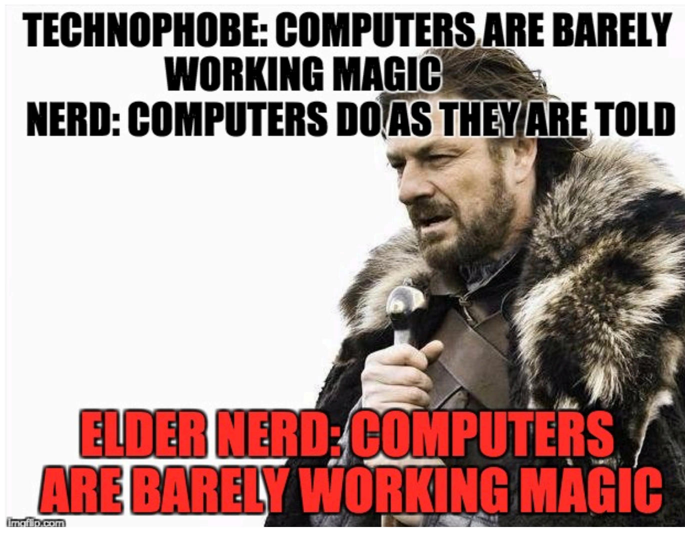
Key (mis)Management Applied Crypto and Crapto


TECHNOPHOBE: COMPUTERS ARE BARELY WORKING MAGIC
NERD: COMPUTERS DO AS THEY ARE TOLD
ELDER NERD: COMPUTERS ARE BARELY WORKING MAGIC
imgflip.com

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
- Alice choses **x** randomly as her private key
  - And publishes **$h = g^x \bmod p$** as her public key
- Bob, to encrypt m to Alice...
  - Selects a *random* **y**, calculates **$c_1 = g^y \bmod p$**, **$s = h^y \bmod p = g^{xy} \bmod p$**
    - **s** becomes a shared secret between Alice and Bob
  - Maps message **m** to create **m'**, calculates **$c_2 = m' * s \bmod p$**
- Bob then sends {**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 $\mathbf{m_1}$...

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

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
IN GOD WE TRUST
LIBERTY
D
QUARTER DOLLAR

CALIFORNIA
1850
JOHN MUIR
YOSEMITE VALLEY
2005
E PLURIBUS U

CALIFORNIA REPUBLIC

CALIFORNIA
1850
JOHN
MUIR
YOSEMITE
VALLEY
DE
2005
E PLURIBUS UNUM

CALIFORNIA REPUBLIC

Jerry Brown's Public Key is
0x6a128b3d3dc67edc74d690b19e072f64.

Trust Anchors

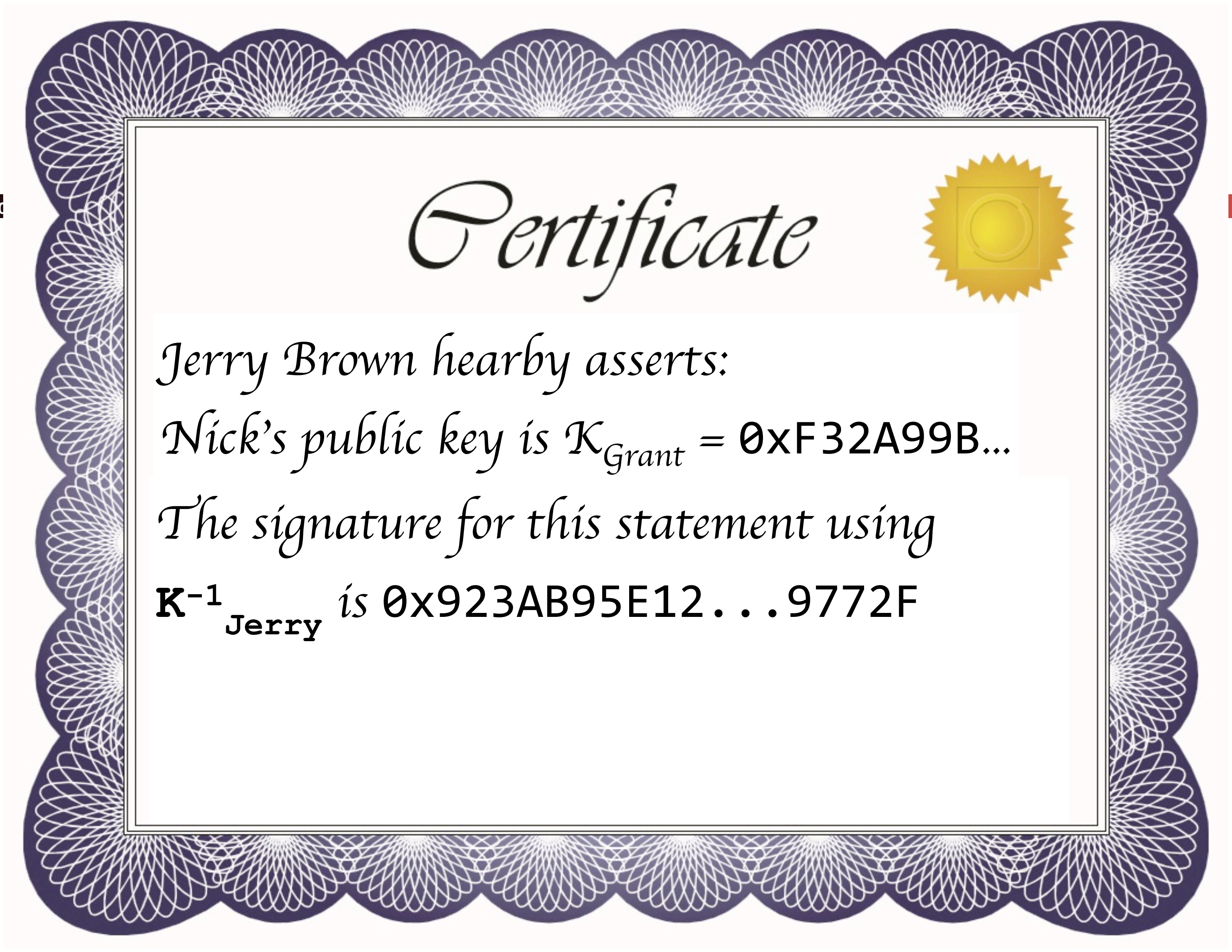
- Suppose the trusted party distributes their key so everyone has it …
- We can then use this to bootstrap trust
  - As long as we have confidence in the decisions that that party makes

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`…" $\}_{K^{-1}_{Jerry}}$
  = `0x923AB95E12...9772F`



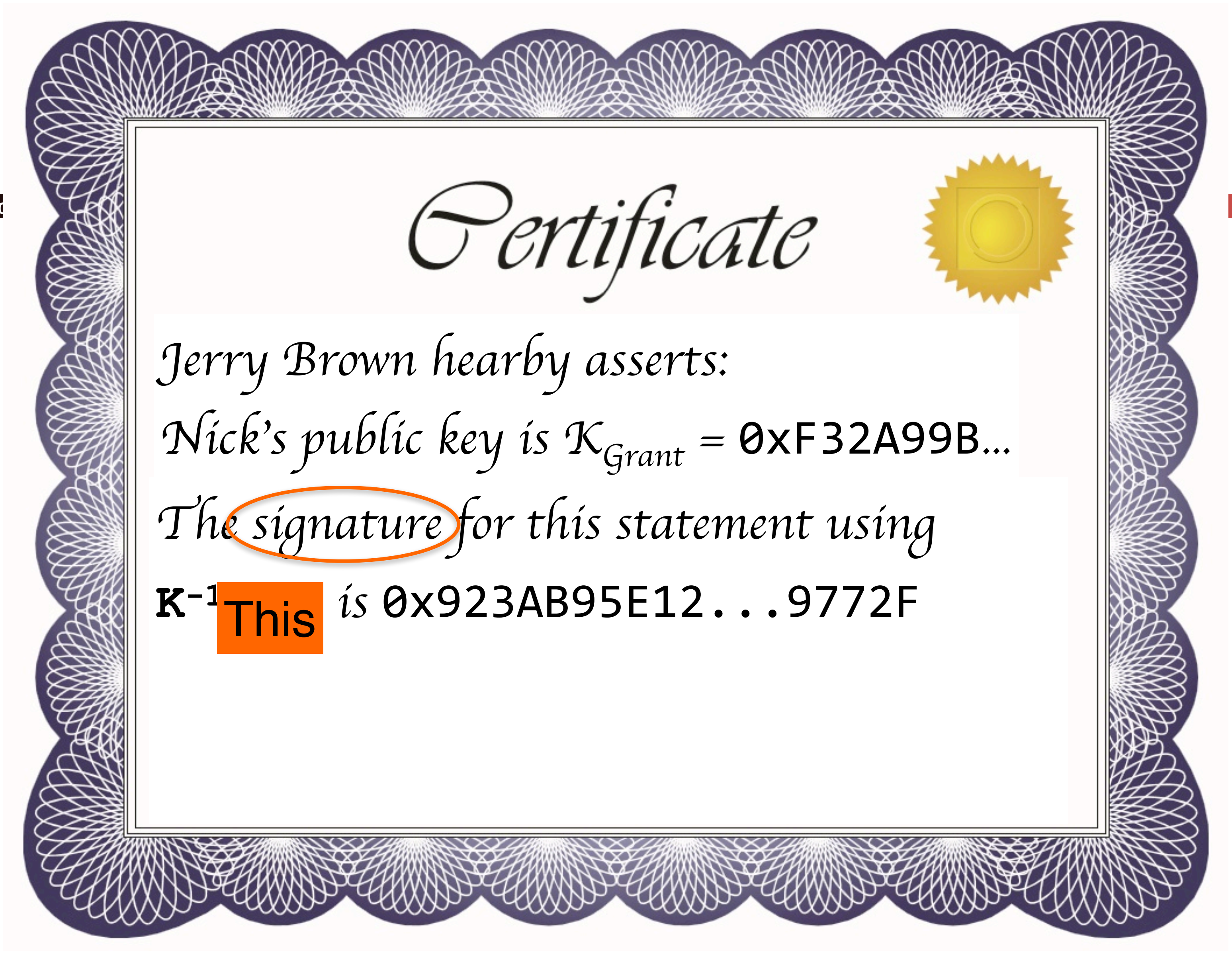
Certificate

Jerry Brown hearby asserts:

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

The signature for this statement using

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



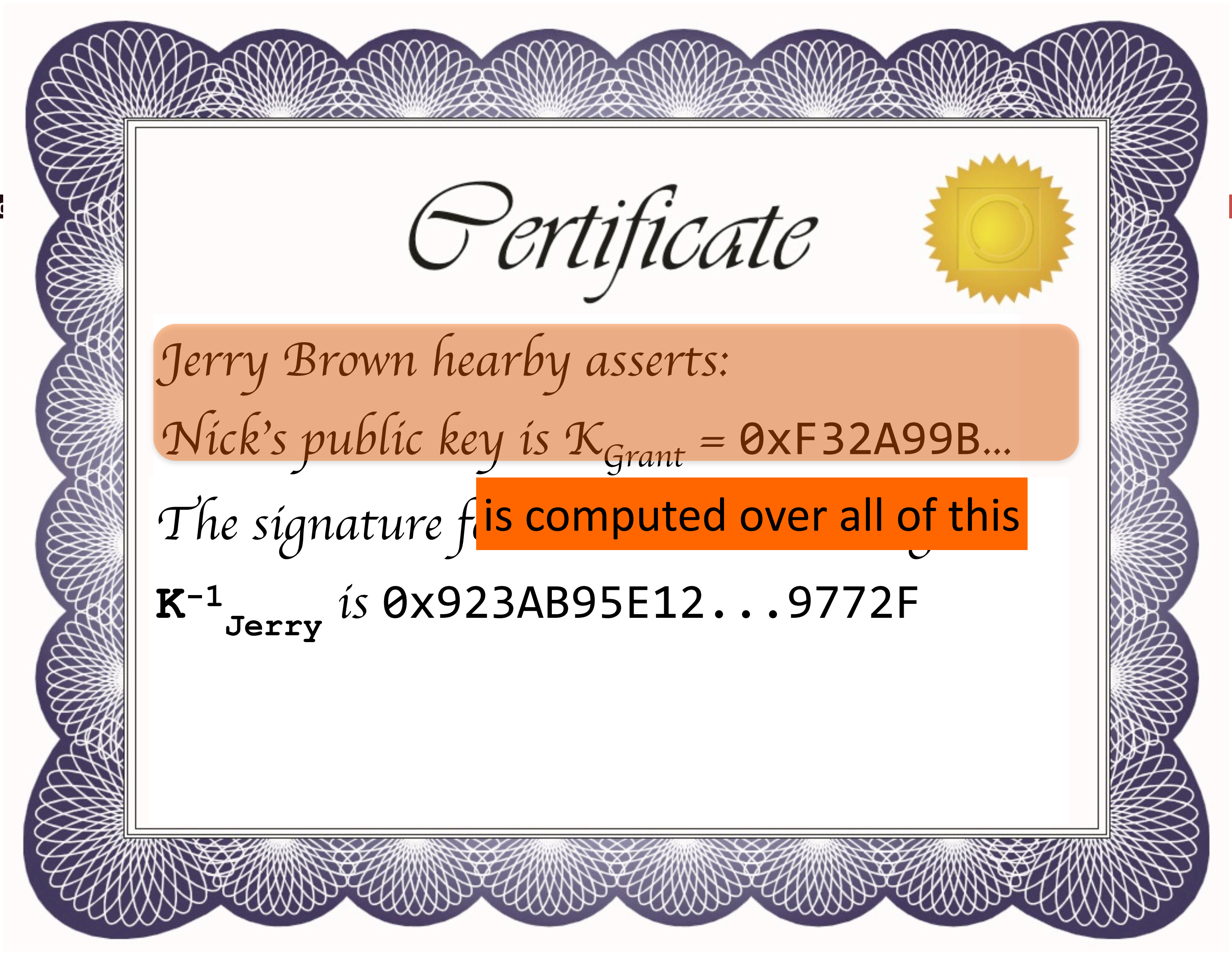
Certificate

Jerry Brown hearby asserts:

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

The signature for this statement using

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



Certificate

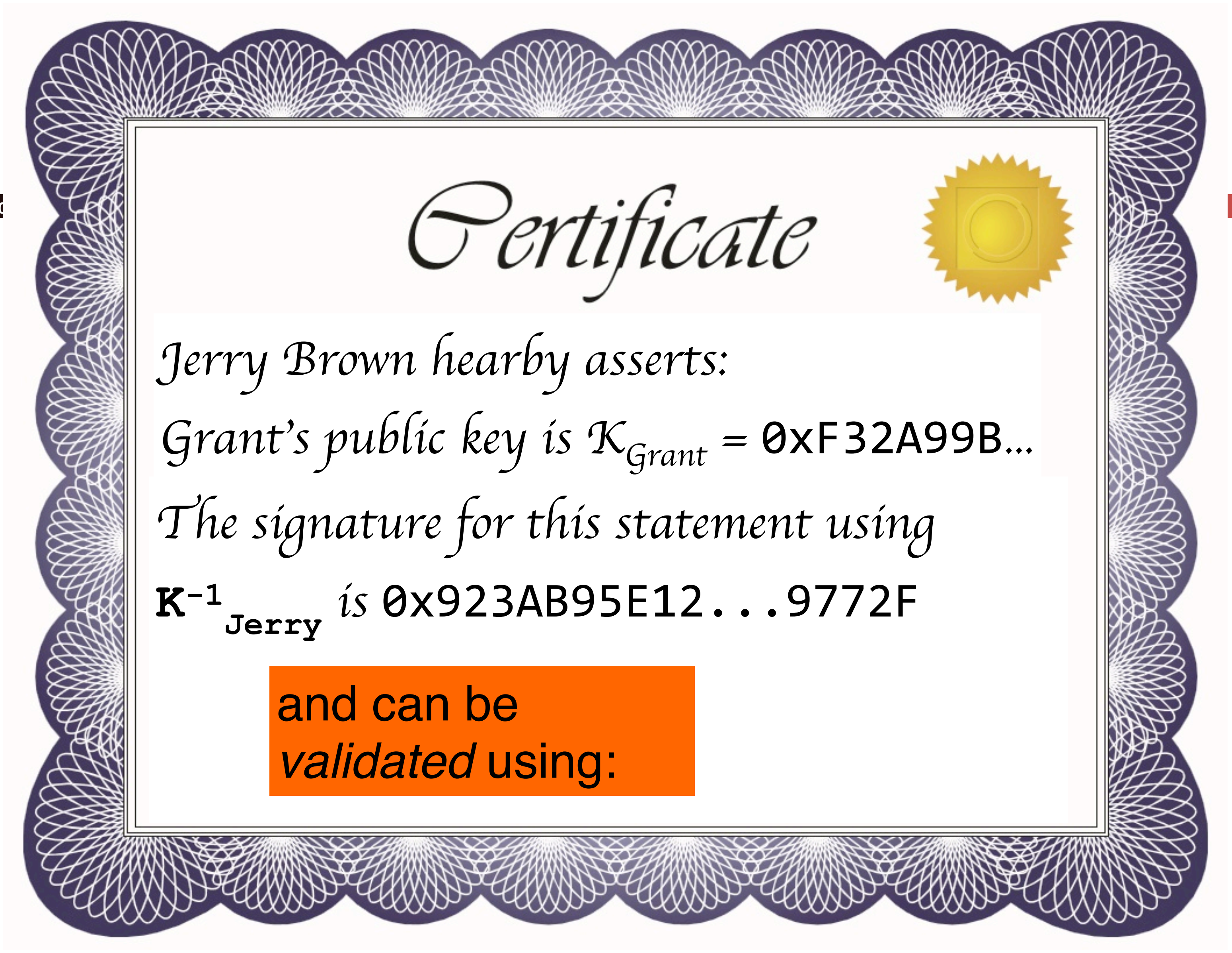
Jerry Brown hearby asserts:

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

The signature f

is computed over all of this

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



Certificate

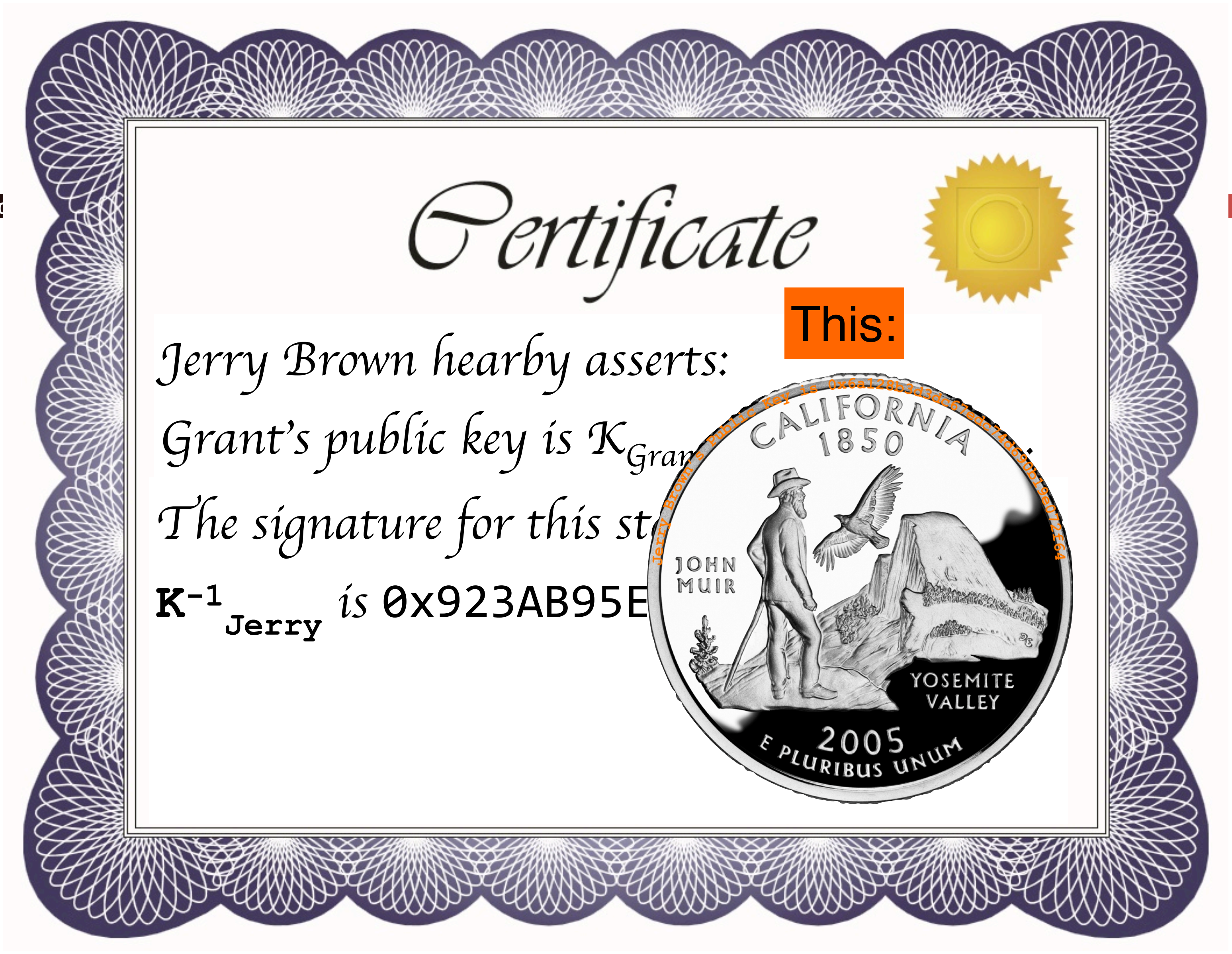
Jerry Brown hearby asserts:

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

The signature for this statement using

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

and can be *validated* using:



Certificate

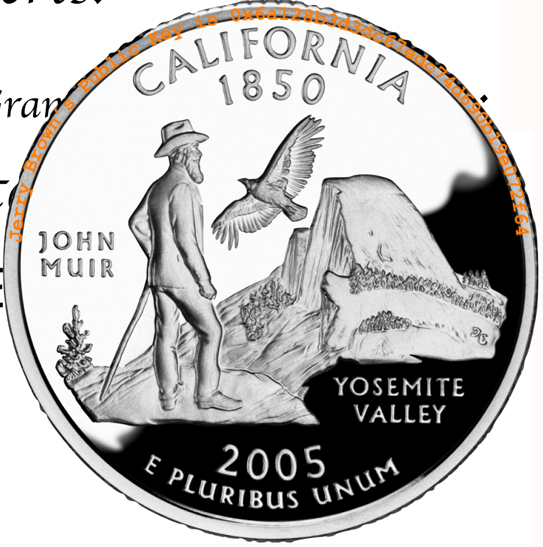
Jerry Brown hearby asserts:

Grant's public key is K_{Gran}

The signature for this st

K^{-1}_{Jerry} *is* 0x923AB95E

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 Jerry’s key, then whether he indeed signed the statement
  - If we trust Jerry’s decisions, then we have confidence we really have Nick's key

- Trust = ?
  - Jerry won’t willy-nilly sign such statements
  - Jerry 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_{Nick}

Scaling Digital Certificates

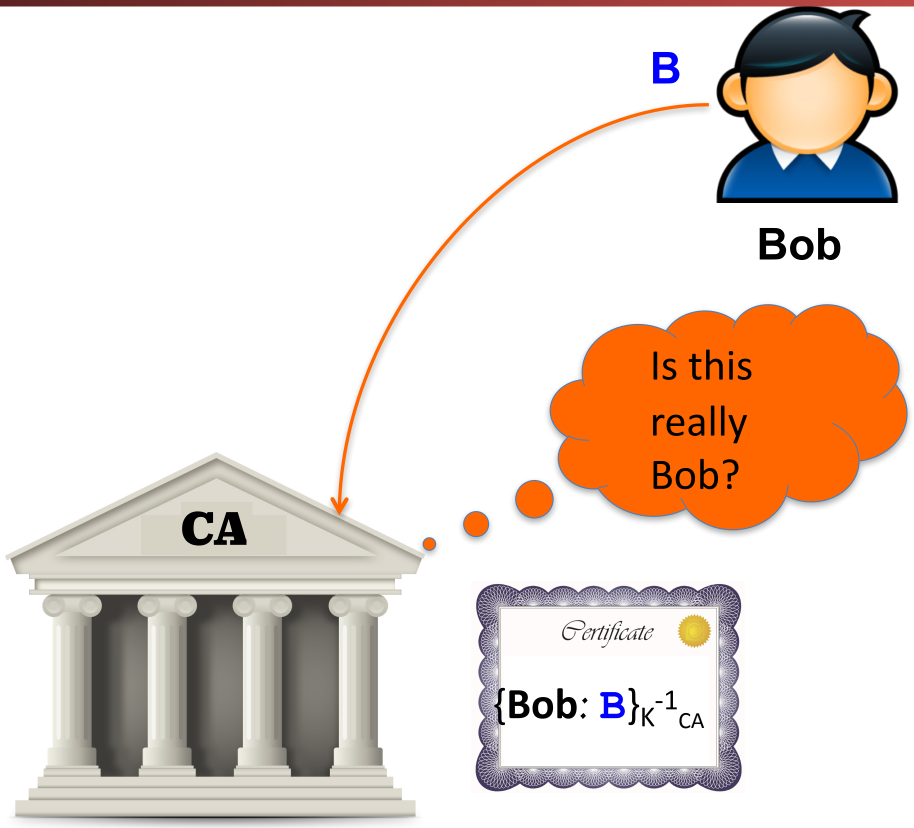
- How can this possibly scale? Surely Jerry can't sign everyone's public key!
- Approach #1: Introduce hierarchy via delegation
  - { "Janet Napolitano's public key is 0x... and I trust her to vouch for UC" }K^{-1}_{Jerry}
  - { "Nicholas Dirk's public key is 0x... and I trust him to vouch for UCB" }K^{-1}_{Janet}
  - { "Jitendra Malik's public key is 0x... and I trust him to vouch for EECS" }K^{-1}_{NickDirk}
  - { "Nick Weaver's public key is 0x..." }K^{-1}_{Jitendra}

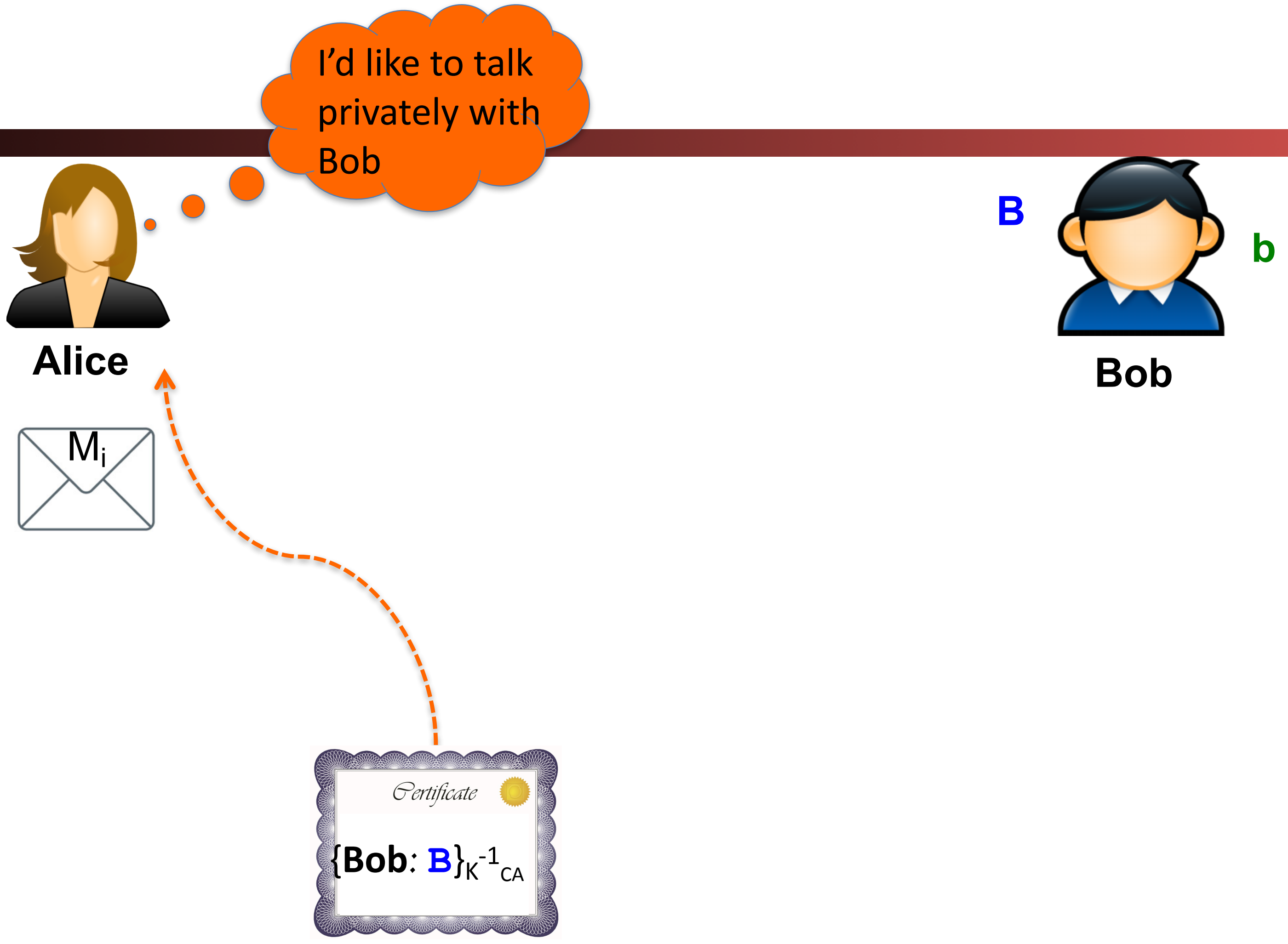
Scaling Digital Certificates, con’t

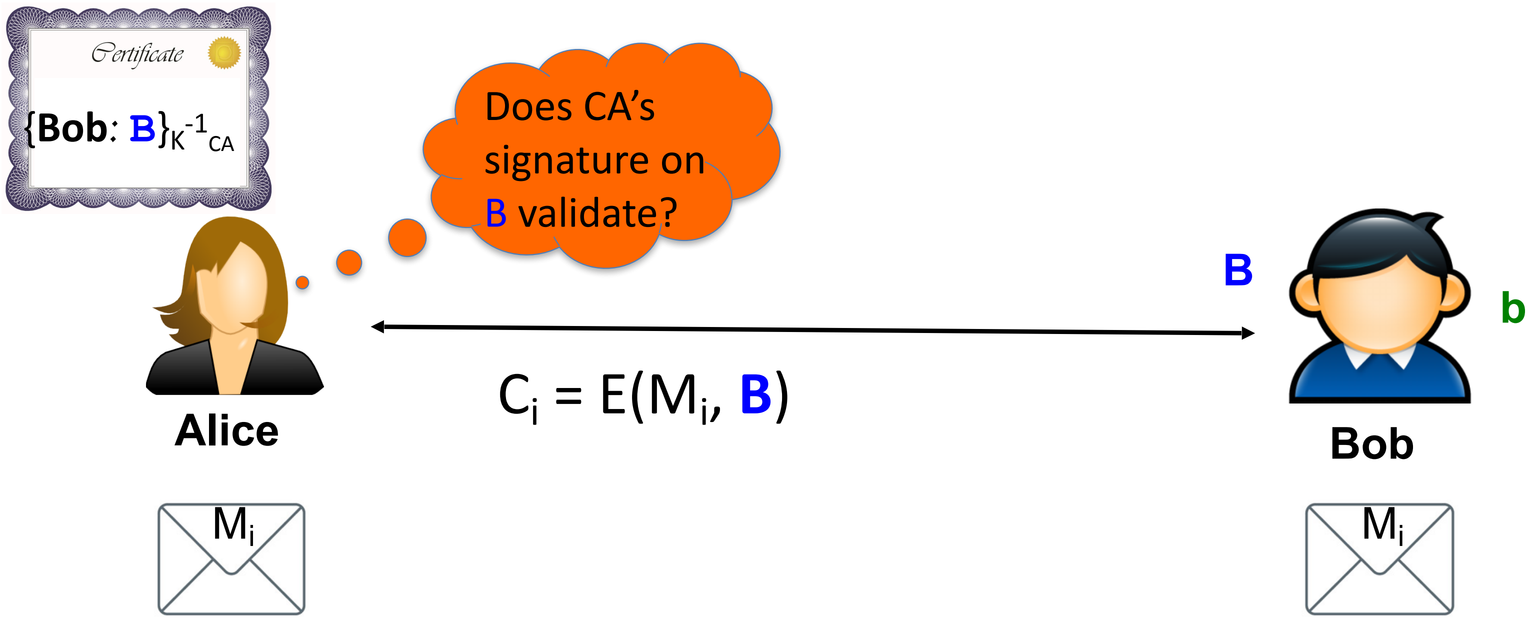
- Nick puts this last on his web page
    - (or shoves it under your door)
- Anyone who can gather the intermediary keys can validate the chain
    - They can get these (other than Jerry’s) from anywhere because they can validate them, too
- Approach #2: have multiple trusted parties who are in the business of signing certs …
    - (The certs might also be hierarchical, per Approach #1)

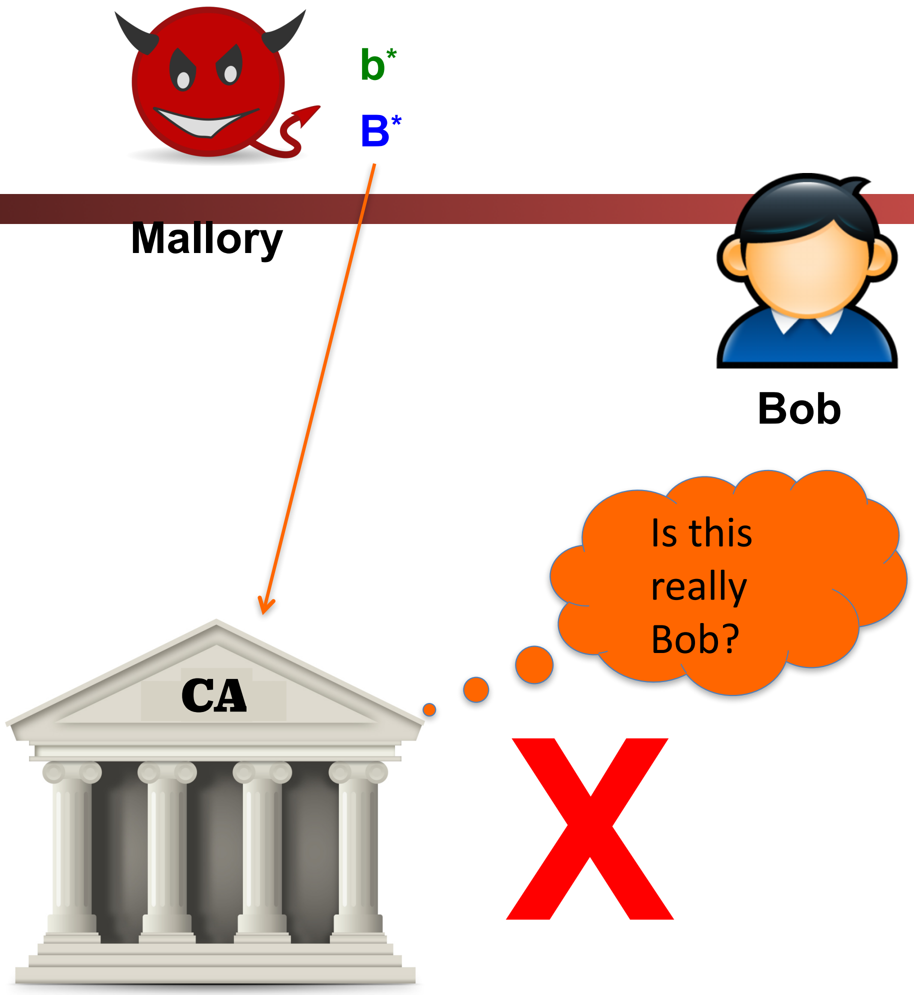
Certificate Authorities

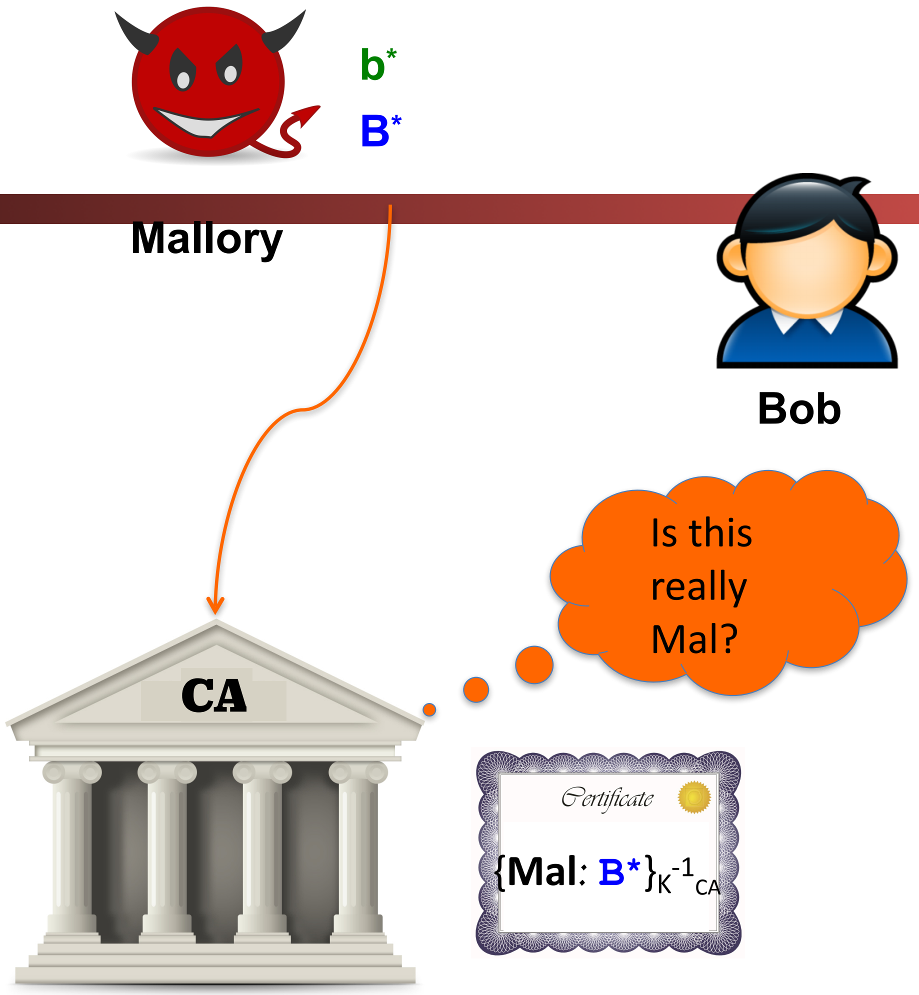
- CAs are trusted parties in a Public Key Infrastructure (PKI)
- They can operate offline
  - They sign (“cut”) certs when convenient, not on-the-fly (… though see below ...)
- Suppose Alice wants to communicate confidentially w/ Bob:
  - Bob gets a CA to issue {Bob’s public key is B} K^{-1}_{CA}
  - 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

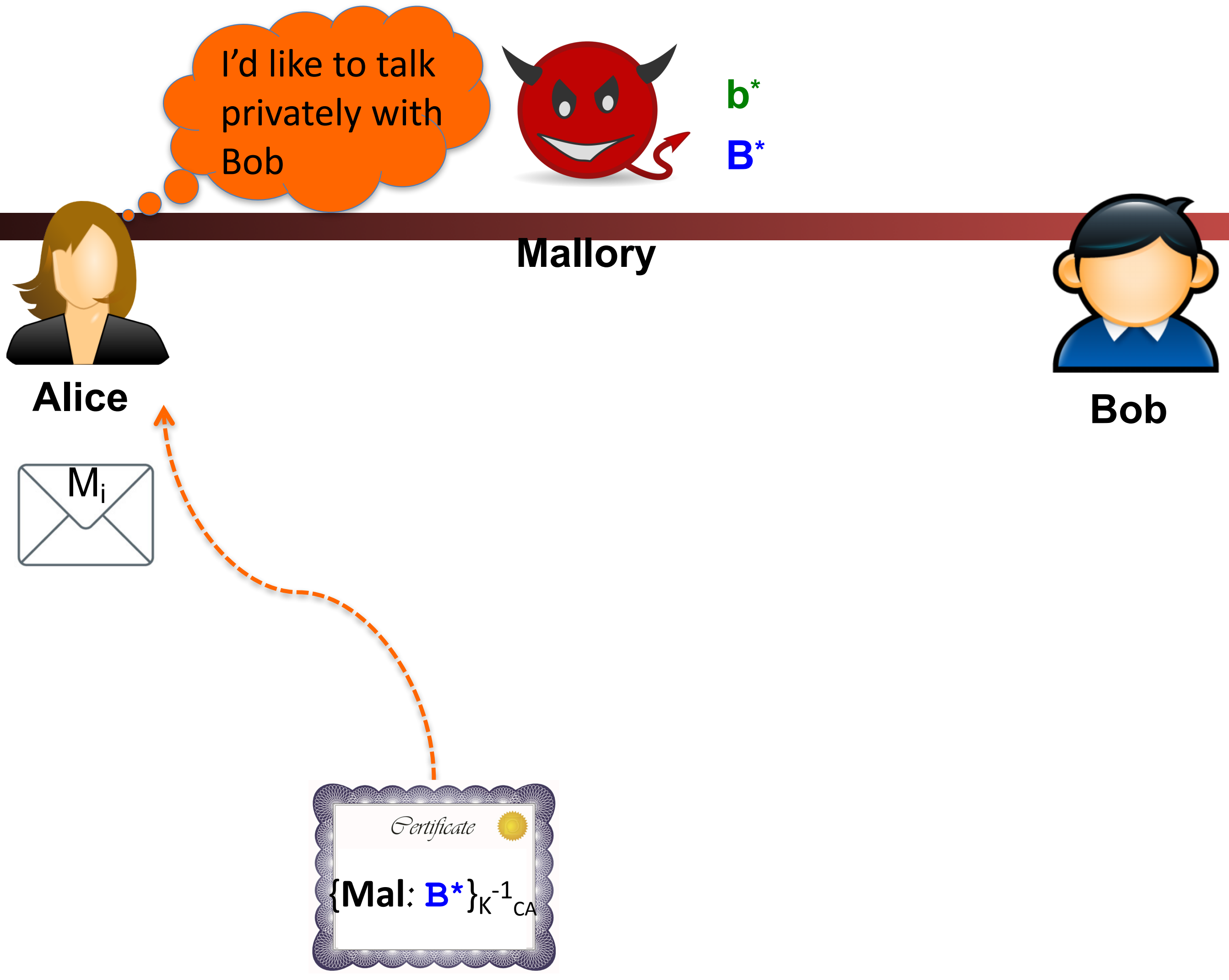

B
b
Bob
Is this really Bob?
CA
Certificate
$\{\text{Bob}: B\}_{K^{-1}_{CA}}$

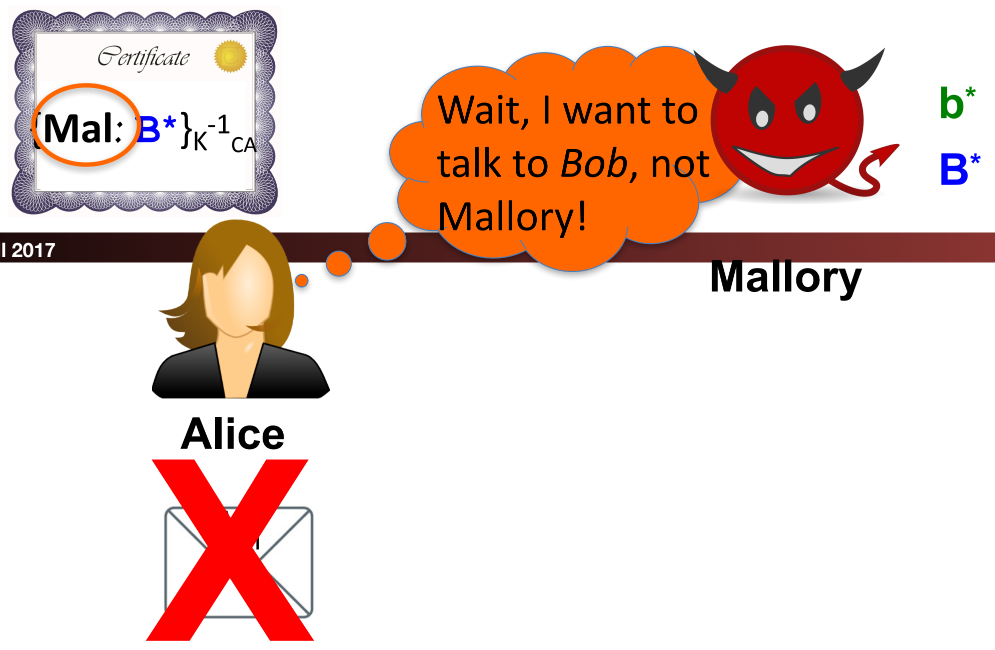

I'd like to talk privately with Bob
Alice
M_i
B
b
Bob
Certificate
$\{Bob: B\}_{K^{-1}_{CA}}$


Certificate
$\{\text{Bob}: B\}_{K^{-1}_{CA}}$
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
X

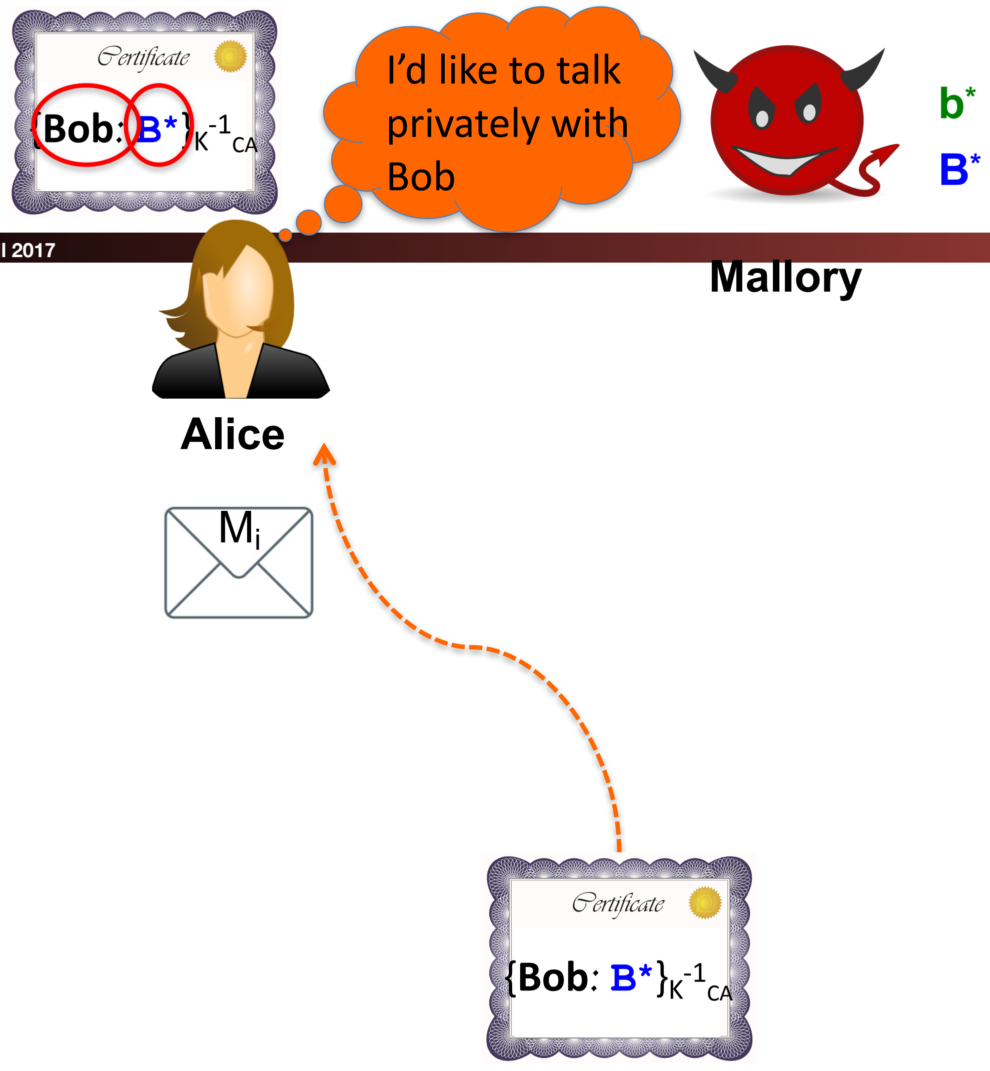

b^*
B^*
Mallory
Bob
Is this really Mal?
CA
Certificate
$\{\text{Mal}: B^*\}_{K^{-1}_{CA}}$


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


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

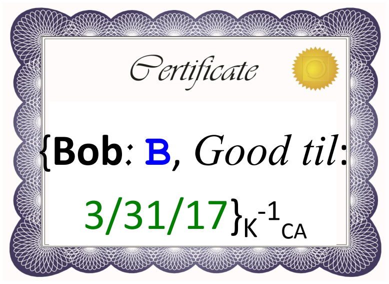
Revocation

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


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

Revocation

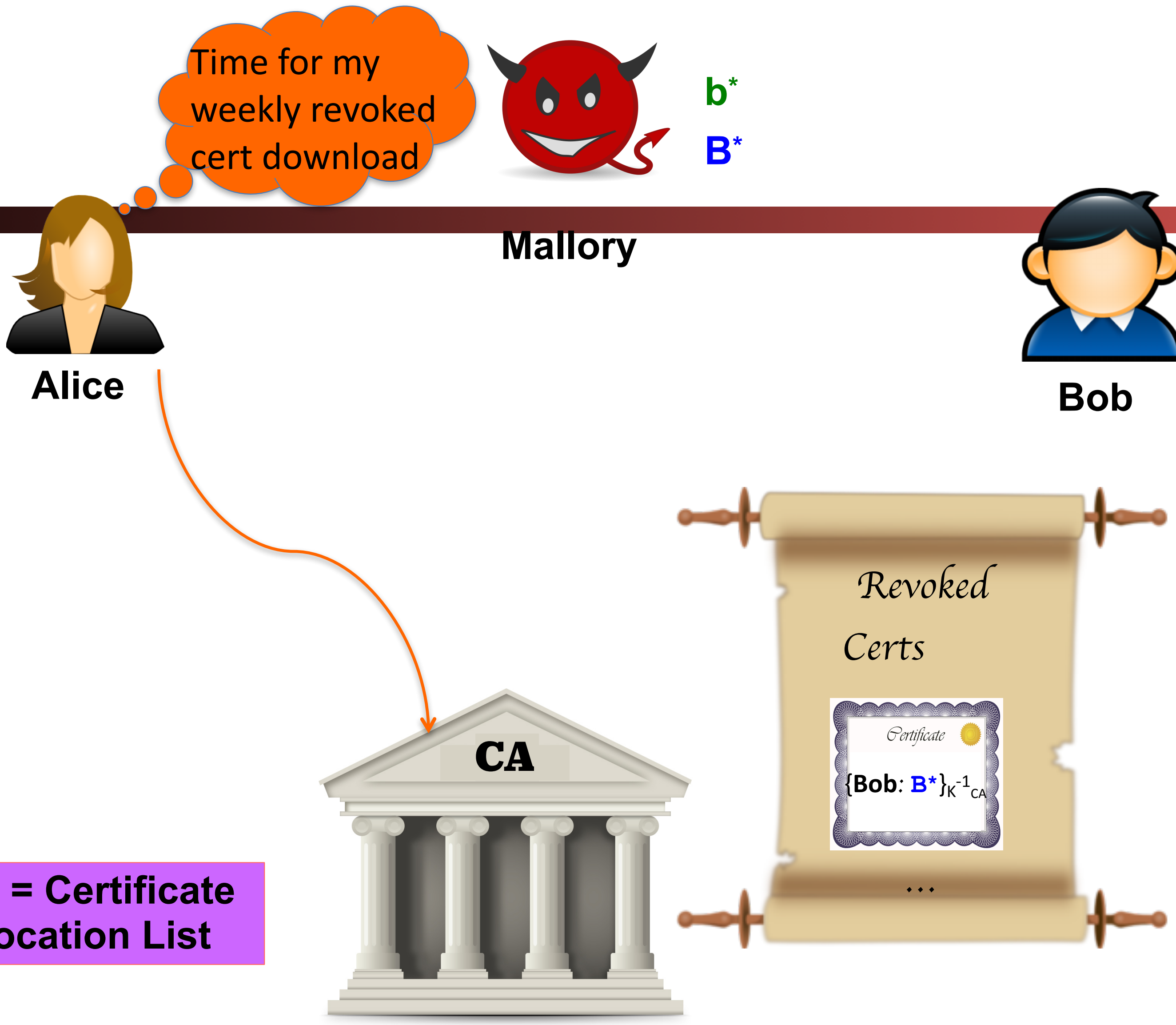
- What do we do if a CA screws up and issues a cert in Bob's name to Mallory?
  - E.g. Verisign issued a `Microsoft.com` cert to a *Random Joe*
  - (Related problem: Bob realizes b has been stolen)
- *How do we recover from the error?*
- Approach #1: expiration dates
  - Mitigates possible damage
  -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


Certificate

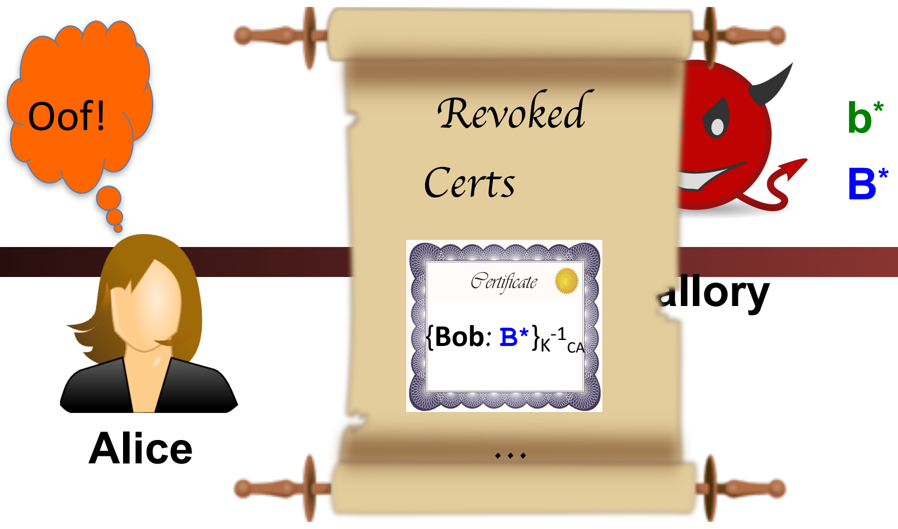
$\{\text{Bob}: \mathbf{B}, \textit{Good til}: 3/31/17\}_{K^{-1}_{CA}}$

Revocation, con’t

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


Time for my weekly revoked cert download
Mallory
b^*
B^*
Alice
Bob
CA
Revoked Certs
Certificate
$\{\text{Bob}: B^*\}_{K^{-1}_{CA}}$
...

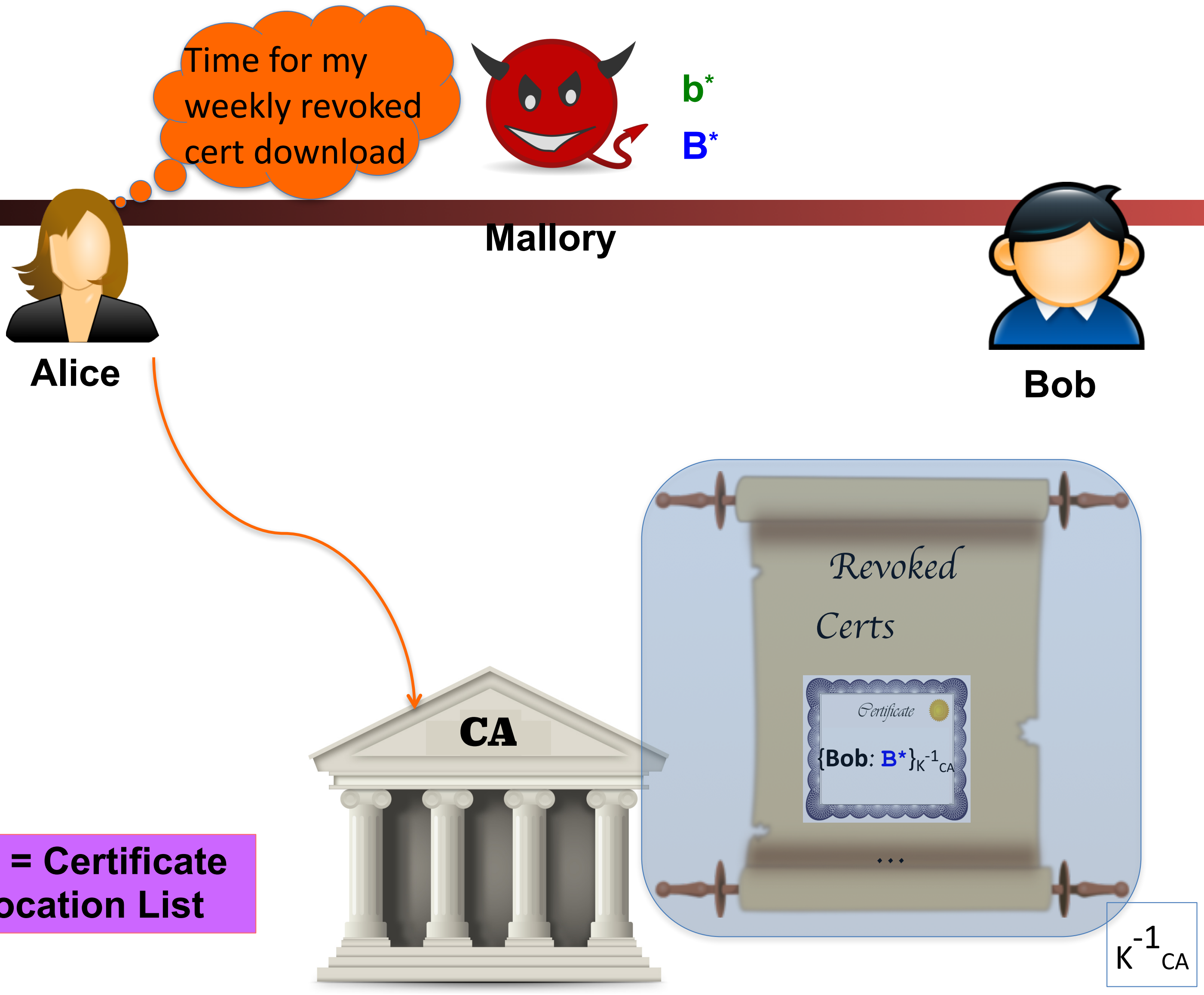
CRL = Certificate Revocation List



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
$\{Bob: B^*\}_{K^{-1}_{CA}}$
…
K^{-1}_{CA}

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)

The Facebook Problem: Applied Cryptography in Action

- Facebook Messenger now has an encrypted chat option
  - Limited to their phone application
- The cryptography in general is very good but uninteresting
  - Used a well regarded asynchronous messenger library (from Signal) with many good properties
- When Alice wants to send a message to Bob
  - Queries for Bob's public key from Facebook's server
  - Encrypts message and send it to Facebook
  - Facebook then forwards the message to Bob
- Both Alice and Bob are using encrypted and authenticated channels to Facebook

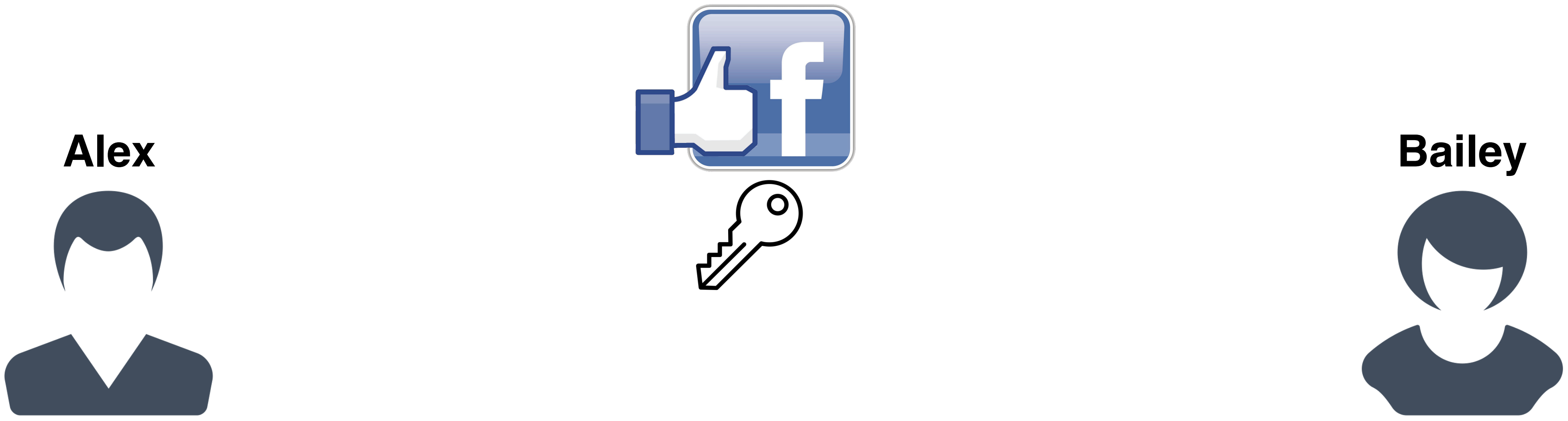
Facebook's Unique Messenger Problem: Abuse

- Much of Facebook's biggest problem is dealing with abuse...
  - What if either Alice or Bob is a stalker, an a-hole, or otherwise problematic?
    - Aside: A huge amount of abuse is explicitly gender based, so I'm going to use "Alex" as the abuser and "Bailey" as the victim through the rest of this example
- Facebook would expect the other side to complain
  - And then perhaps Facebook would kick off the perpetrator for violating Facebook's Terms of Service
- But fake abuse complaints are also a problem
  - So can't just take them on face value
- And abusers might also want to release info publicly
  - Want sender to be able to deny to the public but not to Facebook

Facebook's Problem Quantified

- Unless Bailey forwards the unencrypted message to Facebook
  - Facebook ***must not*** be able to see the contents of the message
- If Bailey does forward the unencrypted message to Facebook
  - Facebook ***must ensure*** that the message is what Alex sent to Bailey
- Nobody ***but*** Facebook should be able to verify this:
  No public signatures!
  - Critical to prevent abusive release of messages to the public being verifiable

The Protocol In Action



What Is Bailey's Public Key?

Aside: Key Transparency...

- Both Alex and Bailey are trusting Facebook's honesty...
  - What if Facebook gave Alex a different key for Bailey? How would he know?
- Facebook messenger has a ***nearly*** hidden option which allows Alex to see Bailey's key
  - If they ever get together, they can manually verify that Facebook was honest
- The mantra of central key servers: ***Trust but Verify***
  - The simple option is enough to force honesty, as each attempt to lie has some probability of being caught
- This is the biggest weakness of Apple iMessage:
  - iMessage has (fairly) good cryptography but there is no way to verify Apple's honesty

The Protocol In Action

Alex

Bailey

```
{message=E(K_pub_b,
  M={"Hey Bailey I'm going to
     say something abusive",
     k_rand}),
 mac=HMAC(k_rand, M),
 to=Bailey}
```

```
{message=E(K_pub_b,
  M={"Hey Bailey I'm going to
     say something abusive",
     k_rand}),
 mac=HMAC(k_rand, M),
 to=Bailey,
 from=Alex,
 time=now,
 fbmac=HMAC(K_fb,{mac, from,
                  to, time})}
```

Some Notes

- Facebook ***can not*** read the message or even verify Alex's HMAC
  - As the key for the HMAC is in the message itself
- Only Facebook knows their HMAC key
  - And its the only information Facebook ***needs*** to retain in this protocol: Everything else can be discarded
- Bailey upon receipt checks that Alex's HMAC is correct
  - Otherwise Bailey's messenger silently rejects the message
    - Forces Alex's messenger to be honest about the HMAC, even thought Facebook never verified it
- Bailey trusts Facebook when Facebook says the message is from Alex
  - Bailey does ***not verify*** a signature, because there is no signature to verify

Now To Report Abuse

Alex

Bailey

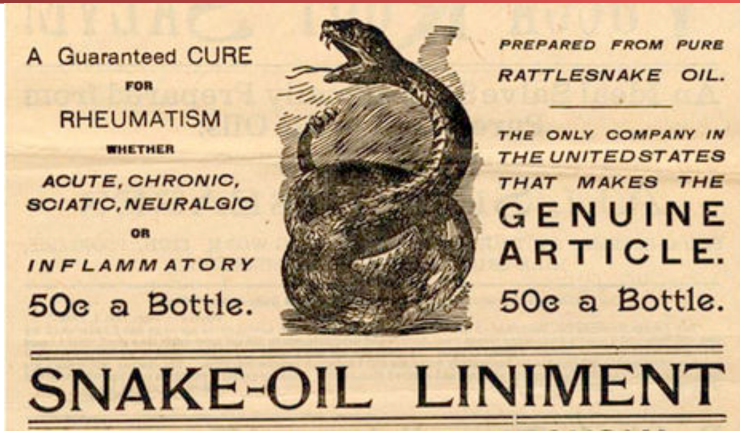
```
{Abuse{
  M={"Hey Bailey I'm going to
      say something abusive",
      k_rand}},
 mac=HMAC(k_rand, M),
 to=Bailey,
 from=Alex,
 time=now,
 fbmac=HMAC(K_fb,{mac, from,
                       to, time})}
```

Facebook's Verification

- First verify that Bailey correctly reported the message sent
  - Verify **`fbmac=HMAC(`**K_{fb}**`,{mac,from,to,time})`**
    - Only Facebook can do this verification since they keep K_{fb} secret
  - This enables Facebook to confirm that this is the message that it relayed from Alex to Bailey
- Then verify that Bailey didn't tamper with the message
  - Verify **`mac=HMAC(`**k_{rand}**`,{M, `**k_{rand}**`})`**
- Now Facebook knows this was sent from Alex to Bailey and can act accordingly
  - But Bailey can't prove that Alex sent this message to anyone other than Facebook
  - And Bailey can't tamper with the message because the HMAC is also a hash

Snake Oil Cryptography: Craptography

- "Snake Oil" refers to 19th century fraudulent "cures"
  - Promises to cure practically every ailment
  - Sold because there was no regulation and no way for the buyers to know


A Guaranteed CURE
FOR
RHEUMATISM
WHETHER
ACUTE, CHRONIC, SCIATIC, NEURALGIC
OR
INFLAMMATORY
50c a Bottle.
PREPARED FROM PURE RATTLESNAKE OIL.
THE ONLY COMPANY IN THE UNITED STATES THAT MAKES THE GENUINE ARTICLE.
50c a Bottle.
SNAKE-OIL LINIMENT

- The security field is practically ***full*** of Snake Oil Security and Snake Oil Cryptography
  - https://www.schneier.com/crypto-gram/archives/1999/0215.html#snakeoil

Anti-Snake Oil: NSA's CNSA cryptographic suite

- Successor to "Suite B"
  - Unclassified algorithms approved for Top Secret:
    - There is nothing higher than TS, you have "compartments" but those are access control modifiers
    - https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm
  - Symmetric key, AES: 256b keys
  - Hashing, SHA-384
  - RSA/Diffie Helman: >= 3072b keys
  - ECDHE/ECDSA: 384b keys over curve P-384
- In an ideal world, I'd only use those parameters,
  - But a lot of "strong" commercial is 128b AES, SHA-256, 2048b RSA/DH, 256b elliptic curves, plus the DJB curves and cyphers (ChaCha20)
  - NSA has a requirement where a Top Secret communication captured today should not be decryptable by an adversary 40 years from now!

Snake Oil Warning Signs...

- Amazingly long key lengths
  - The NSA is super paranoid, and even they don't use >256b keys for symmetric key or >4096b for RSA/DH public key
  - So if a system claims super long keys, be suspicious
- New algorithms and crazy protocols
  - There is ***no reason*** to use a novel block cipher, hash, public key algorithm, or protocol
    - Even a "post quantum" public key algorithm should not be used alone: Combine it with a conventional public key algorithm
  - Anyone who roles their own is asking for trouble!
  - EG, Telegram
    - "It's like someone who had never seen cake but heard it described tried to bake one. With thumbtacks and iron filings." Matthew D Green
    - "Exactly! GLaDOS-cake encryption. Odd ingredients; strange recipe; probably not tasty; may explode oven. :)" Alyssa Rowan

Snake Oil Warning Signs...

- "One Time Pads"
  - One time pads are secure, if you actually have a true one time pad
  - But almost all the snake oil advertising it as a "one time pad" isn't!
  - Instead, they are invariably some wacky stream cypher
- Gobbledygook, new math, and "chaos"
  - Kinda obvious, but such things are never a good sign
- Rigged "cracking contests"
  - Usually "decrypt this message" with no context and no structure
    - Almost invariably a single or a few unknown plaintexts with nothing else
  - Again, Telegram, I'm looking at you here!

Unusability: No Public Keys

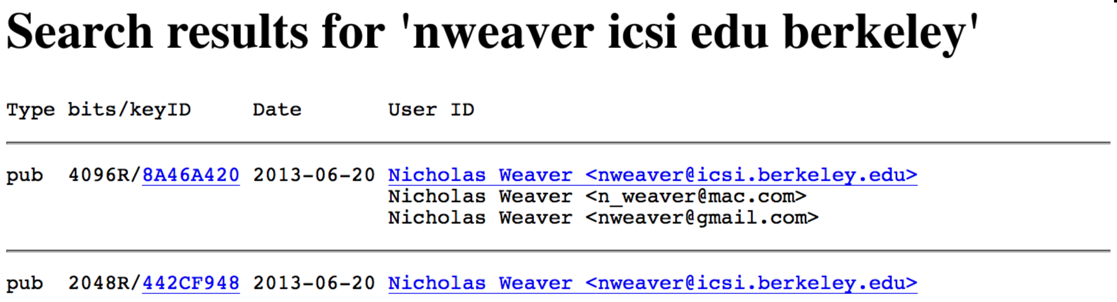
- The APCO Project 25 radio protocol
  - Supports encryption on each traffic group
    - But each traffic group uses a single ***shared*** key
- All fine and good if you set everything up at once...
  - You just load the same key into all the radios
  - But this totally fails in practice: what happens when you need to coordinate with s who doesn't have the same keys?
- Made worse by bad user interface and users who think rekeyin frequently is a good idea
  - If your crypto is good, you shouldn't need to change your crypto keys
- "Why (Special Agent) Johnny (Still) Can't Encrypt
  - http://www.crypto.com/blog/p25

Unusability: PGP

- I ***hate*** Pretty Good Privacy
  - But not because of the cryptography...
- The PGP cryptography is decent...
  - Except it lacks "Forward Secrecy":
    If I can get someone's private key I can decrypt all their old messages
- The metadata is awful...
  - By default, PGP says who every message is from and to
    - It makes it much faster to decrypt
  - It is hard to hide metadata well, but its easy to do things better than what PGP does
- It is never transparent
  - Even with a "good" client like GPG-tools on the Mac
  - And I don't have a client on my cellphone

Unusability:
How do you find someone's PGP key?

- Go to their personal website?
- Check their personal email?
- Ask them to mail it to you
  - In an unencrypted channel?
- Check on the MIT keyserver?
  - And get the old key that was mistakenly uploaded and can never be removed?

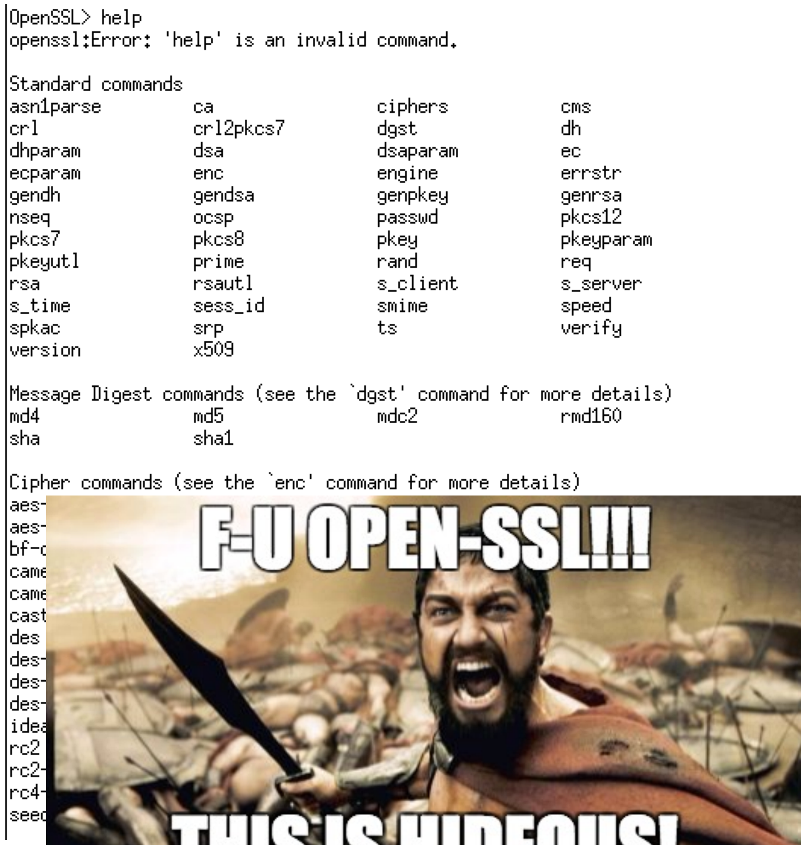


Search results for 'nweaver icsi edu berkeley'

```
Type bits/keyID     Date       User ID
------------------------------------------------------------------------------
pub  4096R/8A46A420 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
                               Nicholas Weaver <n_weaver@mac.com>
                               Nicholas Weaver <nweaver@gmail.com>
------------------------------------------------------------------------------
pub  2048R/442CF948 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
```

Unusable:
openssl libcrypto and libssl

- OpenSSL is a nightmare...
  - A gazillion different little functions needed to do anything
- So much of a nightmare that I'm not going to bother learning it to teach you how bad it is
  - This is why last semester's python-based project didn't give this raw
- But just to give you an idea:
  The command line OpenSSL utility options:

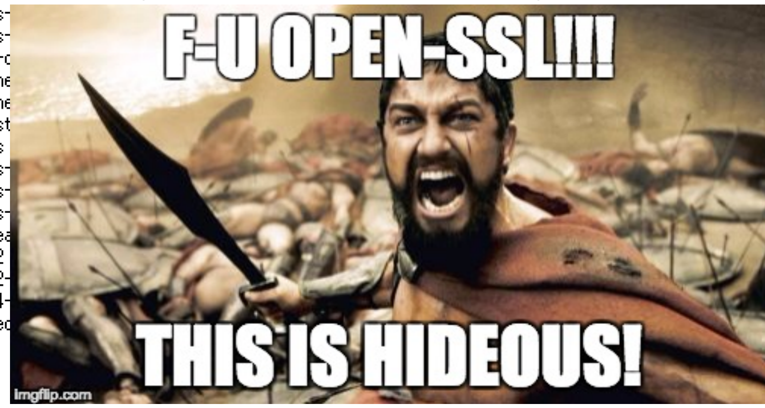


```
OpenSSL> help
openssl:Error: 'help' is an invalid command.

Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst              dh
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
gendh             gendsa            genpkey           genrsa
nseq              ocsp              passwd            pkcs12
pkcs7             pkcs8             pkey              pkeyparam
pkeyutl           prime             rand              req
rsa               rsautl            s_client          s_server
s_time            sess_id           smime             speed
spkac             srp               ts                verify
version           x509

Message Digest commands (see the `dgst' command for more details)
md4               md5               mdc2              rmd160
sha               sha1

Cipher commands (see the `enc' command for more details)
aes-
aes-
bf-c
came
came
cast
des
des-
des-
des-
idea
rc2
rc2-
rc4-
seed
```



Bitcoin's Goal

- A decentralized, distributed digital currency
  - Decentralized: ***no point of authority or control***
  - Distributed: ***lots of independent systems, no central point of trust***
  - Digital Currency: ***Just that, a currency***
- Bitcoin is ***censorship resistant money***:
  - Nobody can say "don't spend your money on X"
- Bitcoin's Crypto: Interesting
  - So I will talk about it
- Bitcoin's Economics: Broken
- Bitcoin's Community: Bat-Shit Insane
  - So I won't bother wasting people's time. This is a subject for a Beer Rant, not a lecture

Bitcoin's Public Key Signature Algorithm ECDSA

- Elliptic Curve Digital Signature Algorithm
  - So different math but conceptually similar to El Gamal and DSA
- 256b private key (32 bytes)
  - Public key is 65 bytes
- Bitcoin "address" is not the public key but the ***hash*** of the public key
  - RIPEMD-160(SHA-256(K_{pub}))
    - Why double hashing? Its a common weirdness in Bitcoin.
  - After adding a checksum and Base 58 encoding you get a "Bitcoin address" of type 1 you can send money to
  - 1FuckBTCqwBQexxs9jiuWTiZeoKfSo9Vyi is a valid address
    - I spent a lot of CPU time randomly generating private keys to find one that would match the desired prefix

Interesting Implications of Hashed Public Keys

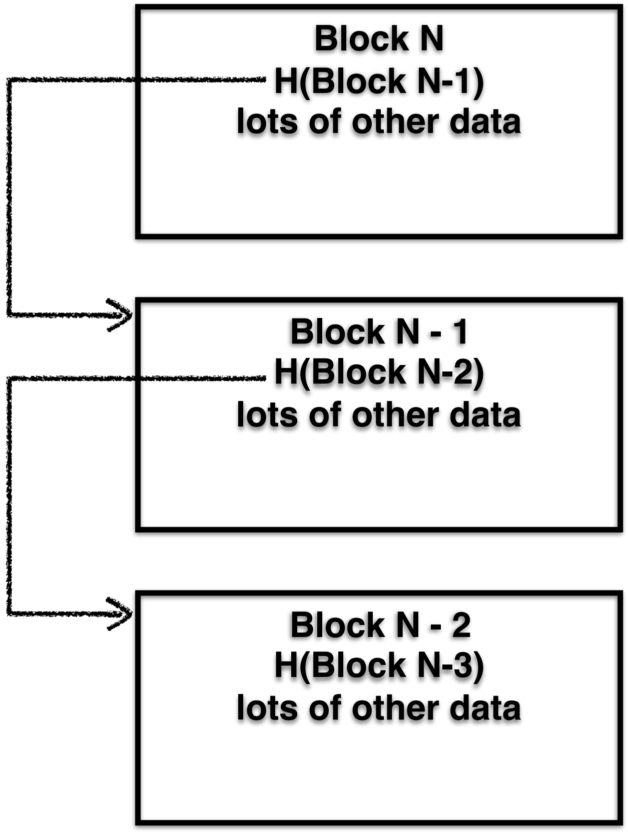
- The ECDSA public key is twice as large as the private key
  - So hashing makes the public key a lot smaller
  - But it makes the signatures themselves larger
    - Since any signature also needs to include the full public key
- Validation of a signature becomes a 2-part process
  - Validate that **$H(K_{pub})$ = Address**
  - Validate that the signature is valid
- But if a private key is only used ***once***, attacks which require the public key in advance can not work!

Why This Matters: Quantum Computing

- A Quantum computer rips through elliptic curve schemes as well as classic discrete log (Diffie/Hellman) and RSA type schemes
  - Given the public key it is trivial to find the private key
    - Since the private key controls money, this would be catastrophic
  - But at the same time, we don't know how to build a quantum computer big enough to factor a number much larger than 15
- If you ***never*** use a private key more than once…
  - By instead transferring all unspent money to a ***new*** random private key
  - A Quantum Computer can't steal your money if it can't come up with a solution before your spending is recorded!
- Many cryptographic systems need to worry today about Quantum computers which don't yet exist.

Hash Chains

- If a data structure includes a hash of the previous block of data
  - This forms a “hash chain”
- So rather than the hash of a block validating just the block
  - The inclusion of the previous block’s hash validates all the previous blocks
- This also makes it easy to add blocks to data structures
  - Only need to hash block + hash of previous block, rather than rehash everything:
    How you can efficiently hash an "append only" datastructure



Merkle Trees

- Lets say you have a lot of elements
  - And you want to add or modify elements
- And you want to make the hash of the set easy to update
- Enter hash trees/merkle trees
  - Elements 0, 1, 2, 3, 4, 5...
  - H(0), H(1), H(2)...
  - H(H(0) + H(1)), H(H(2)+H(3))...
  - The final hash is the root of the top of the tree.
- And so on until you get to the root
  - Allows you to add an element and update lg(n) hashes
    Rather than having to rehash all the data

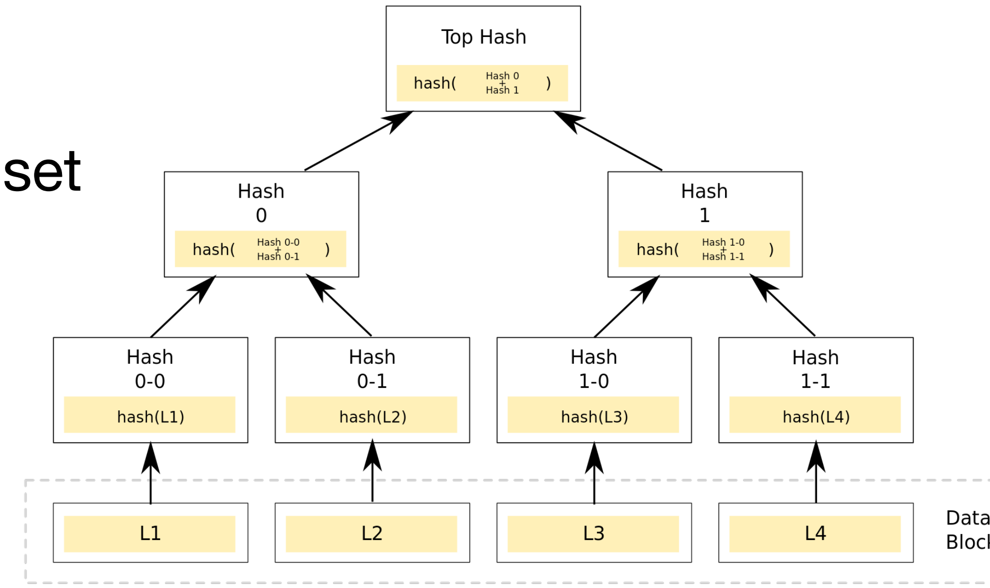


Image Stolen from Wikipedia

Proof of Work To Establish History

- Idea: If creating a block requires so much effort
  - And it includes a pointer to all previous blocks
  - Changing history becomes expensive:
    - To rewrite the last ***k*** blocks of history requires the same amount of effort as recording those ***k*** blocks the first time around
  - But at the same time, it ***must*** be cheap to ***verify*** the work was done
- Easy proof of work: generation ***partial*** hash collisions
  - If the first ***N*** bits of a hash have to be zero…
    - You are expected to need to try 2^N times to find a collision
    - But you only need to do a single hash invocation to ***check*** if someone else did the work

Taken Together this creates Bitcoin

- Every Bitcoin address (**$H(K_{pub})$**) has a corresponding balance in a public ledger (the Blockchain)
- To spend Bitcoin…
  - Sign a message saying "*Pay to address A*"
    - Signature includes the address it is coming from
  - Broadcast that message through the Bitcoin P2P network
- The rest of the P2P network…
  - Confirms that both the signature is valid and the balance exists
  - Then attempts to "mine" it into a new block on the Blockchain
    - This acts to ***confirm*** the transaction

Bitcoin Transactions

- A transaction consists of one or more inputs and 0 or more outputs
  - Each input refers to a single unspent transaction output:
    the input spends the ***entire*** output in the transaction
    - Each input is signed by the corresponding private key and includes the public key
  - Each output simply refers to a destination address and amount
    - If you want to make change, just send that to a new destination address or send it back to one of the input addresses
  - **Sum(outputs) <= Sum(inputs)**
    - Any extra is paid to whoever mines the block (the Transaction Fee)
- Validating transactions:
  - All inputs must refer to ***previously unspent outputs***
    - No double-spending, but requires knowing ALL previous Bitcoin transactions to validate!
  - All inputs must cryptographically validate

The Blockchain… Protected by Proof of Work

- All Bitcoin miners take all unverified transactions they want and compose them into a single block
  - Block header contains a timestamp, a nonce, the hash of the previous block, and the hash of all transactions for this block
    - Transactions are hashed in a Merkle tree to make it easy to add transactions to the block in progress
- Now all the miners try to find a hash collision:
  - Modifying the block so that **H(Block)** < “difficulty” value
    - First by modifying the nonce value and/or timestamp and then modifying the coinbase, a string in the "pay from" for the first transaction
- Once one finds a hash collision, it broadcasts the new block to the entire Bitcoin network
  - Every other miner first verifies that block and then starts working on the next block
- Rule is always trust the longest chain
  - Now to rewrite history to depth ***N*** it takes the same amount of work as used to generate the chain you are rewriting
  - But at the same time, the current chain keeps growing!

The Coinbase Transaction

- The first transaction in any block is special
  - It actually has 0 inputs, instead it has a small amount of arbitrary data called the "coinbase"
- The coinbase data serves two purposes:
  - It allows the miner to make a comment
    - EG, claim credit, vote on proposals, etc
  - It can be easily changed for searching for hash collisions
    - When changing the coinbase the miner needs to update the Merkel tree but that's relatively cheap
- The output of this transaction is the miner's reward
  - The miner fills it out as "pay to me"
    - Both the current block reward (now at 12.5 BTC/block) and any value not otherwise spent

Bitcoin Balances

- Each address has a balance associated with it
  - The balance is in “Satoshi”, a fixed-point value = 0.00000001 BTC
    - There have been Bitcoin systems with bugs related to fixed vs floating point issues
- This is actually the sum of all unspent outputs sent to this address
  - Calculating an address's balance requires looking at ***every*** Bitcoin transaction ever done
- This is a ***problem!***
  - Bitcoin requires knowing every transaction from the dawn of the Blockchain in order to know that things are valid
    - And currently this data grows by 1 MB every 10 minutes!