



University of
Pittsburgh

Applied Cryptography and Network Security

CS 1653



Summer 2023
Sherif Khattab
ksm73@pitt.edu

(Slides are adapted from Prof. Adam Lee's CS1653 slides.)

Announcements

- Homework 4 due this Friday @ 11:59 pm
- Project Phase 2 due on Friday 6/30 @ 11:59 pm
- Makeup lecture on Friday 6/23 from 3:00-4:15 pm

RSA Recap

Key generation:

- ✓ • Choose two large prime numbers p and q , compute $n = pq$
- ✓ • Compute $\phi(n) = (p - 1)(q - 1)$
- ✓ • Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✓ • Calculate d such that $ed \equiv 1 \pmod{\phi(n)}$
 - **Public key:** n, e
 - **Private key:** p, q, d

Usage:

- ✓ • Encryption: $M^e \pmod{n}$
- ✓ • Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

But why is RSA *safe* to use?

Now, why exactly is RSA safe to use?

In the original RSA paper*, the authors identify four avenues for attacking the mathematics behind RSA

1. Factoring n to find p and q
2. Determining $\phi(n)$ without factoring n
3. Determining d without factoring n or learning $\phi(n)$
4. Learning to take e^{th} roots modulo n

As it turns out, all of these attacks are thought to be hard to do

- But you shouldn't take our word for it...
- Let's see why!

*R.L. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21(2): 120-126, Feb. 1978.

It turns out that factoring is a hard* problem

First of all, why is factoring an issue?

- n is the public modulus of the RSA algorithm
 - If we can factor n to find p and q , we can compute $\phi(n)$
 - Given $\phi(n)$ and e , we can easily compute the decryption exponent d
-

Fortunately, mathematicians believe that factoring numbers is a very difficult problem. History backs up this belief.

The fastest general-purpose algorithm for integer factorization is called the **general number field sieve**. This algorithm has running time:

$$O\left(e^{(c+o(1))}(\log n)^{\frac{1}{3}}(\log \log n)^{\frac{2}{3}}\right)$$

Note: This running time is **sub-exponential**

- i.e., Factoring can be done faster than brute force
- This explains why RSA keys are larger than AES keys
 - RSA: Typically 2048-4096 bits
 - AES: Typically 128 bits

What about computing $\phi(n)$ without factoring?

Question: Why would the ability to compute $\phi(n)$ be a bad thing?

- It would allow us to easily compute d , since $ed \equiv 1 \pmod{\phi(n)}$

Good news: If we can compute $\phi(n)$, it will allow us to factor n

- **Note 1:** $\phi(n) = n - p - q + 1$
 $= n - (p + q) + 1$

- Rewriting gives us $(p + q) = n - \phi(n) + 1$

- **Note 2:** $(p - q) = \sqrt{(p + q)^2 - 4n}$

- **Note 3:** $(p + q) - (p - q) = 2q$

- Finally, given q and n , we can easily compute p

$$\begin{aligned}(p + q)^2 - 4n &= p^2 + 2pq + q^2 - 4n \\ &= p^2 + 2pq + q^2 - 4pq \\ &= p^2 - 2pq + q^2 \\ &= (p - q)^2\end{aligned}$$

What does this mean?

- If factoring is actually hard, then so is computing $\phi(n)$ without factoring
- (Recall the concept of **reduction**)

What about computing d without factoring n or knowing $\phi(n)$?

As it turns out, if we can figure out d without knowing $\phi(n)$ and without factoring n , d can be used to help us factor n

Given d , we can compute $ed - 1$, since we know e

Note: $ed - 1$ is a multiple of $\phi(n)$

- $ed \equiv 1 \pmod{\phi(n)}$
- $ed = 1 + k\phi(n)$
- $ed - 1 = k\phi(n)$ ✓

It has been shown that n can be **efficiently** factored using any multiple of $\phi(n)$. As such, if we know e and d , we can efficiently factor n .

Are there any other attacks that we need to worry about?

Recall: $C = M^e \bmod n$

- e is part of the public key, so the adversary knows this
- If we could compute e^{th} roots mod n , we could decrypt without d

It is not known whether breaking RSA yields an efficient factoring algorithm, but the inventors **conjecture** that this is the case

- This conjecture was made in 1978
- To date, it has neither been proved or disproved

Conclusion: *Odds are that breaking RSA efficiently implies that factoring can be done efficiently. Since factoring is hard, RSA is probably safe to use.*

RSA Wrap Up

Hopefully you now have a better understanding of RSA

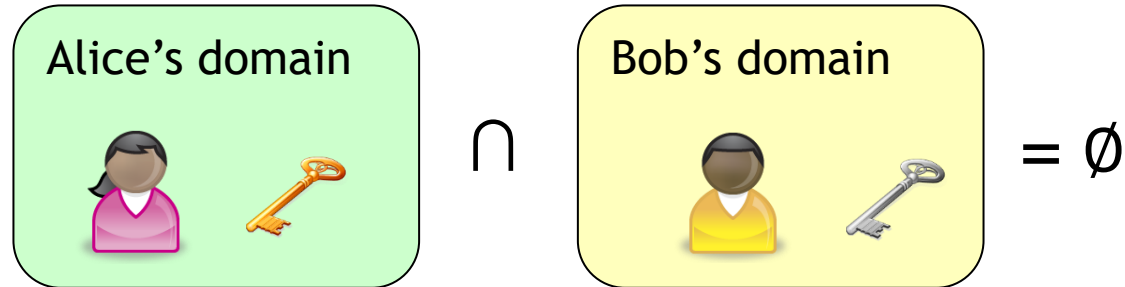
- How each step of the process works
- How these steps can be made reasonably efficient
- Why RSA is safe to use

Unfortunately, this is not the end of the story...

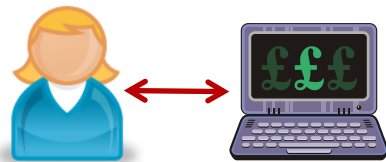
- Although theoretically secure, implementations can be broken
- We'll revisit this in a later lecture

Next: Secret sharing and threshold cryptography

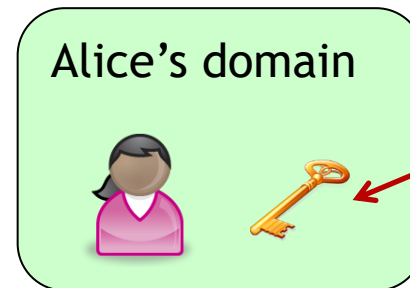
To date, we've (explicitly and implicitly) made several assumptions when discussing cryptosystems



Each private key has one owner



One device per person



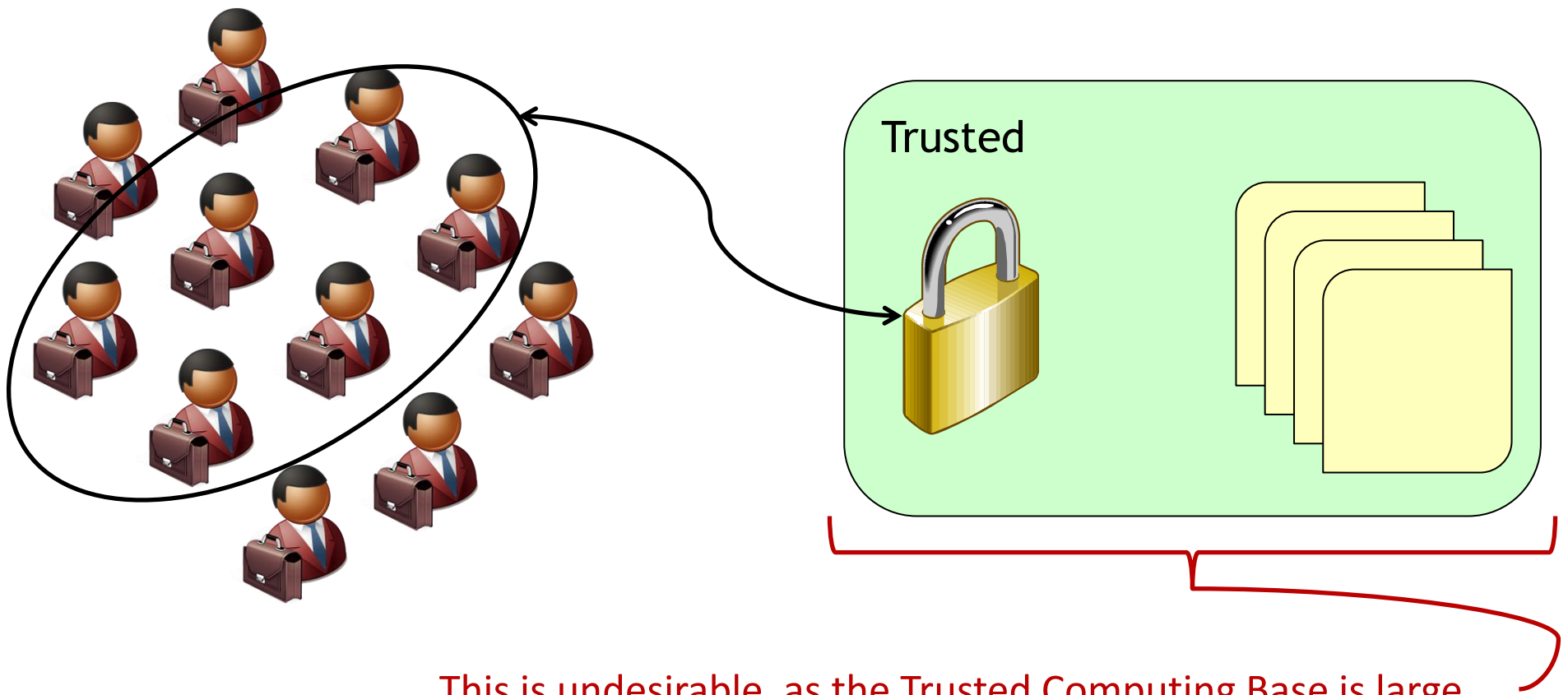
Key Compromise is rare

We've seen several cryptosystems that meet our needs **relative to these assumptions**

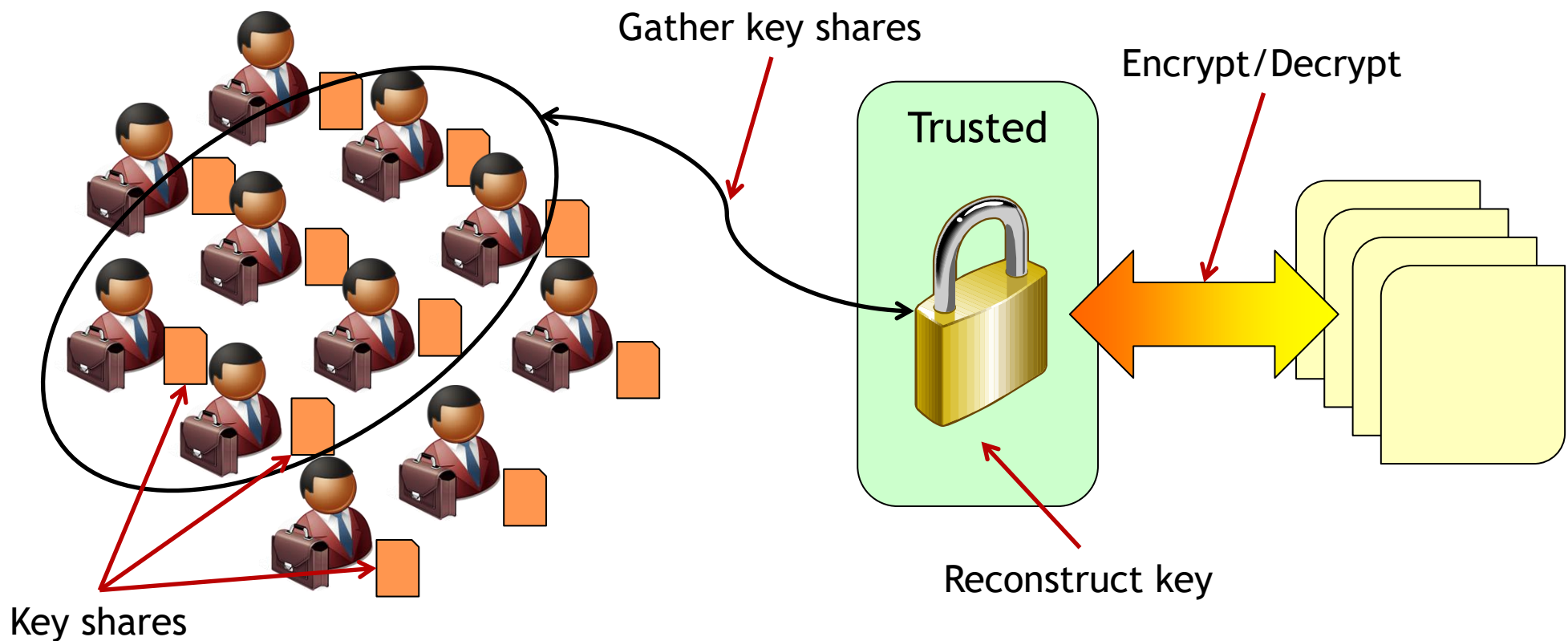
What happens if these assumptions are **violated**?

A motivating example...

Scenario: Eleven scientists are working together on an extremely sensitive joint project. To ensure that results are not tampered with, at least six scientists (i.e., a quorum) must be present in order to read or alter experimental results.



A better solution involves using a shared secret key



Goal: Given n users and a threshold $k < n$, divide a secret D into n pieces D_1, D_2, \dots, D_n such that:

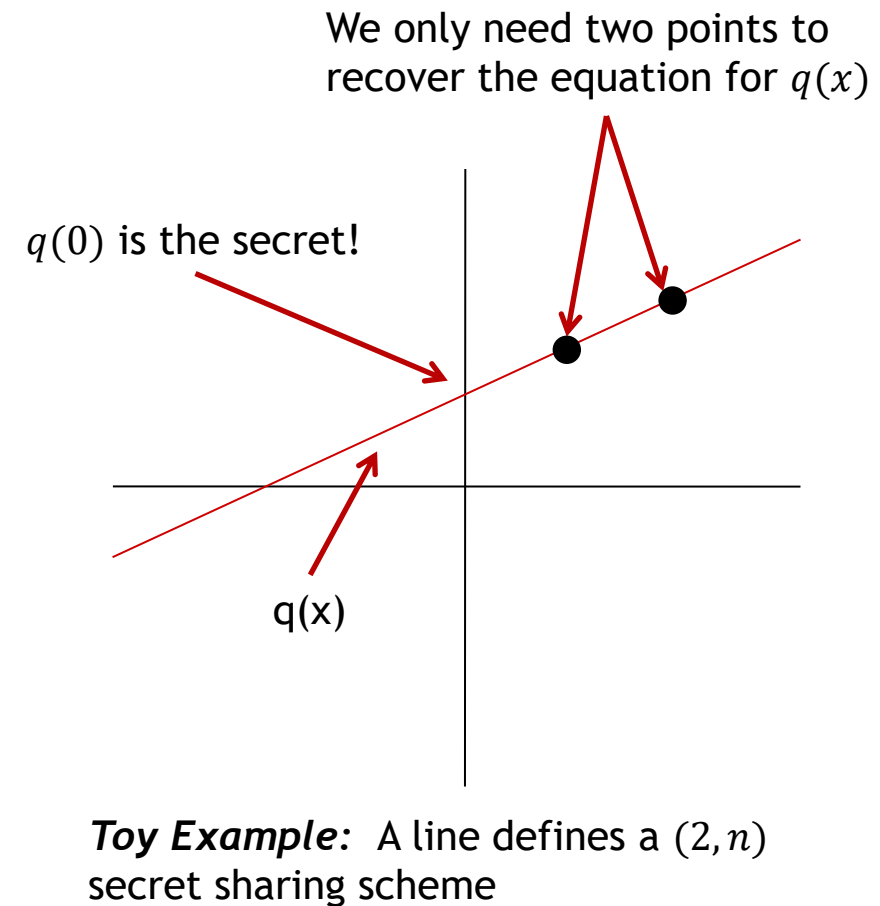
1. Any group of **k or more** users can jointly obtain the secret
2. Any group of $k - 1$ or less users **cannot obtain any information** about the secret (i.e., all possible secrets remain equally likely)

Shamir's (k, n) secret sharing system allows us to accomplish this!

Paper: Adi Shamir, “How to Share a Secret,” Communications of the ACM 22(11): 612-613, November 1979.

Intuitively, this scheme is quite simple

- Choose a random $(k - 1)$ -degree polynomial $q(x)$ whose root is D
- Each key share is a point on $q(x)$
 - $D_1 = q(1)$
 - $D_2 = q(2)$
 - ...
 - $D_n = q(n)$
- Distribute $\{D_1, \dots, D_n\}$ to participants
- $q(x)$ can be recovered by interpolation using any k shares
- $q(0) = D$



Details...

Rather than using real arithmetic, use modular arithmetic

- e.g., The set of integers modulo a large prime p , Z_p
- Equation coefficients and values are $\leq p$
- Note that p must be larger than both D and n

Otherwise, we can't represent D !

Otherwise, we will have collisions in our shares!

How do we randomly generate a $(k - 1)$ -degree polynomial $q(x)$ such that $q(0) = D$?

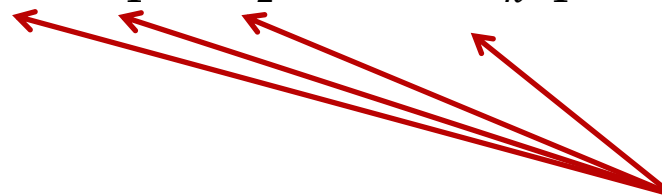
- Note that $q(x) = D + a_1x + a_2x^2 + \dots + a_{k-1}x^{k-1}$ is such a polynomial
- So, we just need to choose a_1, \dots, a_{k-1} at random!
- More precisely, choose a_1, \dots, a_{k-1} from the uniform distribution $[0, p)$

Note: All D_i values are computed modulo p

How is the secret recovered?

To recover D from a set of shares $\{D_1, \dots, D_k\}$, solve the following system of k equations with k unknowns:

- $q(1) = D + a_1 1 + a_2 1^2 + \dots + a_{k-1} 1^{k-1} = D_1$
- $q(2) = D + a_1 2 + a_2 2^2 + \dots + a_{k-1} 2^{k-1} = D_2$
- ...
- $q(k) = D + a_1 k + a_2 k^2 + \dots + a_{k-1} k^{k-1} = D_k$



Unknowns: D, a_1, \dots, a_{k-1}

Note that any k shares can be used, not just $\{D_1, \dots, D_k\}$

There exist $O(n \log^2 n)$ algorithms for polynomial interpolation

- In other words, the complexity of this operation is minimal
- For most (k, n) schemes, even a quadratic algorithm is fine

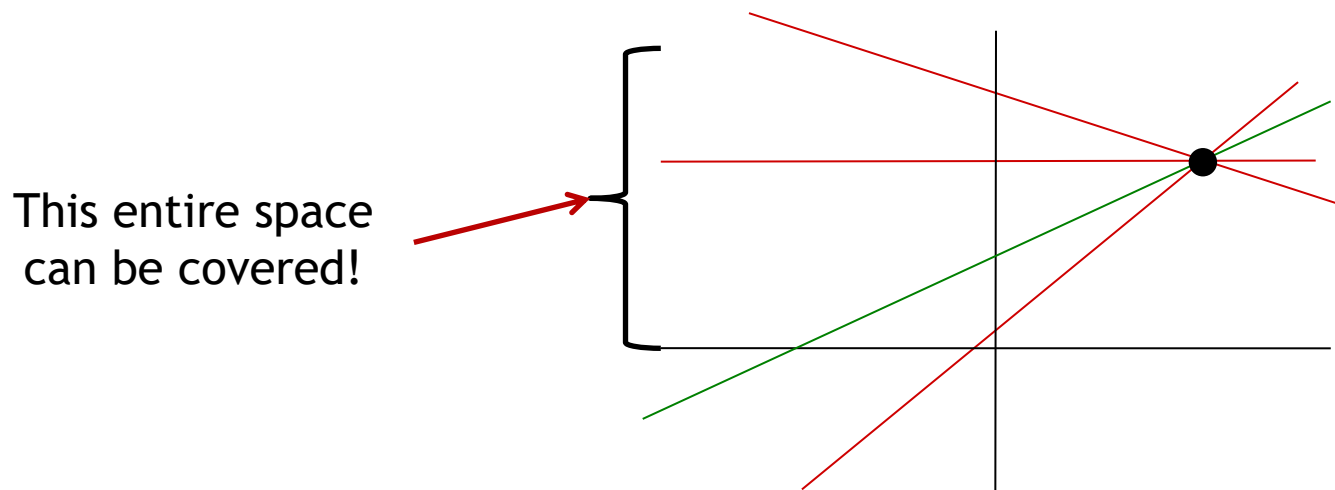
Is Shamir's scheme secure?

Recall: To prove security, we must show that any attacker who has $k - 1$ shares thinks that all possible values of D are equally likely

Assume that the attacker has $k - 1$ shares D_i

- For **each** candidate secret $D_c \in [0, p)$, **exactly one** $k - 1$ degree polynomial $q'(x)$ can be constructed such that $q'(0) = D_c$ and $q'(i) = D_i$ for each known share
- So, all p possible polynomials are equally likely!

Degenerate Example: $(2, n)$ secret sharing



Shamir's scheme has many nice properties

The *size* of each share is less than or equal to the size of the secret D

- Why?

For a fixed k , shares can be dynamically added or deleted without having to recompute other shares from scratch!

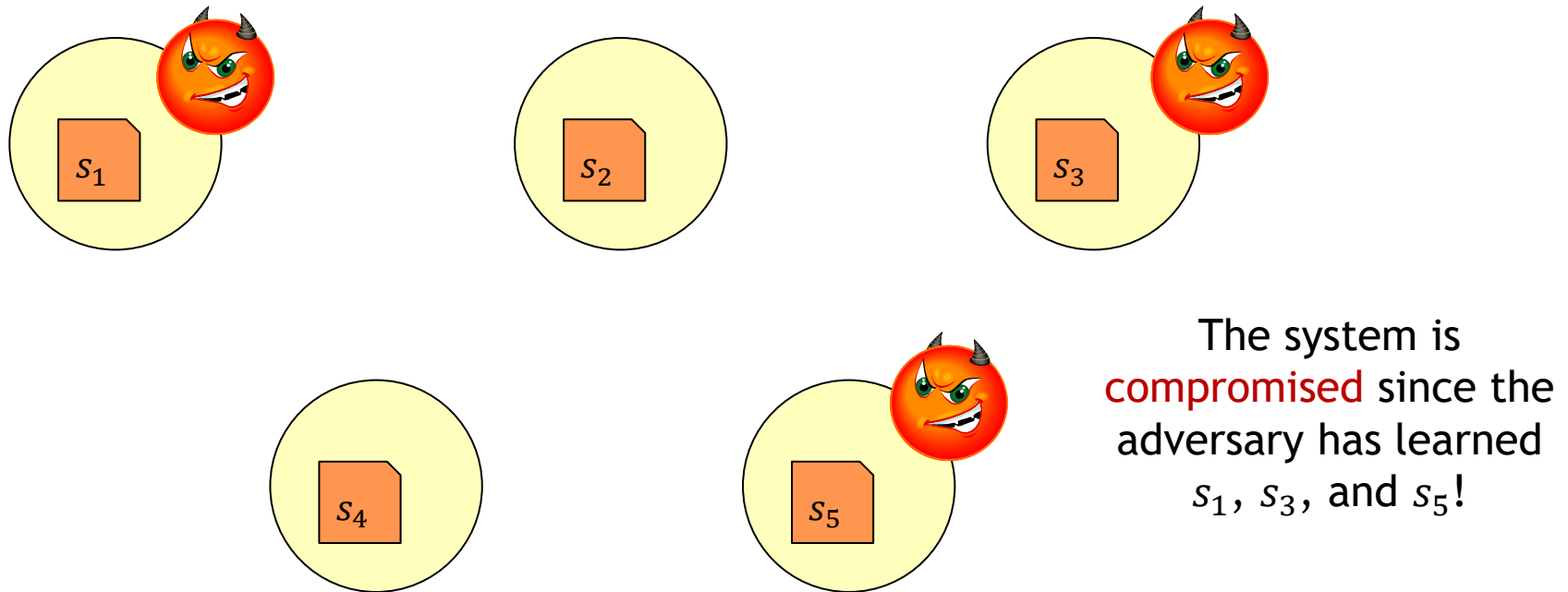
- Adding is easy---just compute new points along $q(x)$
- How do we delete a share?!?

We can develop a hierarchical scheme where the number of shares needed to find D depends upon the importance of principals

- **Example:** $(3, n)$ scheme
- 3 shares to president, 2 shares to each VP, 1 share to each board member
- Who can recover the secret?

This scheme is a building block that allows us to tradeoff security and reliability according to the values of k and n chosen

What happens as machines are compromised over time?



Example: A (3, 5) secret sharing system

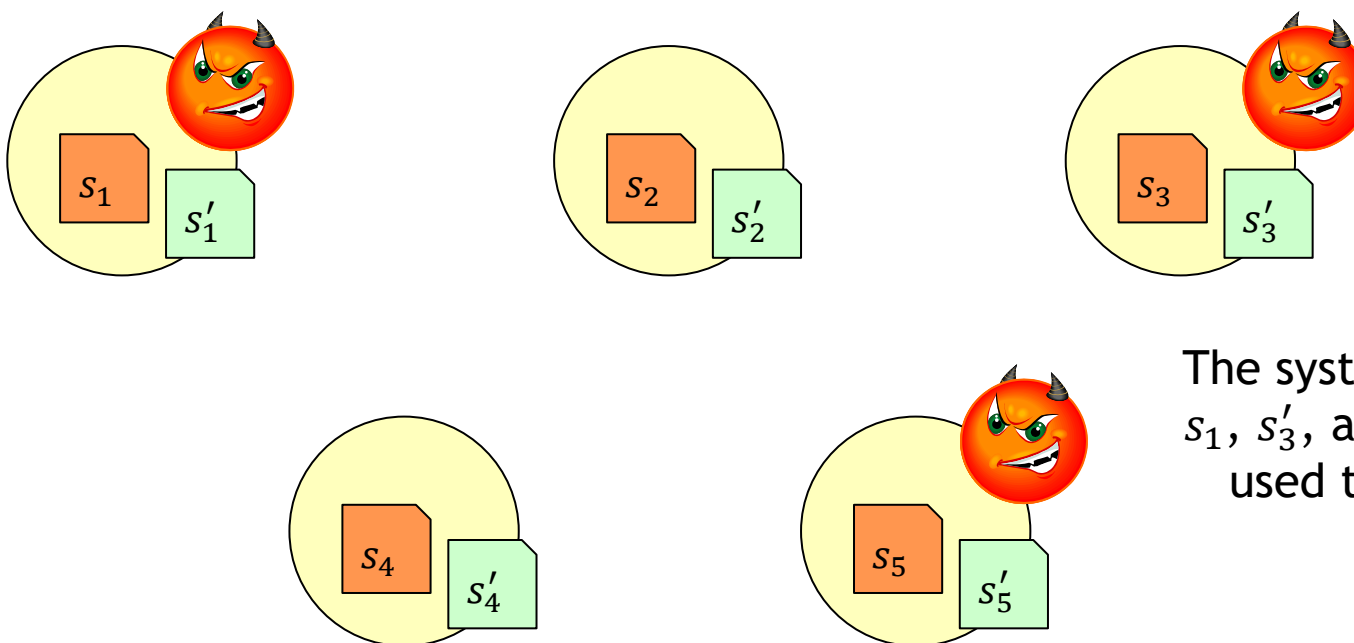
Consider a mobile adversary, capable of compromising nodes

- Over time, more and more nodes compromised
- The adversary learns all secrets stored on each compromised node

If 3 nodes are compromised, the secret is leaked!

- Note: Compromises do **not** need to be simultaneous!

Proactive secret sharing solves this problem!



The system is **safe** since s_1, s'_3 , and s'_5 cannot be used to recover the secret!

Assumptions:

- Time proceeds in **epochs** (hours, days, weeks, etc)
- At the start of each epoch, machines can be rebooted into a safe state

Main idea:

- Update secret shares at the start of each epoch
- New shares should **invalidate** old shares!
- Secret should **not** change (despite new shares)
- Shares from new epoch **cannot** be used in conjunction with old shares

How can we do this?

During the setup phase, a trusted dealer

- Determines $q(x)$ and shares as in Shamir's scheme
- Shares sent out to participants

To evolve shares, each participant P_i

- Creates a random $(k - 1)$ -degree polynomial $\delta_i(x)$ s.t., $\delta_i(0) = 0$
- Send $u_{ij} = \delta_i(j) \pmod{q}$ to each P_j
- After receiving all u_{ji} , set $D'_i = D_i + (u_{1i} + \dots + u_{ni}) \pmod{p}$

Note: After share evolution $q'(x) = q(x) + \delta(x)$ This share is independent of the old share

- $\delta(x) = \delta_1(x) + \dots + \delta_n(x)$
- Since $\delta_i(0) = 0$ for each i , $\delta(0) = 0$
- So $q'(0) = q(0) + \delta(0) = D + 0 = D$

The secret is unchanged, despite new shares!

Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung, "Proactive Secret Sharing or: How to Cope with Perpetual Leakage," CRYPTO 1995.

What are the properties of this PSS scheme?

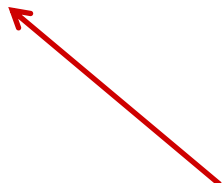
The authors define security in terms of two important properties

Robustness: The new shares computed at the end of the update phase correspond to the secret D (i.e., any subset of k new shares can be used to interpolate the secret D).



Informally, we get this because $\delta(0) = 0$

Secrecy: An adversary that at any time period knows no more than $k - 1$ shares learns nothing about the secret.



Note that old shares become obsolete, since from the adversary's perspective, all possible update polynomials are equally likely. Then, within a single epoch, the proof follows from the secrecy of the Shamir scheme.

Certain distributed computing problems cannot be solved by using shared symmetric keys

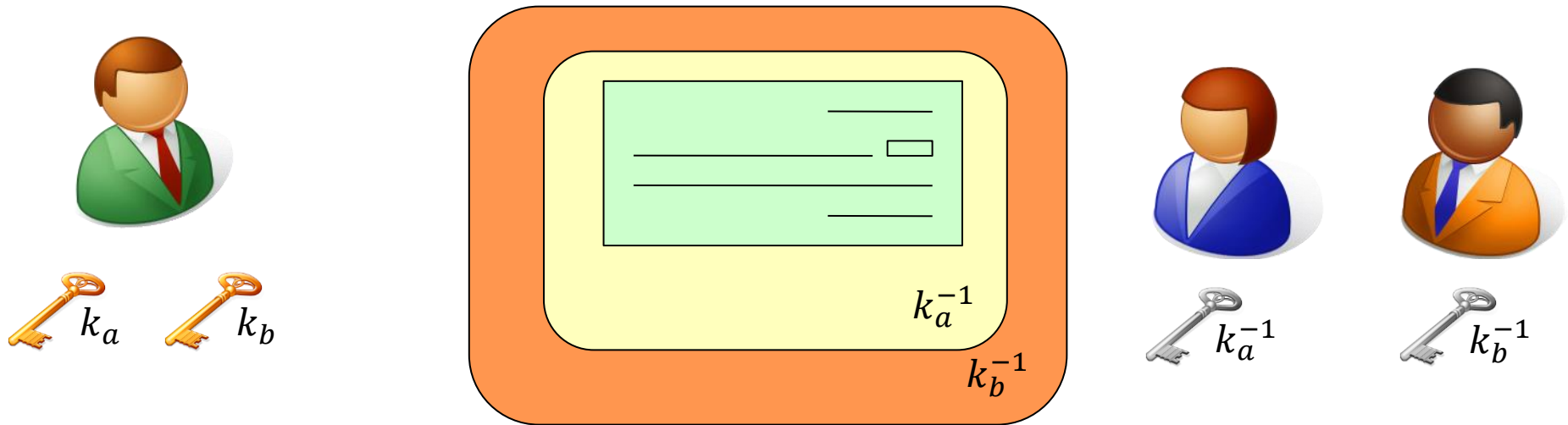
Problem 1: Suppose that a company issues electronic checks, but requires the signatures of at least 3 executives to be considered valid.



Problem 2: A user wants to send an encrypted message to an organization. However, for accountability, the users wants at least 2 members of that organization to view the message together.

*Solving these types of problems requires a **public key** cryptosystem where encryption and/or decryption requires cooperation!*

A Straw-Man Approach



Require **multiple** digital signatures on the message!

Unfortunately, this solution has several pitfalls

- Verifying k signatures is a waste of time
- Message size is proportional to k
- Does not allow for group anonymity
- Doesn't work for decryption case

A better solution would be to define threshold variants of cryptosystems such as RSA

Main idea: Split the RSA decryption exponent d into shares using Shamir's secret sharing approach

How does the protocol work?

- Given a message m , each signer can compute a partial signature of m using their share of the decryption exponent
- These partial signatures can be multiplied together
- *Result:* A signature on m

This protocol has a rigorous proof of security based upon standard cryptographic hardness assumptions

Want more details? See: Victor Shoup, "Practical Threshold Signatures," Proceedings of EuroCrypt 2000.

So... in what applications would someone actually use all of this stuff?

One Application: An online certificate authority

A certificate authority is responsible for managing the bindings between users (or services) and their public keys

This is typically done offline to protect the private signing key of the CA

This entails three main jobs

- Verifying user identity and/or attributes
- Issuing certificates attesting to that identity or those attributes
- Managing lists of revoked or expired certificates

Ideally, this requires each request/response to be digitally signed

Challenge: How can we make a secure online CA?

Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A Secure Distributed On-line Certification Authority. ACM Transactions on Computer Systems 20(4) : 329-368, November 2002.

COCA uses a set of very **weak** system assumptions in order to define a very **resilient** online CA

Assumption 1: Byzantine Failures

- An adversary can compromise servers that implement the service
- Compromised servers can **behave arbitrarily** and/or disclose any local information (including secrets) stored on that server
- An adversary can only compromise **fewer than 1/3** of all the servers in any given epoch

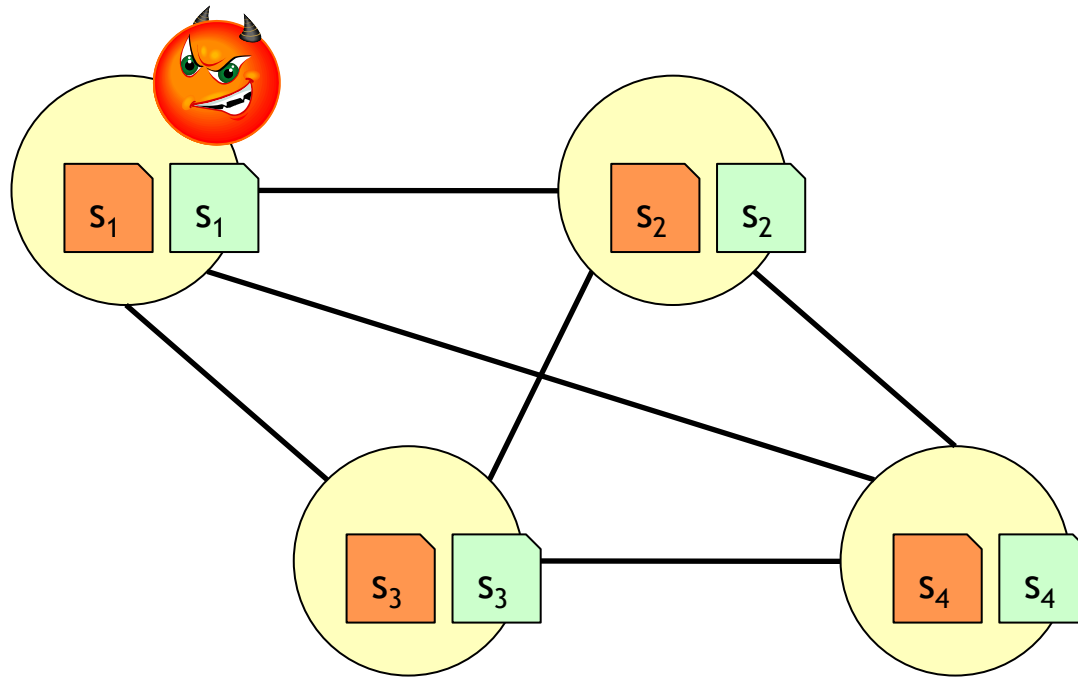
Assumption 2: Active Link Attacks

- An adversary can **insert**, **modify**, **replay**, and **delete** messages

Assumption 3: Asynchronous Systems

- There is **no bound** on message processing time or transmission delay
- DoS attacks can delay messages or slow servers by **arbitrary finite** amounts

So how does COCA work?



COCA uses an interesting combination of cryptographic and distributed systems techniques

- **Replication** with Byzantine quorum
 - i.e., Group “votes” on actions to take
- **Threshold cryptography** to protect private signing key from compromise
- **Proactive secret sharing** to defend against a mobile adversary
- Other DoS resistance techniques

Despite all of these protections, COCA maintains reasonable performance

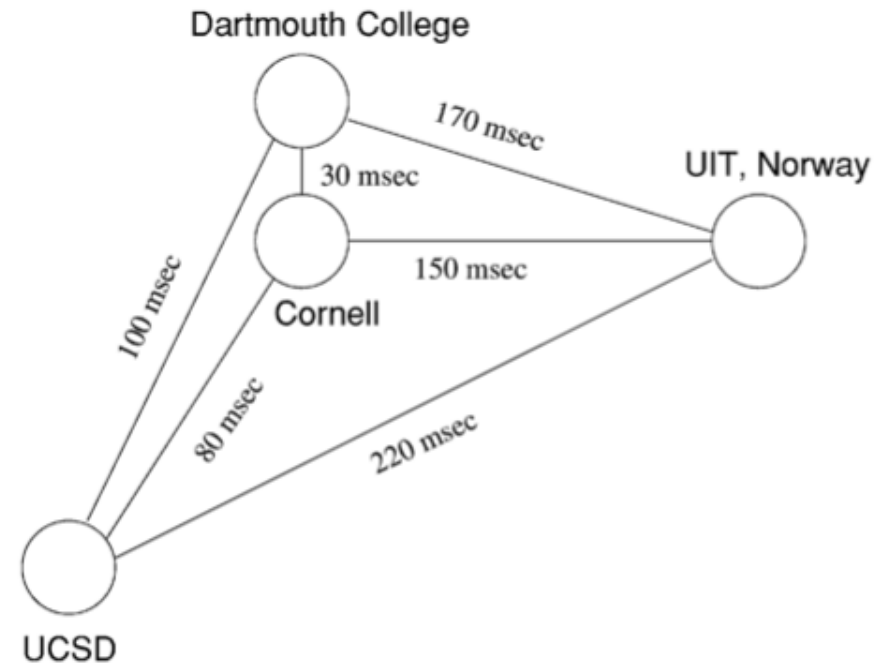
Check certificate validity

COCA Operation	Mean (msec)	Std dev. (msec)
Query	2270	340
Update	3710	440
PSS	5200	620

^aThe averages and sample standard deviations are from 100 repeated executions during a three-day period.

Evolve secret shares

Create new certificate



	Query (%)	Update (%)	PSS (%)
Partial Signature	8.0	8.7	
Message Signing	3.2	2.5	2.6
One-Way Function			7.8
SSL			1.6
Idle	88.0	87.7	87.4
Other	0.8	1.1	0.6

Most time is spent waiting on quorum protocols

Summary of Threshold Crypto Systems

Threshold systems are useful when trying to secure distributed systems

Interesting properties:

- Knowing k shares allows a secret to be reconstructed
- Knowing even $k - 1$ shares provides **no** information

Extensions to this basic model have been proposed to allow

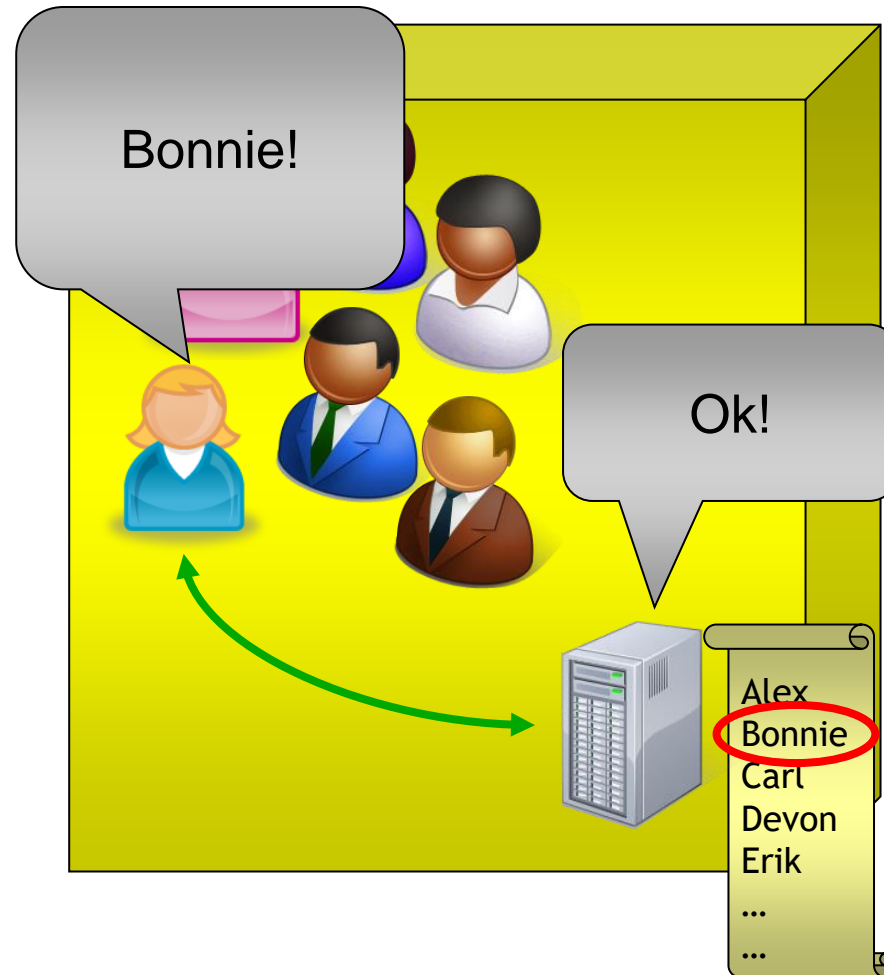
- Share evolution
- Tolerance of an untrusted “dealer”
- Shared signatures
- ...

Next: Authentication and identity

What is authentication? As related to identity?

Informally, **identity** defines who you are...

... while **authentication** is the process through which you prove it!



What's in a name?

```
Username: wcg6  
Password:
```

Standard user IDs within a single domain



IP Address



Email Addresses

Subject: O=University of Pittsburgh,
OU=School of Computing and Information,
OU=Computer Science,
CN=Sherif Khattab

Digital Certificates

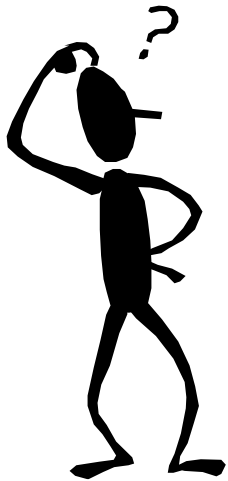


Authentication

Definition: **Authentication** is the process through which an identity is bound to a subject.

Since most computer security policy models are based on user identity, authentication is essentially the root of system security

Typically, people talk about three types of authentication:



Something you **know**



Something you **have**



Something you **are**

Something You Know

Informally, the system asks you a question that only you* know the answer to and verifies the correctness of the response

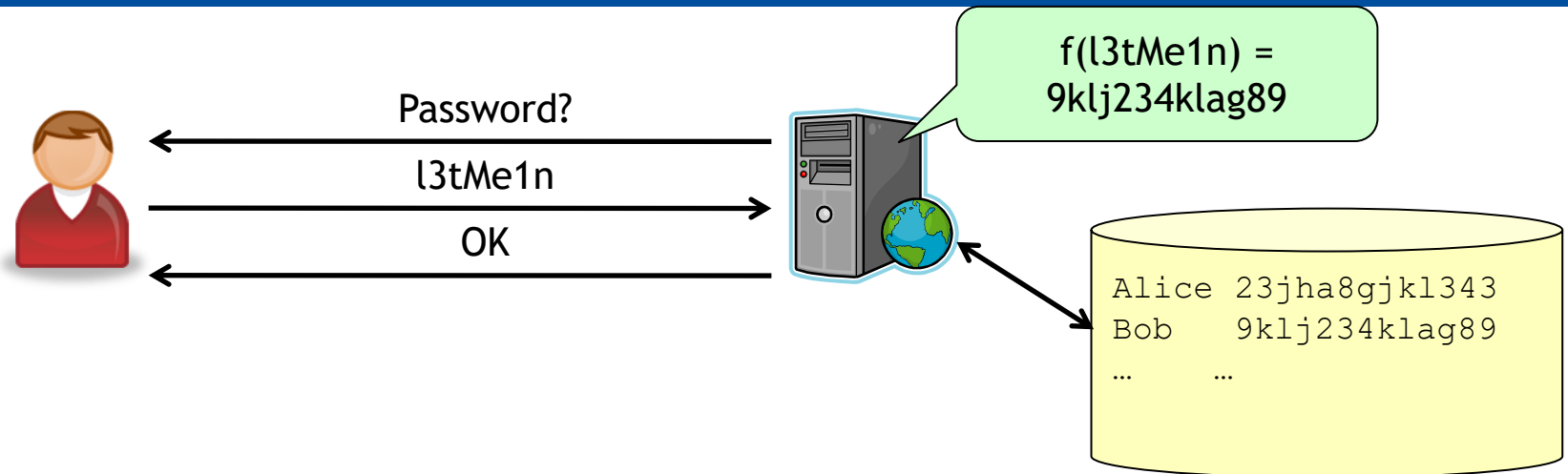
This by far the most commonly used authentication approach

- Passwords
- Passphrases
- PIN numbers
- “Site keys”
- ...

Pro: These systems are easy to implement

Con: Many times, the assumption of secrecy is questionable

How does a password system work?



For each user, the password system stores **complementary information**

- Typically, it stores $f(\langle \text{password} \rangle)$ for some function f

Requirements on the password database

- Users do not need access
- Authentication process should have read access only
- Password update program should have write access

Although ubiquitous, password systems are not perfect

- Attacks possible against **users** and the **system** itself

Password systems assume that only the user knows his or her password

In general, people are helpful and thus subject to **social engineering** attacks



Password systems assume that only the user knows his or her password

In general, people are helpful and thus subject to **social engineering** attacks

If users can choose their own passwords, they typically do an awful job

- For example, people choose passwords based on
 - Username and/or account names
 - Words from the dictionary (possibly with minor modifications)
 - Patterns from the keyboard (e.g., “asdfjkl;”)
 - Family or pet names
 - Passwords from other accounts
 - ...
- These are easy to guess!

As a result, the space of possible passwords is greatly reduced

- $26^8 = 208,827,064,576$ 8-character sequences
- About 29,000 8-character English words

Proactive password checkers try to eliminate these types of threats

Question: What are some issues surrounding proactive password validation?

Unfortunately, allowing the system to choose passwords doesn't buy us much...

In general, people are bad at memorizing random strings

- Studies show that the average person can remember about eight meaningful random things (characters, numbers, etc.)
- This is only one password!

These sort of defeat the purpose of a good password...

What does this mean?

- The same password ends up getting used for multiple systems
- People write down their passwords

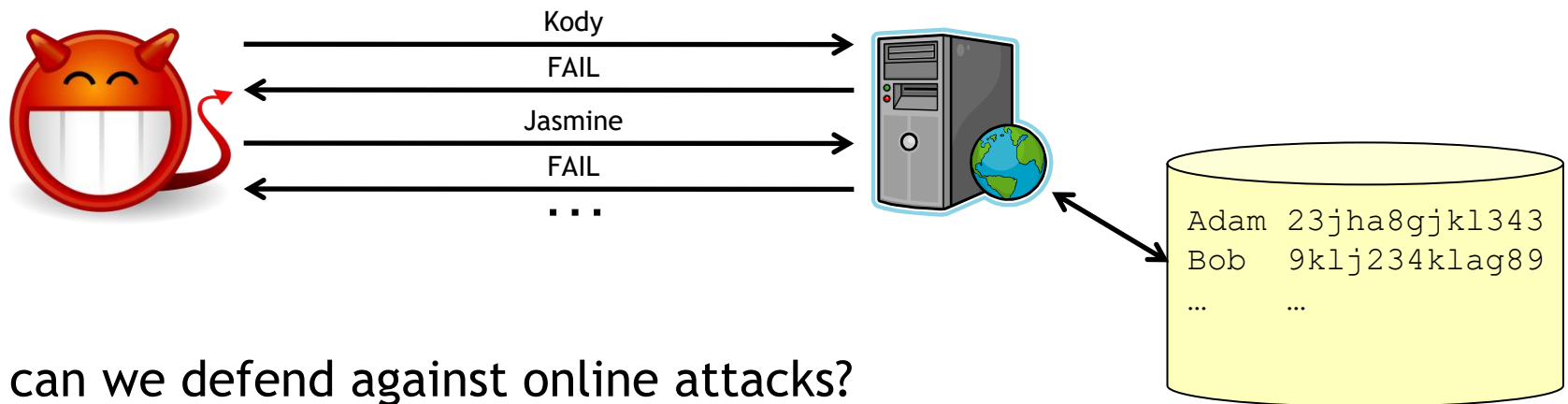


Writing down passwords doesn't **always** need to be a bad thing

- Allows for more secure passwords (too hard to remember)
- Keep password sheet in a secure place
 - Wallet, locked cabinet
- Use an (encrypted) password saving program
 - 1Password, KeePass, LastPass, Apple's keychain
 - Browser's autofill (sometimes)
 - More coming later...

If passwords are reusable (which most of them are) we can attack the system itself

Such an attack can either be conducted **online** or **offline**



How can we defend against online attacks?

- Make it **expensive** to carry out multiple guesses
 - Exponential backoffs in wait time
 - Solve a CAPTCHA before password entry
 - ...
- **Lock** accounts
 - Many ATMs “eat” cards after a set number of failures
 - Online banks often lock accounts after three failures

Although some attackers have amazing patience, online attacks are less fruitful than offline attacks

In an offline attack, the adversary has access to the complementary information associated with some account

The attacker can either guess passwords and check whether $f(<\text{pwd}>)$ matches the complementary information, or she can use a **precomputed mapping** of complementary information to passwords

This is just a hash table lookup!

How can we defend against these lookup attacks?

- You can't stop the attacker, so you need to make her job costly
- e.g., force the computation of a new dictionary for each password

Example: Salting in Unix

- A random 12-bit **salt** is chosen by the system
- The user's **password** is used as a key to the `crypt()` function
 - `crypt()` is effectively DES encryption 25 times (Blowfish in OpenBSD)
 - The salt is used to permute some of the DES tables
 - The result is that encryption depends on the key (**password**) and the salt (**random**)
- The username, salt value, and the “encrypted password” are stored

Question: What are the strong points of salting? The weaknesses?

A password authenticates the user, but how does the user authenticate the authentication process?

Approach 1: Trusted paths in Windows



Ctrl+Alt+Delete traps to the OS and cannot be intercepted by other programs

Approach 2: SiteKeys

These approaches are a first line of defense that allows users to detect tampering with the authentication process

Question: What are some problems with these approaches?

Most password systems are weak because passwords can be used more than one time

Challenge/response systems can address this problem

- System sends user a challenge
- User computes $f(\text{challenge}, \text{secret})$ and returns results
- System checks the correctness of $f(\text{challenge}, \text{secret})$

For such a system to be of any use, knowledge of old (challenge, response) pairs should provide no information about future login attempts

Example: Encrypting a random challenge using a shared key

- If a semantically secure encryption algorithm is used, the interception of one challenge reveals no information about other challenges



This is (roughly) how military identify-friend-or-foe (IFF) systems work

Something You Have

Challenge/response schemes often involve complicated calculations

- Digital signatures
- Repeated hashing
- Time-dependent functions
- ...

Problems:

1. Password reuse leads to attacks
2. People are basically terrible calculators

Solution: Give people a *token* that does these things for them!

- Small device (hook to keychain, keep in wallet, etc.)
- Capable of performing “simple” calculations that people can’t do

You are probably somewhat familiar with token-based authentication

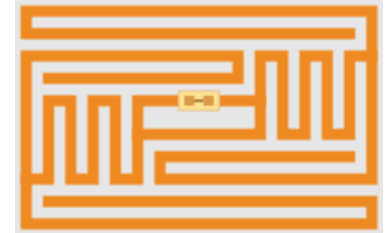
Magnetic strip cards are used at ATMs and for building entry

- Unique ID number stored on magnetic stripe
- ID number used as database key
- Access granted if database says so



Radio Frequency ID (RFID) cards are often used for “no touch” entry

- Many, many uses
 - EZ-Pass
 - Inventory control (libraries, warehouses, stores, etc.)
 - Pet identification
 - PAT bus passes
 - ...
- Two types of tag: passive and active
- Many privacy advocates question the use of RFID



2FA: What happens if a token is lost?

To protect against misuse of lost tokens, many token-based schemes rely on **two-factor authentication**

- Something you have (the token)
- Something you know (a PIN or password)

You use two factor authentication every time you go to the ATM

- First present the card, which defines who you **should** be
- Then enter the PIN, which **verifies** this identity

Starting to appear everywhere

- Google 2-Factor
- Steam Guard
- ...

Crypto cards and smart cards completely change the assumptions surrounding user authentication

These devices can perform sophisticated calculations and are often able to solicit user input during the authentication process

Example: RSA SecurID

How does it work?

- Token contains a 128-bit secret
- Every 30-60 seconds, token generates a new pass code
- To log in:
 - Access service
 - (Possibly) Enter PIN number
 - Enter code from token
- “Duress PINs” help protect against physical threats



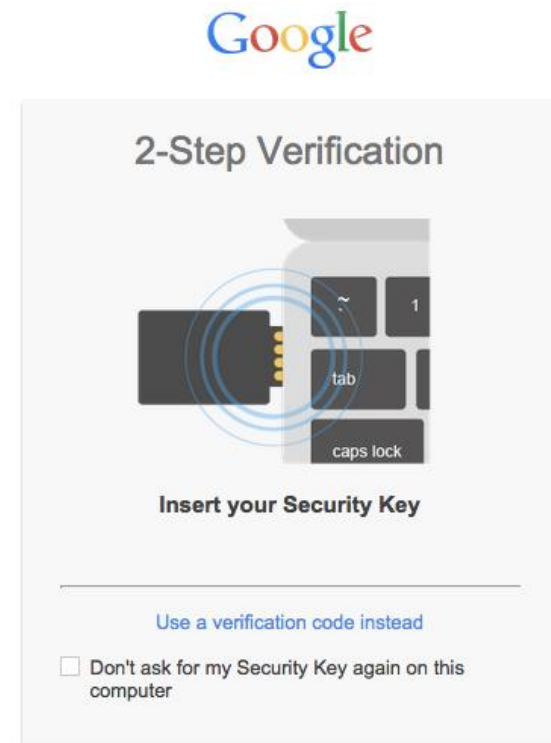
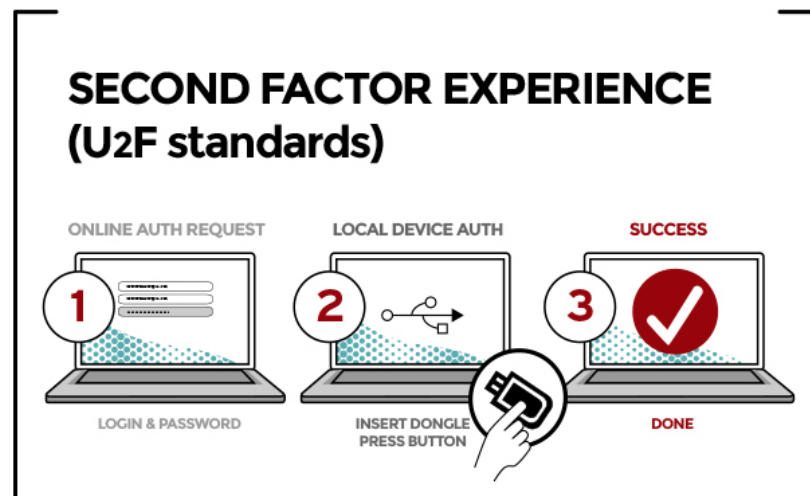
Newer SecurIDs have USB connections. These can store certificates and act as smart cards.

Per-user cost: ~\$40

Newer tokens can do mutual authentication to prevent phishing attacks

FIDO Alliance: Open standard for U2F (universal 2-factor) security keys using public-key crypto

- Different key per site, generated and stored with public key
- In a phishing or MITM attack, no key given without proving ownership of the corresponding private key



Something You Are

Authentication through physical features is not new!



Imprinting



Uniforms



Survival

Furthermore, people often

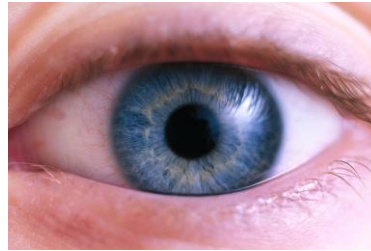
- Choose terrible passwords
- Forget good passwords
- Lose hardware tokens

So why not just do what mother nature does, and identify people by their physical makeup?

Biometric authentication does exactly this!



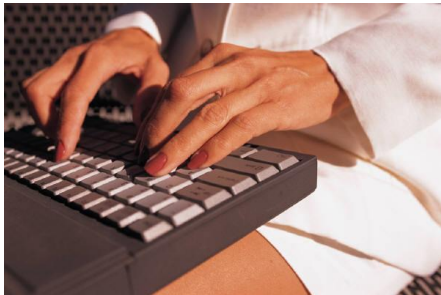
Fingerprints



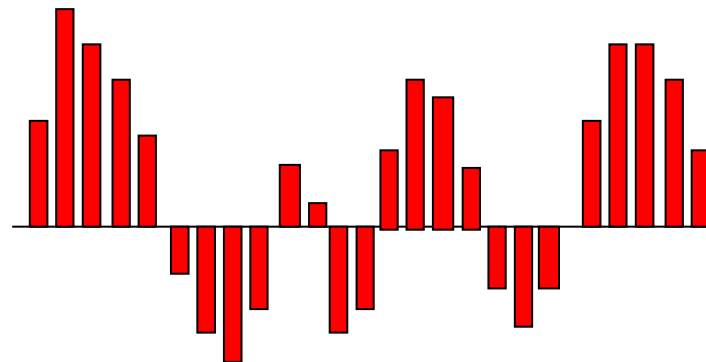
Iris/retina patterns



Hand geometry



Typing patterns

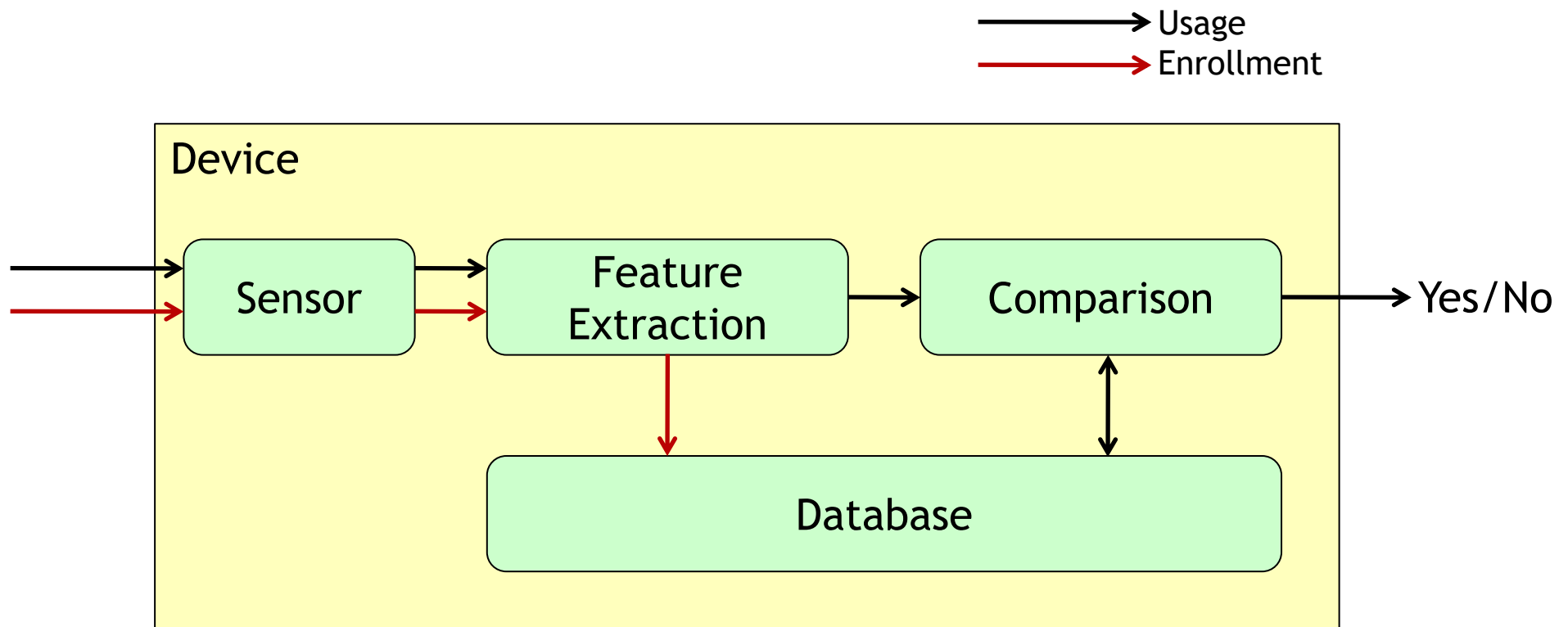


Voice recognition



Facial recognition

Biometrics work by sampling some physical phenomena and comparing this sample with a recorded sample



What happens if we can “steal” a biometric sample?

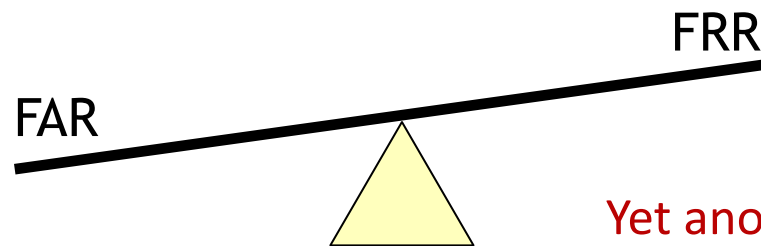
- If we can bypass the sensor, we’re in trouble!

To protect against threats across devices, each database should be based on a unique salt (as in password systems)

When considering biometric authentication, several areas deserve special attention

False acceptance rate (FAR) versus false rejection rate (FRR)

- Each false acceptance is an unauthorized entry (**bad**)
- Each false rejection is a denial of service (**also bad**)



Yet another of Saltzer and Schroeder's principles...

How costly is the approach being considered?

Some biometrics are very intrusive, which leads to problems with psychological acceptability

- e.g., Retinal scans involve shining beams of light into the eye
- Not harmful, but many people do not like this
- **Result:** Retinal scans mostly used in high-assurance environments
- Face unlock

The security provided by biometrics depends on the environment in which they are deployed

In particular, two dimensions play a very large role

- Is the biometric login **monitored** or **unmonitored**?
- Will the biometric provide **primary** authentication, or is it **secondary**?

Example: The “stolen finger” attack

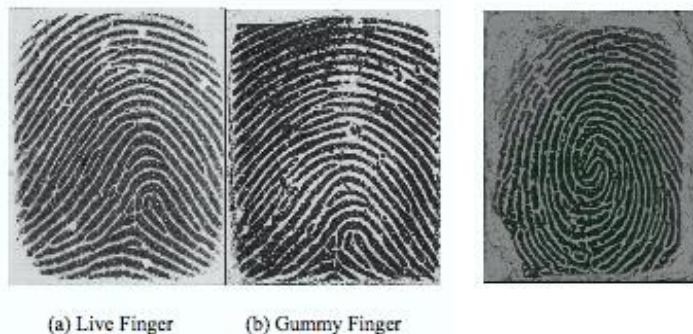
	<i>Monitored</i>	<i>Unmonitored</i>
<i>Primary</i>		
<i>Secondary</i>		

Case study: The nonviolent theft of fingerprints

Matsumoto et al. showed how to fool commercial fingerprint readers using gummy bears

Experiment 1: Use a real finger

- Make a mold of the finger
- Heat gelatin
- Pour gelatin into mold
- Place gelatin on real finger



Experiment 2: Use a latent print

- Take print from glass
- Use adhesive to enhance the print
- Adjust contrast in Photoshop
- Print on to transparency
- Transfer to photo-sensitive PCB
- Etch PCB
- Make mold from etched PCB

Scan of mold from latent print

In both experiments, 11 commercial fingerprint readers accepted the false fingerprints about 80% of the time!

Case study: Finger geometry and Mickey Mouse

Motivation: People buying tickets to Disney World, using them for part of the day, and then giving/selling them to someone else

To defend against this, Disney began recording finger geometry in a coded field of the ticket!

To enter the park:

1. Present ticket
2. Present finger
3. Verify match



System is tuned for a low FRR at the cost of a higher FAR (**Why?**)

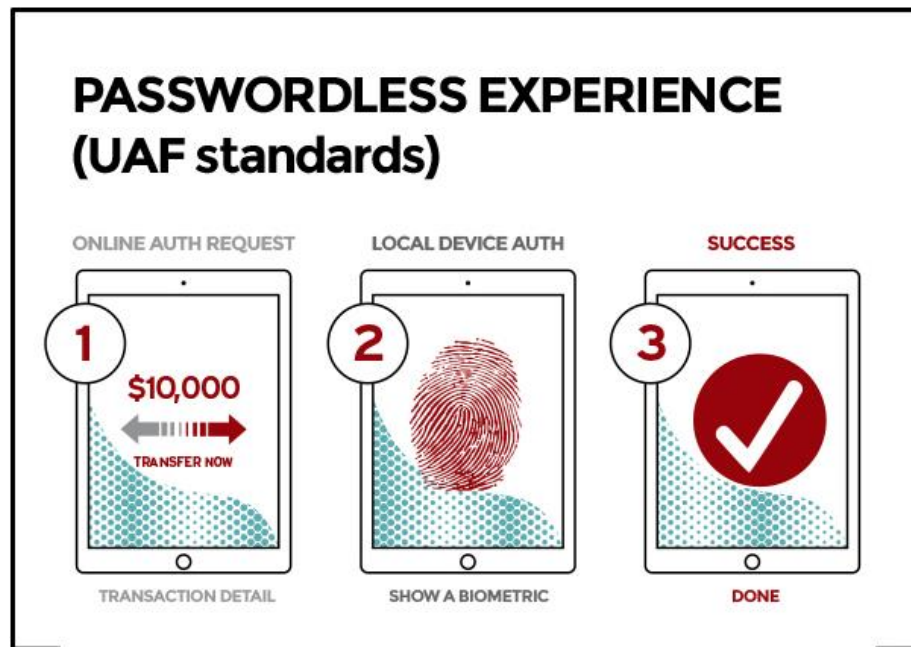
What about privacy?

- Records purged from the system after 30 days
- Clients can opt to present a photo ID instead

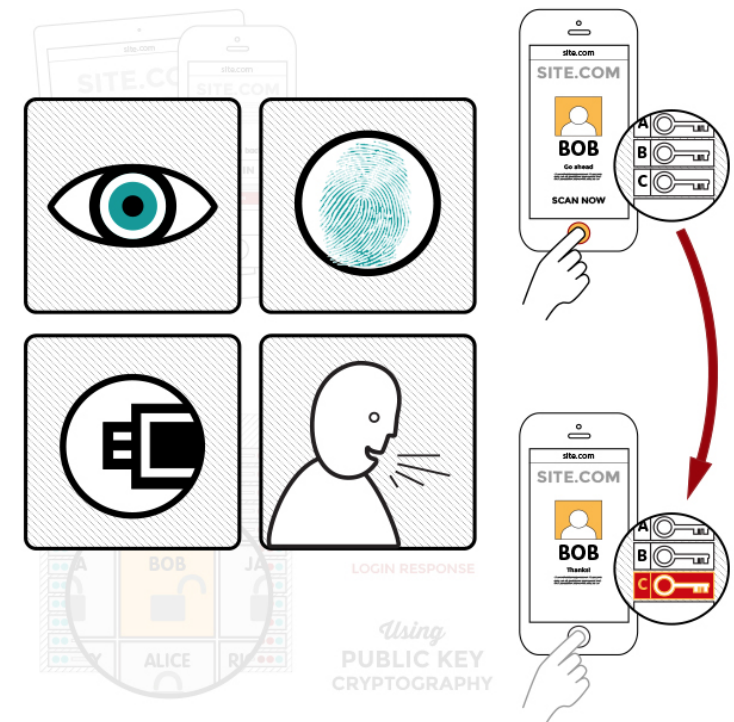
FIDO also allows plugins for biometrics

FIDO Alliance standards include “password-free login”

- When logging in, service sends request to registered device
- Registered device checks biometrics and returns authorization
- Biometric remains on device

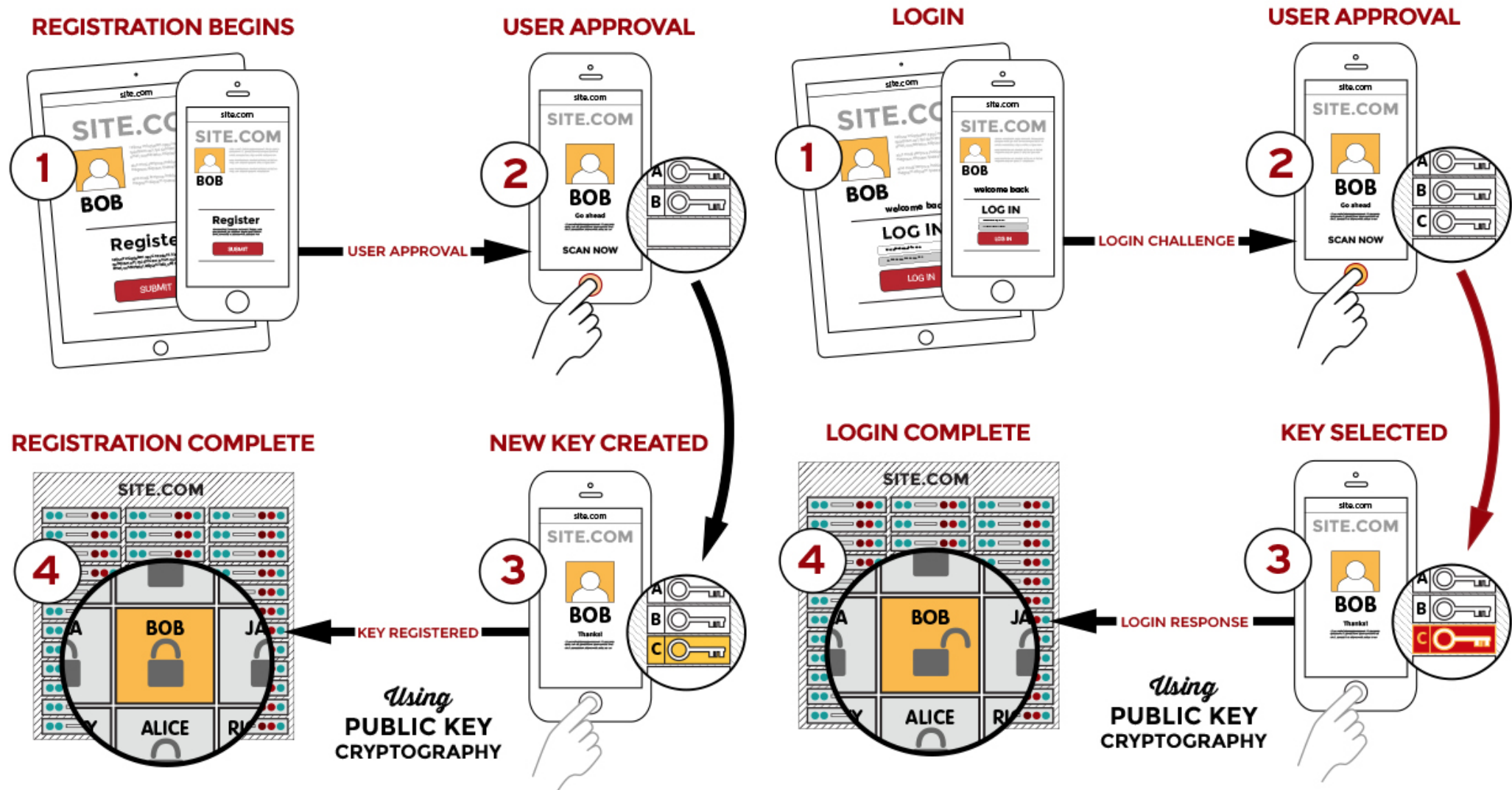


PLUGGABLE LOCAL AUTH



- Kind of two-factor (something you have, something you are)

Other ways to use biometric second factor: something you know + something you are



Summary So Far ...

Passwords are the most commonly used method of authenticating users

A better approach is to combine **something you know** (a password) with **something you have** (an authentication token)

Biometric authentication (i.e., **something you are**) is a good secondary mechanism, but the FRR/FAR tradeoff makes their use as primary authenticators questionable

- Also good as one factor of two-factor authentication

My long term predication for strong authentication: something you have plus (something you know or something you are)

Handshake Protocol

We'll start looking at four types of handshake protocols:

- Login-only protocols
- Mutual authentication protocols
- Integrity/encryption setup protocols
- Mediated authentication protocols

As we'll see, there is a lot of subtlety that goes into designing these types of protocols

Notation Overview

Types of keys:

- K_{AB} : The secret key shared between A and B
- k_A : The public key belonging to A
- k_A^{-1} : The private key belonging to A

Types of cryptographic operations

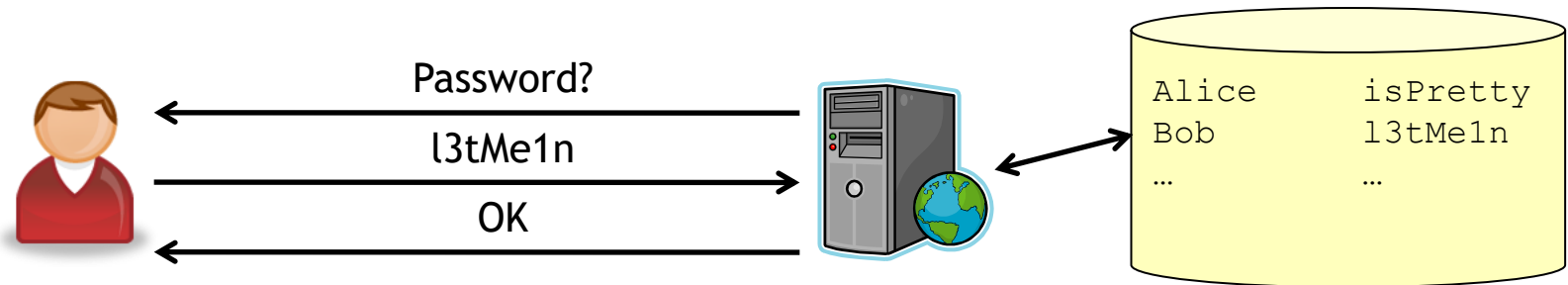
- $\{ M \}_{K_{AB}}$: Message M encrypted with the secret key K_{AB}
- $\{ M \}_{k_A}$: Message M encrypted using A's public key
- $[M]_{k_A^{-1}}$: Message M signed using A's private key

Misc:

- R_A : A random number chosen by A
- $\{ M || R_A \}$: The concatenation of M and R_A

Login-only protocols

Login-only protocols are designed to authenticate the user prior to permitting system access



This protocol operates under the assumptions that:

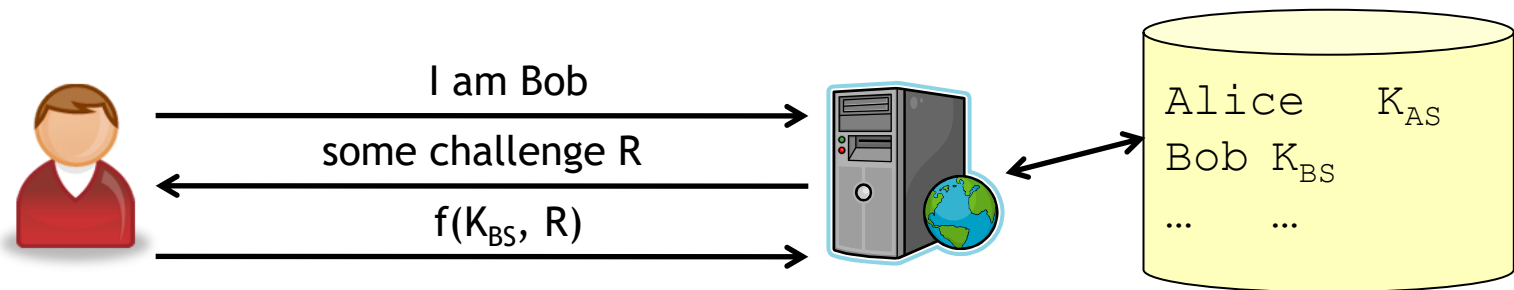
- Only the user knows his password
- The password database is private
- No one is listening on the communication channel

Question: Are these assumptions always valid? Always invalid?

- It depends!

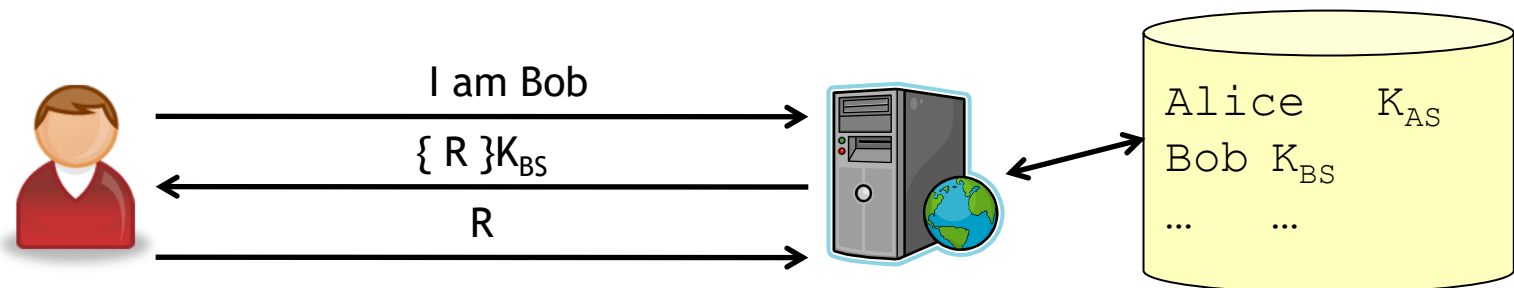
Fortunately, this botched protocol is easy to (partially) fix

It is natural to turn this basic protocol into a cryptographic challenge/response using a shared secret



Interesting notes:

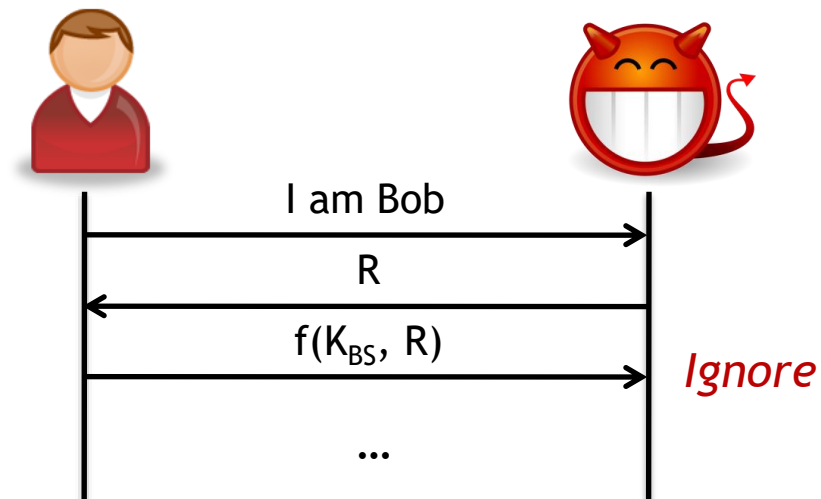
- The use of K_{BS} convinces the server that he is talking to Bob
- f can be either a two-way (encryption) or one-way (hash) function



Note: We **must** use a two-way (encryption) function (**Why?**)

Unfortunately, these protocols are subject to a variety of attacks...

Since Bob does not authenticate the server, anyone can pose as the server!

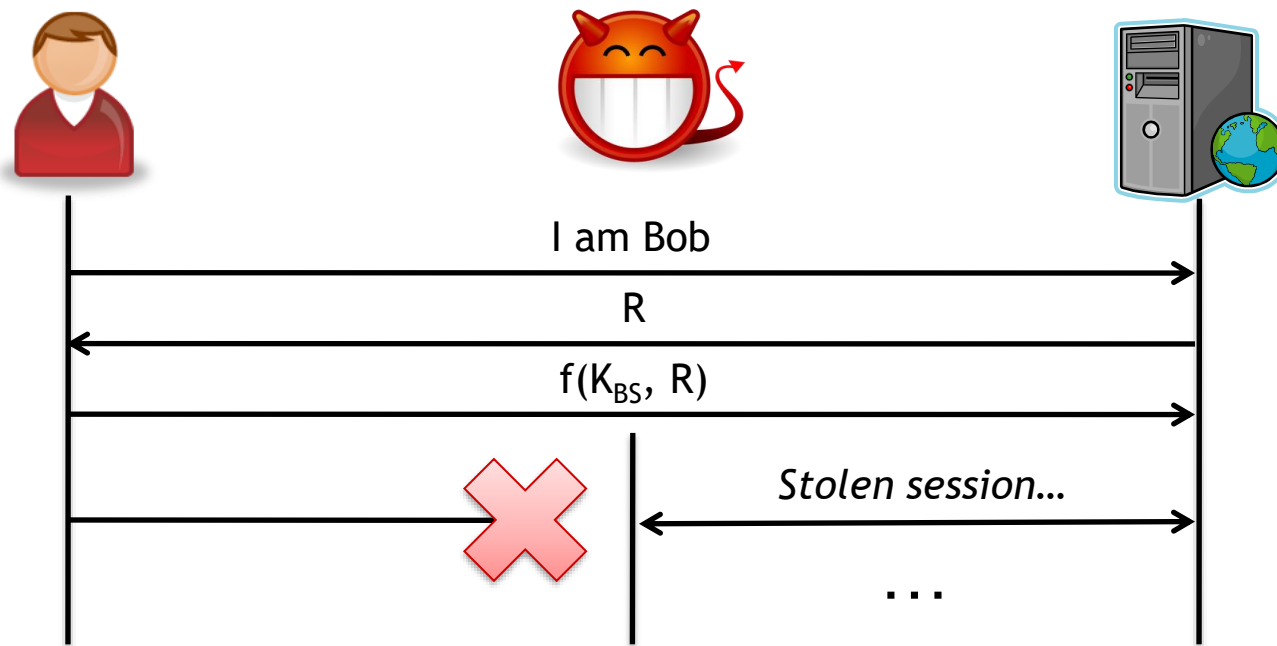


Question: Why might this be a problem?

If the attacker has any knowledge of the server, he can:

- Steal Bob's password(s) as he logs in to, e.g., his email
- Use $f(K_{BS}, R)$ to act as a **man in the middle**
- If K_{BS} is derived from a password, $f(K_{BS}, R)$ can be used to launch an **offline** password guessing attack

An attacker capable of blocking traffic from Bob can hijack Bob's session after login



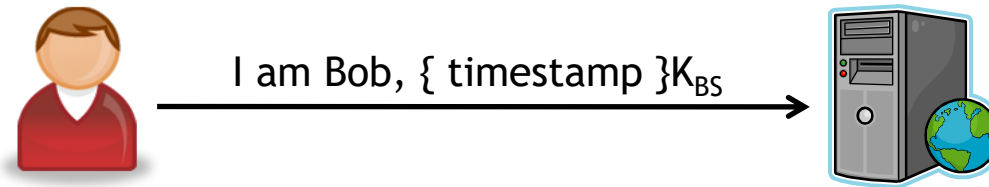
Who could this attacker be?

- Malicious router
- Peer on same network
- ...

The protocol ends after the user is authenticated!

Question: Why is this attack possible?

What if we only have one message in the protocol to work with?



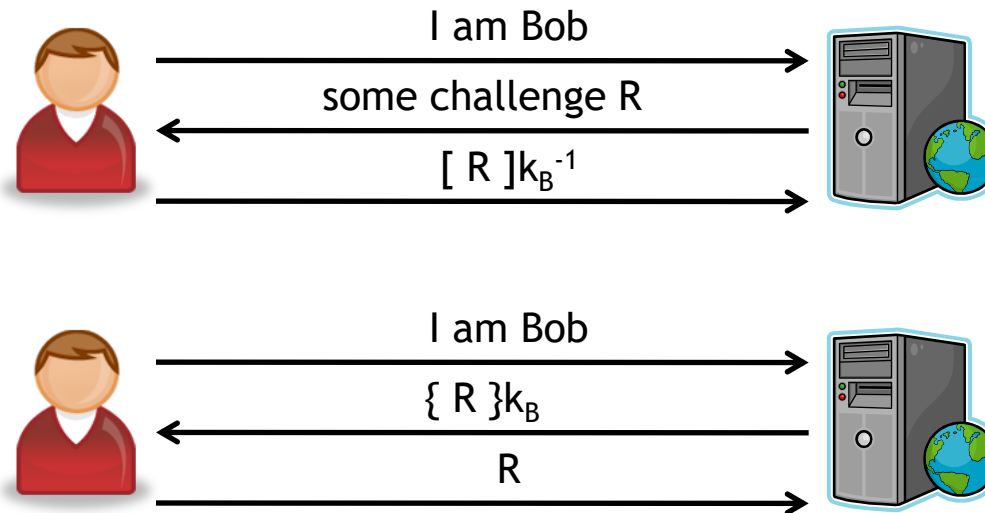
This protocol has several **strong** points:

- Easily replaces protocols that rely on simply sending a cleartext password
- Only requires one message, not three
- The server does not need to maintain volatile state (e.g., previously used R_s)

Sadly, it also has some **weaknesses**:

- Bob and the server need synchronized clocks!
- Attackers snooping on the wire can reuse Bob's encrypted timestamp to log into other servers within an acceptable window of time
- If an attacker can convince the server to roll back its clock, old encrypted timestamps can be reused!

These protocols can also be adapted to use public key cryptography



Why do these protocols work?

- Protocol 1: Only Bob can generate $[R]_{k_B}^{-1}$
- Protocol 2: Only Bob can open $\{R\}_{k_B}$

Interesting note: No more sensitive databases of user passwords or shared secrets!

What is the problem with these protocols?