https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2026

Lecture 9: Authentication II

Instructor: **Nikos Triandopoulos**

February 24, 2026

BROWN

CS1660: Announcements

- Course updates
  - HW 1 is due this Thursday (Feb 26)
  - Project 1 “Flag” is going out today and due March 13
  - Reminder on exam dates
    - **Midterm exam: March 12**
      - Covering primarily: Cryptography, Authentication and Web Security
    - **Final exam: April 28 (reading period)** [to avoid the late May 12 exam date]
      - Covering primarily: OS Security, Network Security and any extra topics

Last class

- Cryptography
  - Introduction to modern cryptography
  - Secure communication & symmetric-key encryption in practice
  - Integrity & reliable communication
  - Public-key encryption & digital signatures
    - Motivation, key management, hybrid encryption, implementation, assumptions
- Authentication
  - User authentication: something you know, are, have
    - Password security and cracking, more on password cracking
  - The Merkle tree

Today

- Cryptography
  - Introduction to modern cryptography
  - Secure communication & symmetric-key encryption in practice
  - Integrity & reliable communication
  - Public-key encryption & digital signatures
    - Motivation, key management, hybrid encryption, implementation, assumptions
- Authentication
  - User authentication: something you know, are, have
    - Password security and cracking, more on password cracking*
  - Data authentication: The Merkle tree, DNS security*

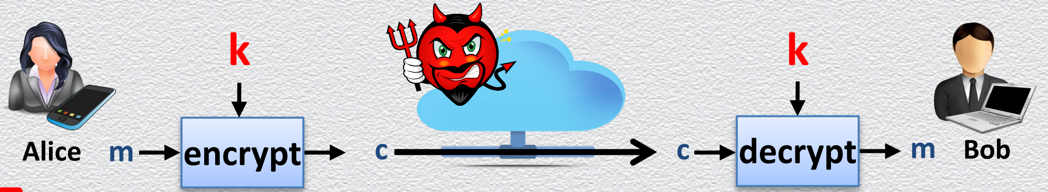
9.0 Public-key crypto in practice

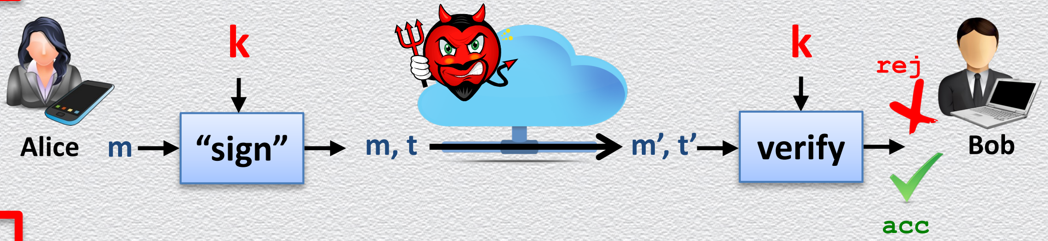
Recall: Principles of modern cryptography

(A) security definitions, **(B) precise assumptions**, (C) formal proofs

For **symmetric-key** message encryption/authentication

- adversary
  - types of attacks
- trusted set-up
  - secret key is distributed securely
  - secret key remains secret
- trust basis
  - underlying primitives are secure
  - PRG, PRF, hashing, ...
    - e.g., block ciphers, AES, etc.



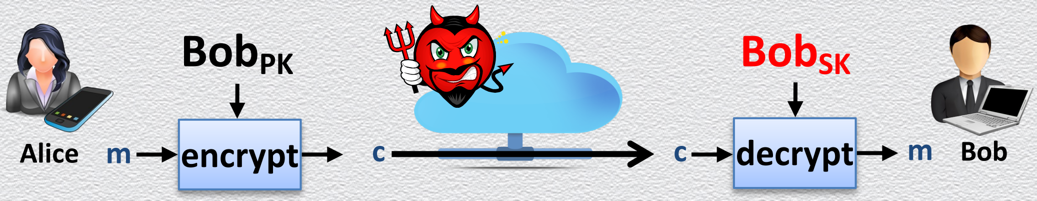


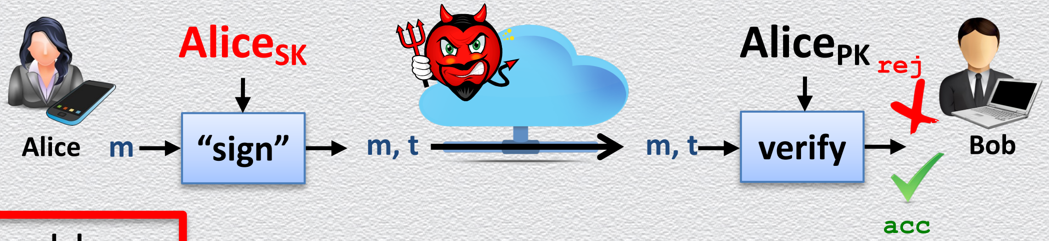
Recall: Principles of modern cryptography

(A) security definitions, **(B) precise assumptions**, (C) formal proofs

For **asymmetric-key** message encryption/authentication

- adversary
    - types of attacks
- trusted set-up
    - PKI is needed
    - secret keys remain secret
- trust basis
    - underlying primitives are secure
    - algebraic computationally-hard problems
        - e.g., discrete log, factoring, etc.





Recall: General comparison

Symmetric crypto

- key management
  - less scalable & riskier
- assumptions
  - secret & authentic communication
  - secure storage
- primitives
  - generic assumptions
  - more efficient in practice

Asymmetric crypto

- key management
  - more scalable & simpler
- assumptions
  - authenticity (PKI)
  - secure storage
- primitives
  - math assumptions
  - less efficient in practice (2-3 o.o.m.)

Recall: Multiplicative group Z_n^*

Subsets of Z_n with elements that are relative prime to n (i.e., have an inverse)

- **CASE 1**: if n is a prime number, then all non-zero elements in Z_n have an inverse
- **CASE 2**: if n is not prime, then not all integers in Z_n have an inverse

Order of group is simply it's size: $\phi(n) = |Z^*_n|$

- if **n = p is prime**, then $\phi(n) = p-1$
- if **n = p q** (product of primes), then $\phi(n) = (p-1)(q-1)$

Structure

- There exist generators in in Z^*_n so that their powers produce all Z^*_n
- For each element x in Z^*_n, we have $x^{\phi(n)} \bmod n = 1$

9.0.1 The DH key-agreement protocol

The Discrete-Log assumptions

Discrete-log setting

- cyclic $G = (Z_p^*, \cdot)$ of order $p - 1$ generated by g, prime p of length t ($|p|=t$)

Discrete Log assumption

- given G, g, p and x in Z_p^*, computing the discrete log k of x (mod p) is infeasible
- holds for groups of specific structure, best solution involves exhaustive search ($2^{-t/2}$)

Computational Diffie-Hellman assumption – e.g., ElGamal encryption

- given G, g, p, g^a and g^b, computing g^{ab} (mod p) is infeasible

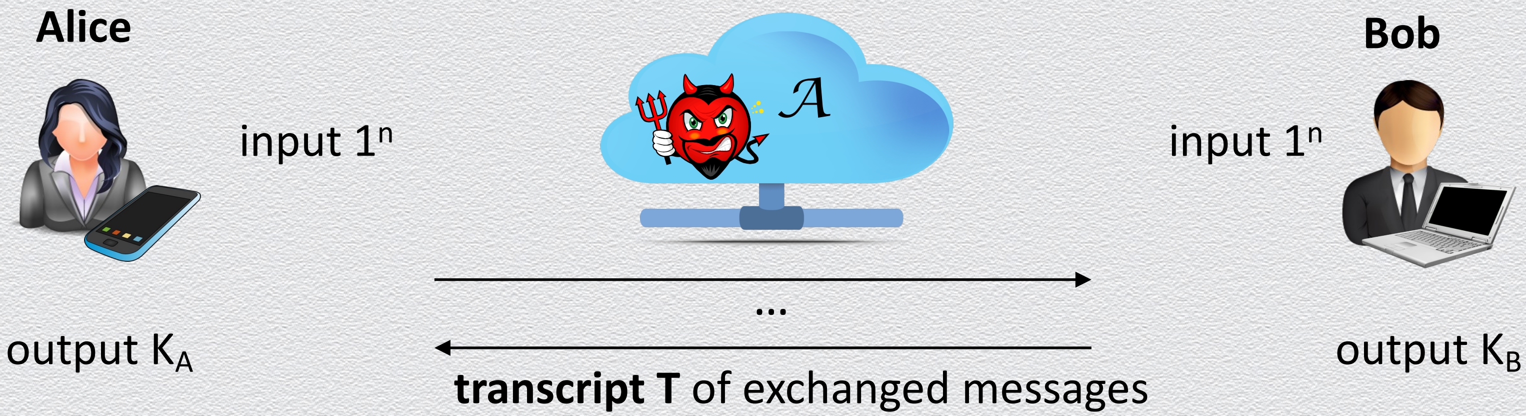
Decisional Diffie-Hellman assumption – e.g., DH key agreement

- given G, g, p, g^a and g^b, g^{ab} is computationally indistinguishable from uniform

Application: Key-agreement (KA scheme)

Alice and Bob want to securely establish a **shared key** for secure chatting over an **insecure** line

- instead of meeting in person in a secret place, they want to use the insecure line…
- KA scheme: they run a key-agreement protocol Π to contribute to a shared key K
- correctness: $K_A = K_B$
- security: no PPT adversary $\mathcal{A}$, given T, can distinguish K from a trully random one



Alice

input 1^n

output K_A

$\mathcal{A}$

…

transcript T of exchanged messages

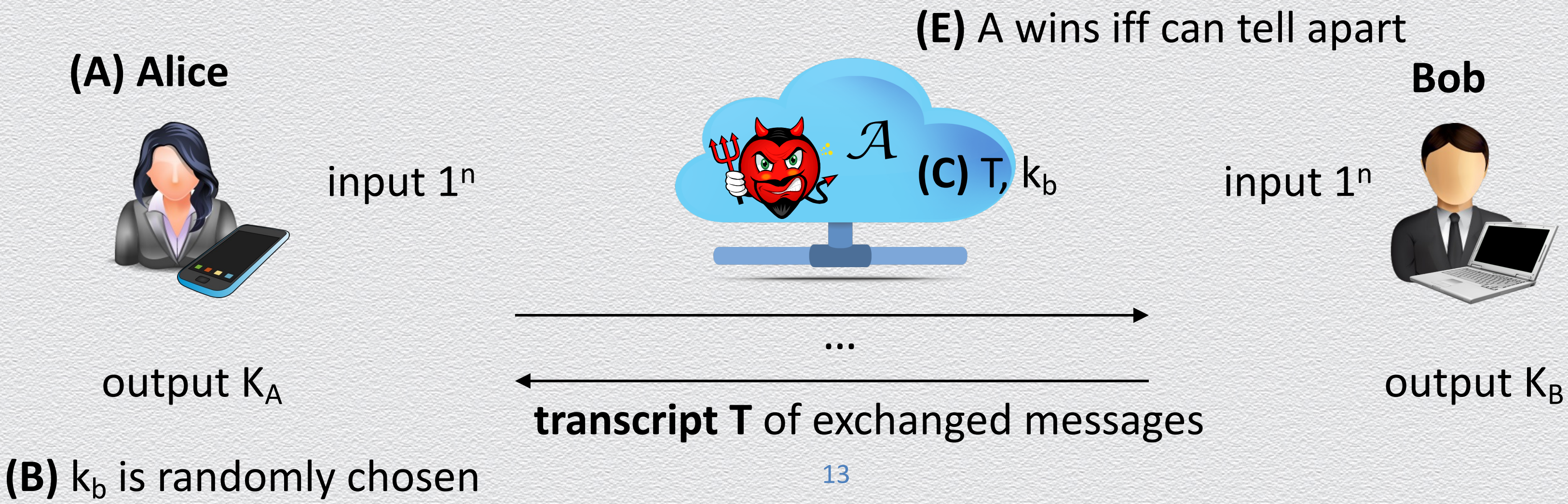
Bob

input 1^n

output K_B

Security definition

- scheme $\Pi(1^n)$ runs to generate $K = K_A = K_B$ and transcript T
- adversary $\mathcal{A}$ is given T and $k_b = K$ with prob. 50% or k_b = random with prob. 50%
- **Π is secure if no PPT A can win non-negligibly often** (better then guessing)



(E) A wins iff can tell apart

(A) Alice

Bob

$\mathcal{A}$

input 1^n

(C) T, k_b

input 1^n

...

output K_A

output K_B

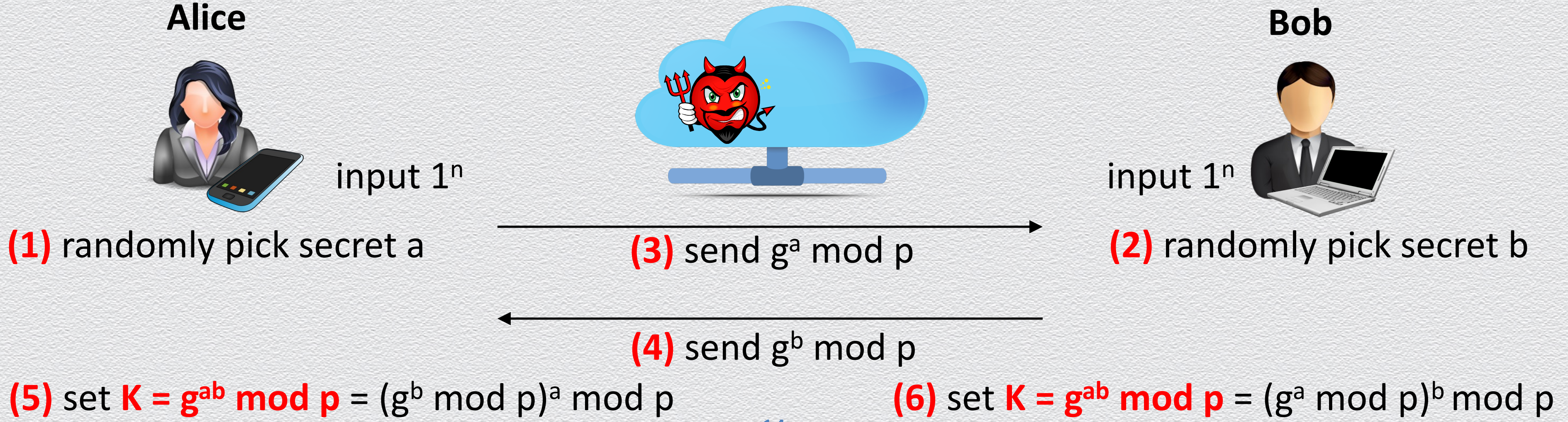
transcript T of exchanged messages

(B) k_b is randomly chosen

The Diffie-Hellman KA scheme

Alice and Bob want to securely establish a **shared key** for secure chatting over an **insecure** line

- DH KA scheme Π
  - discrete log setting: p, g public, where <g> = Z^*_p and p prime



Alice

input 1^n

(1) randomly pick secret a

Bob

input 1^n

(2) randomly pick secret b

(3) send g^a mod p

(4) send g^b mod p

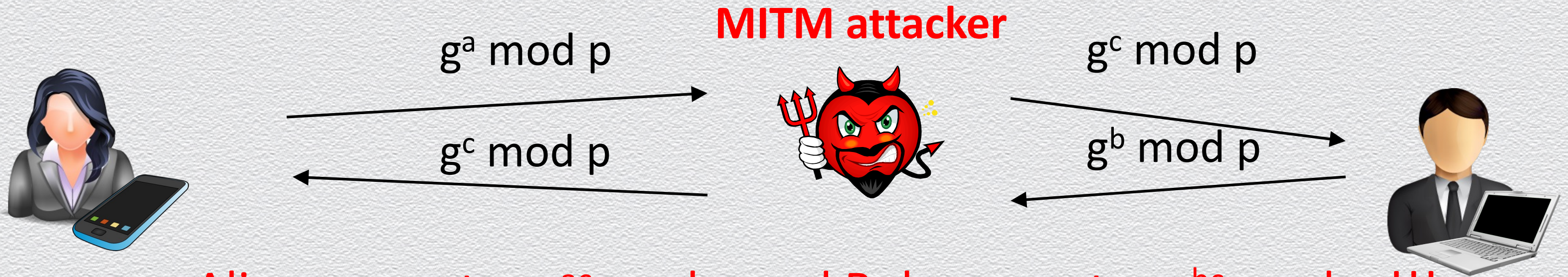
(5) set **K = g^{ab} mod p** = (g^b mod p)^a mod p

(6) set **K = g^{ab} mod p** = (g^a mod p)^b mod p

Security

- discrete log assumption is necessary but not sufficient
- decisional DH assumption
    - given g, g^a and g^b, g^{ab} is computationally indistinguishable from uniform

Authenticated Diffie-Hellman



MITM attacker

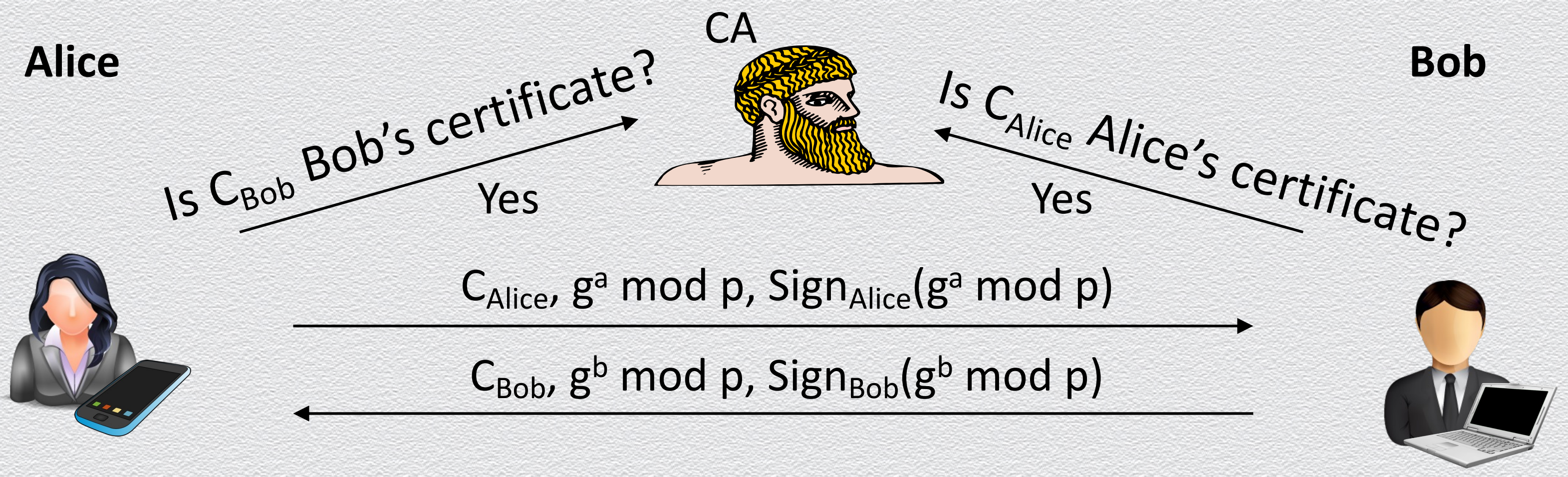
g^a mod p

g^c mod p

g^c mod p

g^b mod p

Alice computes g^{ac} mod p and Bob computes g^{bc} mod p !!!



CA

Alice

Bob

Is C_{Bob} Bob’s certificate?

Yes

Is C_{Alice} Alice’s certificate?

Yes

C_{Alice}, g^a mod p, $Sign_{Alice}(g^a \bmod p)$

C_{Bob}, g^b mod p, $Sign_{Bob}(g^b \bmod p)$

9.0.2 The RSA algorithm

The RSA algorithm (for encryption)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\boldsymbol{\phi}(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$

Keys

- public key is $\mathbf{K}_{\mathbf{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K}_{\mathbf{SK}} = \mathbf{d}$

Encryption

- $\mathbf{C} = \mathbf{M}^{\mathbf{e}} \bmod \mathbf{n}$ for plaintext **M** in $\mathbf{Z}_{\mathbf{n}}$

Decryption

- $\mathbf{M} = \mathbf{C}^{\mathbf{d}} \bmod \mathbf{n}$

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Encryption

- $\mathbf{C} = 19^{5} \bmod 119 = 66$ for $\mathbf{M} = 19$ in $\mathbf{Z}_{\mathbf{119}}$

Decryption

- $\mathbf{M} = 66^{77} \bmod 119 = 19$

Another complete example

- Setup
  - **p** = 5, **q** = 11, **n** = 5 · 11 = 55
  - **ϕ**(**n**) = 4 · 10 = 40
  - **e** = 3, **d** = 27 (3·27 = 81 = 2·40 + 1)
- Encryption
  - **C** = **M**^{3} mod 55 for **M** in **Z**$_{55}$
- Decryption
  - **M** = **C**^{27} mod 55

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Correctness of RSA

Given

Setup

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\boldsymbol{\phi}(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$ **(1)**

Encryption

- $\mathbf{C} = \mathbf{M}^e$ mod **n** for plaintext **M** in $\mathbf{Z}_\mathbf{n}$

Decryption

- $\mathbf{M} = \mathbf{C}^d$ mod **n**

Fermat's Little Theorem **(2)**

- for prime p, non-zero x: $x^{p-1} \bmod p = 1$

Analysis

Need to show

- $M^{ed} = M \bmod p \cdot q$

Use **(1)** and apply **(2)** for prime p

- $M^{ed} = M^{ed-1} M = (M^{p-1})^{h(q-1)} M$
- $\mathbf{M}^{ed} = 1^{h(q-1)} M \bmod p = \mathbf{M \bmod p}$

Similarly (w.r.t. prime q)

- $\mathbf{M^{ed} = M \bmod q}$

Thus, since p, q are co-primes

- $\mathbf{M^{ed} = M \bmod p \cdot q}$

A useful symmetry

[1] RSA setting

- modulo **n** = **p** · **q**, p & q are **primes**, public & private keys (e,d): **d · e = 1 mod (p-1)(q-1)**

[2] RSA operations involve **exponentiations**, thus they are **interchangeable**

- **C** = M^e mod **n** (encryption of plaintext **M in Z_n**)
- **M** = C^d mod **n** (decryption of ciphertext **C in Z_n**)

Indeed, their order of execution does not matter: $(M^e)^d = (M^d)^e \bmod n$

[3] RSA operations involve exponents that **"cancel out",** thus they are **complementary**

- $x^{(p-1)(q-1)} \bmod n = 1$ (Euler's Theorem)

Indeed, they invert each other: $(M^e)^d = (M^d)^e = M^{ed} = M^{k(p-1)(q-1)+1} \bmod n$

$= (M^{(p-1)(q-1)})^k \cdot M = 1^k \cdot M = M \bmod n$

Signing with RSA

RSA functions are complementary & interchangeable w.r.t. order of execution

- **core property**: $\mathbf{M^{ed} = M \bmod p \cdot q}$ for any message $\mathbf{M}$ **in** $\mathbf{Z_n}$

RSA cryptosystem lends itself to a **signature scheme**

- ‘reverse’ use of keys is possible : $\mathbf{(M^d)^e = M} \bmod \mathbf{p \cdot q}$
- signing algorithm **Sign(M,d,n)**: $\boldsymbol{\sigma} = \mathbf{M^d} \bmod \mathbf{n}$ for message $\mathbf{M}$ in $\mathbf{Z_n}$
- verifying algorithm **Vrfy(σ,M,e,n)**: return $\mathbf{M} == \boldsymbol{\sigma}^{\mathbf{e}} \bmod \mathbf{n}$

The RSA algorithm (for signing)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\boldsymbol{\phi}(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$

Keys (same as in encryption)

- public key is $\mathbf{K}_{\mathbf{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K}_{\mathbf{SK}} = \mathbf{d}$

Sign

- $\boldsymbol{\sigma} = \mathbf{M}^{\mathbf{d}} \bmod \mathbf{n}$ for message **M** in $\mathbf{Z}_{\mathbf{n}}$

Verify

- Check if $\mathbf{M} = \boldsymbol{\sigma}^{\mathbf{e}} \bmod \mathbf{n}$

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Signing

- $\boldsymbol{\sigma} = 66^{77} \bmod 119 = 19$ for $\mathbf{M} = 66$ in $\mathbf{Z}_{\mathbf{119}}$

Verification

- Check if $\mathbf{M} = 19^{5} \bmod 119 = 66$

Digital signatures & hashing

Very often digital signatures are used with hash functions

- the hash of a message is signed, instead of the message itself

Signing message M

- let h be a cryptographic hash function, assume RSA setting (n, d, e)
- compute signature σ on message M as: $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

- use public key (e, n) to compute (candidate) hash value $H = \sigma^e \bmod n$
- if $H = h(M)$ output ACCEPT, else output REJECT

Security of RSA

Based on difficulty of **factoring** large numbers (into large primes), i.e., n = p · q into p, q

- note that for RSA to be secure, both p and q must be large primes
- widely believed to hold true
  - since 1978, subject of extensive cryptanalysis without any serious flaws found
  - best known algorithm takes exponential time in security parameter (key length |n|)
- how can you break RSA if you can factor?

In fact, it corresponds to a new assumption, the RSA assumption.

Security of RSA (cont.)

Based on difficulty of **factoring** large numbers (into large primes), i.e., $n = p \cdot q$ into p, q

Current practice is using 2,048-bit long RSA keys (617 decimal digits)

- estimated computing/memory resources needed to factor an RSA number within one year
- Bit security
  - **1024-bit RSA:** ~80-bit security
  - **2048-bit RSA:** ~112-bit security
  - **3072-bit RSA:** ~128-bit security
  - **7680-bit RSA:** ~192-bit security

Length (bits)	PCs	Memory
430	1	128MB
760	215,000	4GB
1,020	342×10^{6}	170GB
1,620	1.6×10^{15}	120TB

RSA challenges

Challenges for breaking the RSA cryptosystem of various key lengths (i.e., |n|)

- known in the form RSA-`key bit length’ expressed in bits or decimal digits
- provide empirical evidence/confidence on strength of specific RSA instantiations

Known attacks

- RSA-155 (**512-bit**) factored in **4 mo**. using 35.7 CPU-years or 8000 Mips-years (**1999**) and 292 machines
    - 160 175-400MHz SGI/Sun, 8 250MHz SGI/Origin, 120 300-450MHz Pent. II, 4 500MHz Digital/Compaq
- RSA-**640** factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-220 (**729-bit**) factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-232 (**768-bit**) factored in **2 years** using **parallel** computers 2K CPU-years (1-core 2.2GHz AMD Opteron) (**2009**)

Most interesting challenges

- prizes for factoring RSA-**1024**, RSA-**2048** is $100K, $200K – estimated at 800K, 20B MIPS-centuries

Deriving an RSA key pair

- public key is pair of integers (e,n), secret key is (d, n) or d
- the value of n should be quite large, a product of two large primes, p and q
- often p, q are nearly 100 digits each, so n ~= 200 decimal digits (~512 bits)
  - but 2048-bit keys are becoming a standard requirement nowadays
- the larger the value of n the harder to factor to infer p and q
  - but also the slower to process messages
- a relatively large integer e is chosen
  - e.g., by choosing e as a prime that is larger than both $(p - 1)$ and $(q - 1)$; why?
- d is chosen s.t. $e \cdot d = 1 \bmod (p - 1)(q - 1)$; how?

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ(n)** = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is **C** = **M**^{3} mod 55 for **M** in $\mathbf{Z_{55}}$

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ**(**n**) = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is **C** = $\mathbf{M}^3$ mod 55 for **M** in $\mathbf{Z_{55}}$
- Assume n = 20434394384355534343545428943483434356091 = p · q
  - can e be the number 4343253453434536?
- Are there problems with applying RSA in practice?
  - what other algorithms are required to be available to the user?
- Are there problem with respect to RSA security?
  - does it satisfy CPA (advanced) security?

Algorithmic issues

The implementation of the RSA cryptosystem requires various algorithms

- Main issues
  - representation of integers of arbitrarily large size; and
  - arithmetic operations on them, namely computing modular powers
- Required algorithms (at setup)
  - generation of **random numbers** of a given number of bits (to compute candidates **p**, **q**)
  - **primality testing** (to check that candidates **p, q** are prime)
  - computation of the **GCD** (to verify that **e** and **ϕ**(**n**) are relatively prime)
  - computation of the **multiplicative inverse** (to compute **d** from **e**)

Pseudo-primality testing

Testing whether a number is prime (**primality testing**) is a difficult problem

An integer **n** $\geq$ 2 is said to be a base-**x** **pseudo-prime** if

  - $\mathbf{x}^{\mathbf{n}-1}$ mod **n** = 1 (Fermat’s little theorem)

- Composite base-**x** pseudo-primes are rare
  - a random 100-bit integer is a composite base-2 pseudo-prime with probability less than 10^{-13}
  - the smallest composite base-2 pseudo-prime is 341
- Base-**x** pseudo-primality testing for an integer **n**
  - check whether $\mathbf{x}^{\mathbf{n}-1}$ mod **n** = 1
  - can be performed efficiently with the repeated squaring algorithm

Security properties

- Plain RSA is deterministic
  - why is this a problem?
- Plain RSA is also homomorphic
  - what does this mean?
  - multiply ciphertexts to get ciphertext of multiplication!
  - $[(m_1)^e \bmod N][(m_2)^e \bmod N] = (m_1 m_2)^e \bmod N$
  - however, not additively homomorphic

Real-world usage of RSA

- Randomized RSA
  - to encrypt message M under an RSA public key (e,n), generate a new random session AES key K, compute the ciphertext as [K^e mod n, $AES_K(M)$]
  - prevents an adversary distinguishing two encryptions of the same M since K is chosen at random every time encryption takes place
- Optimal Asymmetric Encryption Padding (OAEP)
  - roughly, to encrypt M, choose random r, encode M as $M' = [X = M \oplus H_1(r)\ ,\ Y= r \oplus H_2(X)\ ]$ where H_1 and H_2 are cryptographic hash functions, then encrypt it as $(M')^{e}$ mod n

Summary of message-authentication crypto tools

	Hash (SHA2-256)	MAC	Digital signature
Integrity	Yes	Yes	Yes
Authentication	No	Yes	Yes
Non-repudiation	No	No	Yes
Crypto system	None	Symmetric (AES)	Asymmetric (e.g., RSA)

9.1 User authentication

User identification & authentication

Identification

- asserting who a person is

Authentication

- proving that a user is who she says she is
- methods
  - something the user *knows*
  - something the user *is*
  - something user *has*

Does authentication imply identification?

Suppose that a user

- provides her (login) name and
- uses one of the three methods to authenticate into a computer system
  - either terminal or remote server via a web browser
- when does user authentication imply user identification?
  - not quite…

Example: Something you know

The user has to know some secret to be authenticated

- password, personal identification number (PIN), personal information like home address, date of birth, name of spouse (“security” questions)

But anybody who obtains your secret “is you…”

- impersonation Vs. delegation
- you leave no trace if you pass your secret to somebody else

What if there is a case of computer misuse?

- i.e., where somebody has logged in using your username & password…
- can you prove your innocence?
- can you prove that you have not divulged your password?

Thus…

- a password does not authenticate a person
- successful authentication only implies that the user knew a particular secret
- there is no way of telling the difference between the legitimate user and an intruder who has obtained that user’s password
- **unfortunately: this holds true for almost all of authentication methods…**

9.1.1 Something you know – password authentication

Something you know

- passwords
  - or PINs
  - or answers to “security” questions (e.g., where did you meet your wife?)

Problems with passwords

Many attack vectors…

- password "live" in different "places:"
  1) user's brain, 2) channel & 3) authentication server

1) password guessing

- predict weak passwords

2) phishing & spoofing or **cached passwords**

- deceive users to reveal their password

3) leaked password files

- steal user credentials

Password guessing

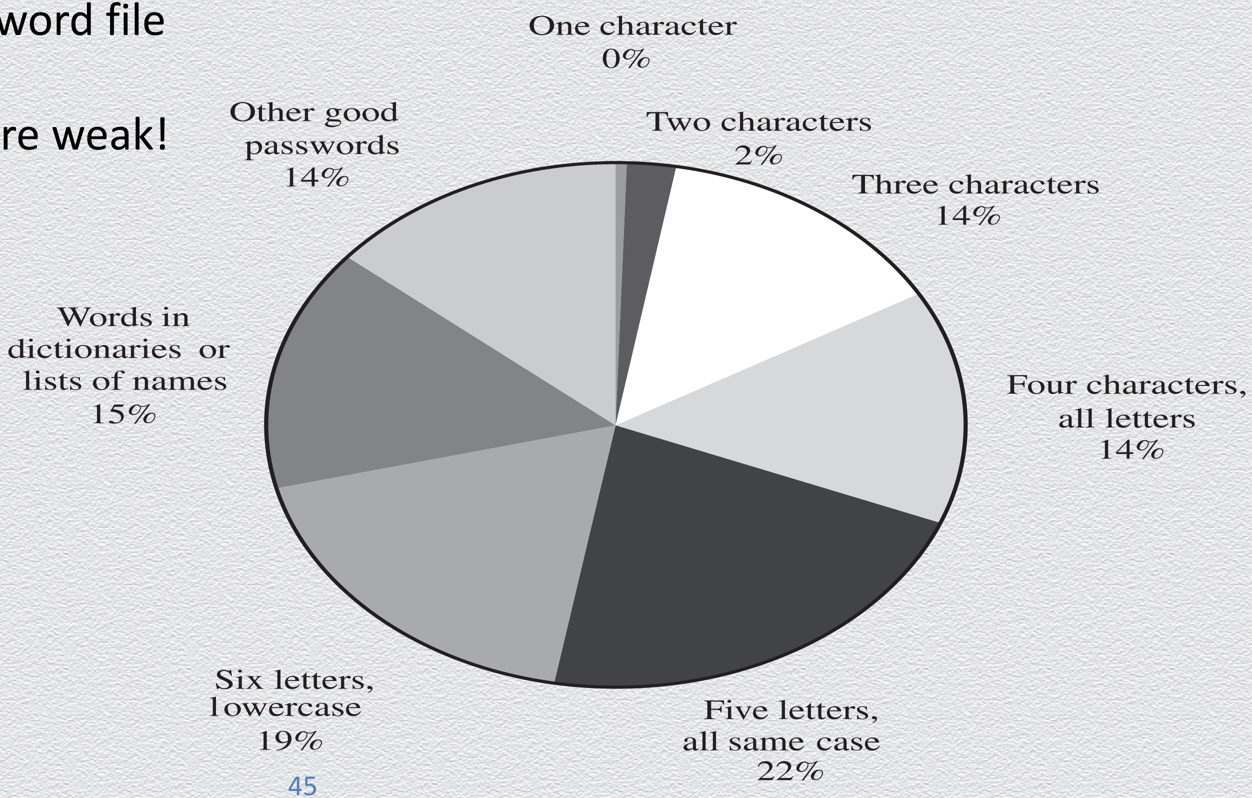
Infer passwords through guessing

- Low-entropy passwords
  - To be easy to remember, passwords are often weak easy-to-predict secrets
  - e.g., password is “Password1”
- Password reuse
  - To be easy to remember, passwords are often reused across many authentication servers
  - e.g., same password for all accounts

Distribution of password types

Graph from an old leaked password file

The point is: Most passwords are weak!



Online dictionary attacks

- Direct brute-force or dictionary attacks against passwords
  - employs only the authentication system
  - attacker tries to impersonate a victim by trying
    - all possible (short length) passwords or
    - passwords coming from a known dictionary
  - (cf. offline brute-force or dictionary attacks using leaked hashed passwords)
- Countermeasure
  - block login & lock account after many consecutive failed authentication attempts
  - false negatives…

Phishing & spoofing

- Identification and authentication through username and password provide **unilateral authentication**
- Computer verifies the user’s identity but the user has no guarantees about the identity of the party that has received the password
- In **phishing** and **spoofing** attacks a party voluntarily sends the password over a channel, but is misled about the end point of the channel

Spoofing

- Attacker starts a malicious program that presents a fake login screen and leaves the computer

- If the next user coming to this machine enters username and password on the fake login screen, these values are captured by the malicious program
    - login is then typically aborted with a (fake) error message and the spoofing program terminates
    - control returns to operating system, which now prompts the user with a genuine login request
    - thus, the victim does not suspect that something wrong has happened
        - the victim may think that the password was mistyped…

Counteracting password spoofing

- display **number of failed logins**
  - may indicate to the user that an attack has happened
- **trusted path**
  - guarantee that user communicates with the operating system and not with a spoofing program
- **mutual authentication**
  - user authenticated to system, system authenticated to user

Phishing

- attacker impersonates the system to trick a user into releasing the password
- e.g.,
  - a message could claim to come from a service you are using
  - tell you about an upgrade of the security procedures
  - and ask you to enter your username and password
    at the new security site that will offer stronger protection
- attacker impersonates the user to trick a system operator into releasing the password to the attacker
  - **social engineering**

Cached passwords

- description of login has been quite abstract
  - password travels directly from user to the password checking routine
- in reality, it will be held temporarily in intermediate storage locations
  - e.g., like buffers, caches, or a web page
- management of these storage locations is normally beyond user's control
  - a password may be kept longer than the user has bargained for

Leaked password files

- Breach authentication server to steal user credentials
  - e.g., plaintext passwords

- Countermeasures
  - protect passwords via encryption (e.g., a symmetric-key cipher)
    - subject to keeping the secret key secure against the server's compromise…
    - hard to achieve in practice…
  - concealed password via hashing
    - subject to meeting conditions for secret concealment via hashing…

Protecting the password file

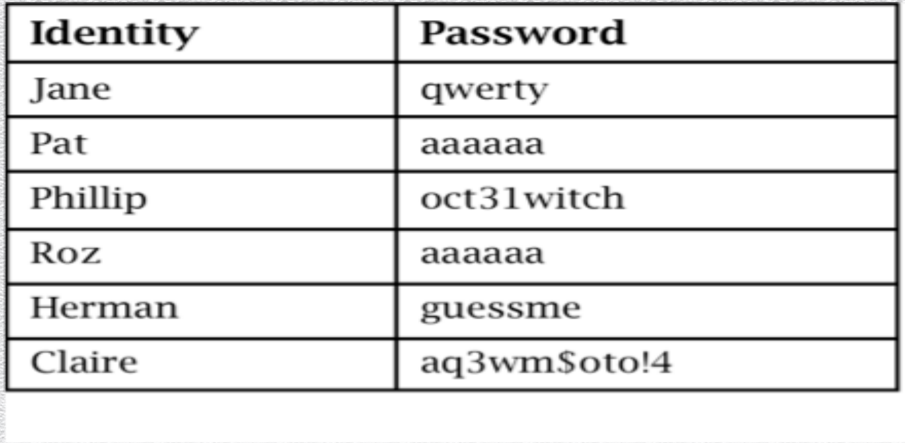
Operating system maintains a password file (with user names & passwords)

- attacker could try to compromise its confidentiality or integrity
- options for protecting the password file
  - cryptographic protection
  - access control enforced by the operating system
  - combination of cryptographic protection and access control, possibly with further measures to slow down dictionary attacks

Access control settings

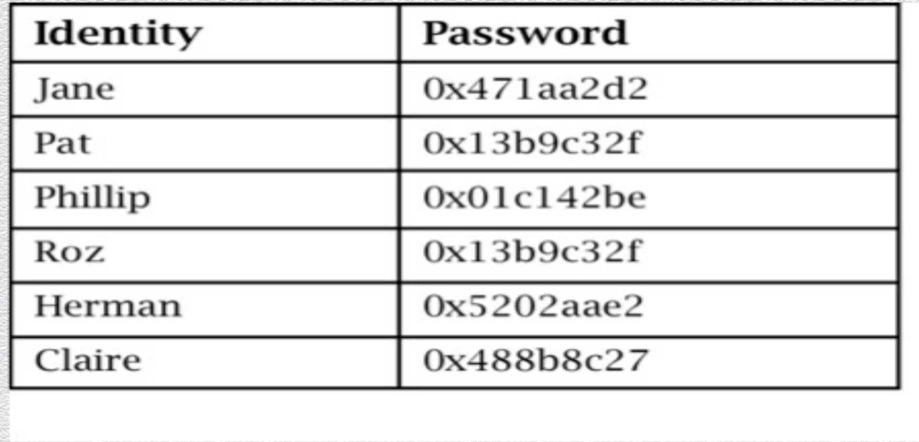
- only privileged users must have write access to the password file
  - an attacker could get access to the data of other users simply by changing their password
  - even if it is protected by cryptographic means
- if read access is restricted to privileged users, passwords could be stored unencrypted
  - in theory – in practice, bad idea because of breaches
- if password file contains data required by unprivileged users, passwords must be "encrypted"; such a leaked file can still be used in dictionary attacks
  - typical example is **/etc/passwd** in Unix
  - many Unix versions store encrypted passwords in a shadow password file (not publicly accessible)

Example: Password storage via hashing



Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto!4

Plaintext



Identity	Password
Jane	0x471aa2d2
Pat	0x13b9c32f
Phillip	0x01c142be
Roz	0x13b9c32f
Herman	0x5202aae2
Claire	0x488b8c27

Concealed

Subject to “concealment” preconditions

If fully concealed, are we safe?

Any hash pre-image leads to impersonation

Hashing passwords is not enough

An immediate control against password leakage through stolen password files, involves concealing passwords stored at the authentication server via hashing

Why are offline dictionary attacks quite effective using leaked hashed passwords in practice?

- Most hashed passwords are weak passwords
- Thus, they can be “cracked”
  - Invert the hash
  - Find a 2nd preimage of the hash

Password cracking

Given leaked hashed passwords, recover passwords

- Use exhaustive search by hashing over guessed passwords…
    - brute-force attack: try all possibilities
    - dictionary attacks: try all words in a dictionary & variations of them
    - rainbow tables: try possibilities in a systematic way via a data structure
- These methods impose different time-space trade-offs on attacker’s workload
    - preprocessing is often very useful, e.g.,
        - precompute a dictionary-based set of password-hash pairs
        - use precomputed set for cracking any newly leaked hashed passwords

Countermeasures

Password salting

U, h(PU||**SU**), SU

Now preprocessing is useless;
or it must be **user specific**!

- to slow down dictionary attacks
  - a user-specific **salt** is appended to a user's password before it is being hashed
  - each salt value is stored in the clear along with its corresponding hashed password
  - if two users have the same password, they will have different hashed passwords
  - example: Unix uses a 12 bit salt

Hash strengthening

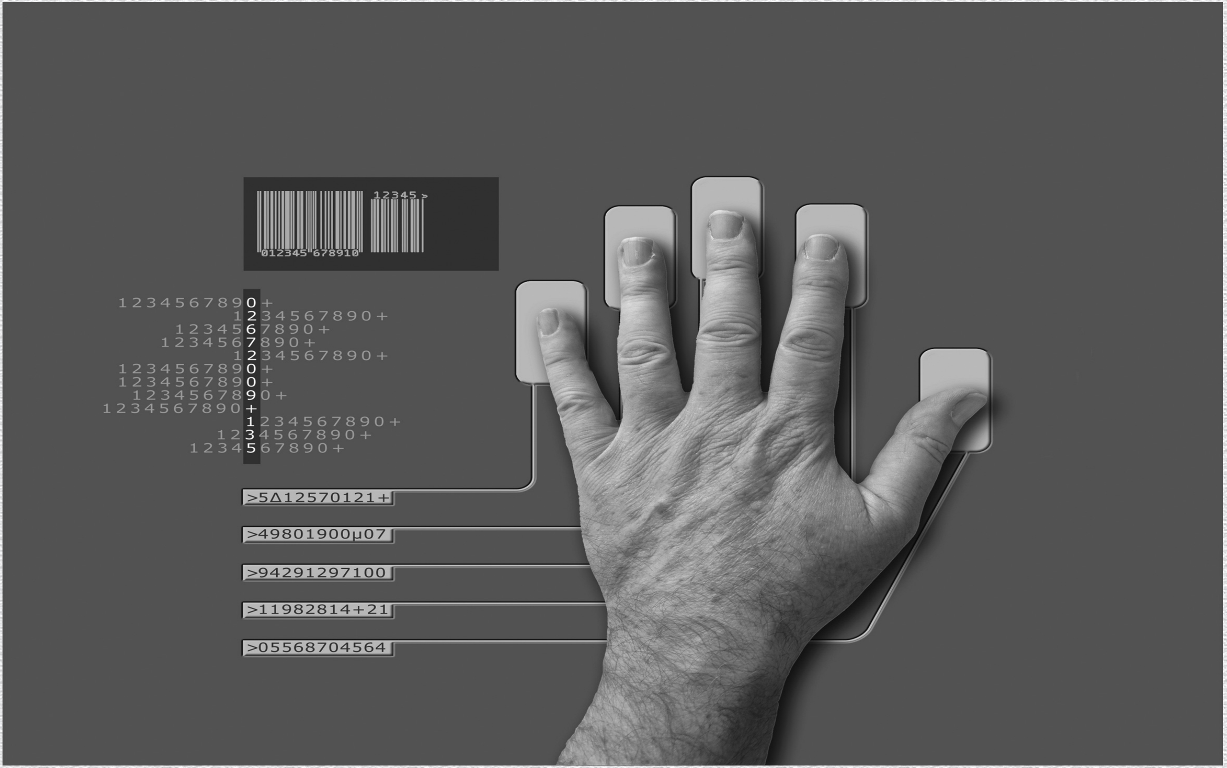
- to slow down dictionary attacks
  - a password is hashed k times before being stored

9.1.2 Something you are – biometric authentication

Something you are

- biometric schemes use people’s unique physical characteristics
    - traits, features
    - face, finger prints, iris patterns, hand geometry
- biometrics may seem to be the most secure solution for user authentication
- biometric schemes are still quite new

Biometrics: Something you are



Problems with biometrics

- Intrusive
- Expensive
- Single point of failure
- Sampling error
- False readings
- Speed
- Forgery

Fingerprint

- Enrolment
  - reference sample of the user’s fingerprint is acquired at a fingerprint reader
- Features are derived from the sample
  - fingerprint minutiae
    - end points of ridges, bifurcation points, core, delta, loops, whorls, …
- For higher accuracy, record features for more than one finger
- Feature vectors are stored in a secure database
- When the user logs on, a new reading of the fingerprint is taken
  - features are compared against the reference features

Identification Vs. verification

- Biometrics are used for two purposes
  - Identification: 1:n comparison, i.e., identify user from a database of n persons
  - Verification: 1:1 comparison, i.e., check whether there is a match for a given user
- Authentication by password
  - clear reject or accept at each authentication attempt
- Biometrics
  - stored reference features will hardly ever match precisely features derived from the current measurements

Failure rates

- Measure similarity between reference features and current features
- User is accepted if match is above a predefined threshold
- **New issue: false positives and false negatives**
- Accept wrong user (false positive)
  - security problem
- Reject legitimate user (false negative)
  - creates embarrassment and an inefficient work environment

Forgeries

Fingerprints, and biometric traits in general, may be unique but they are no secrets!

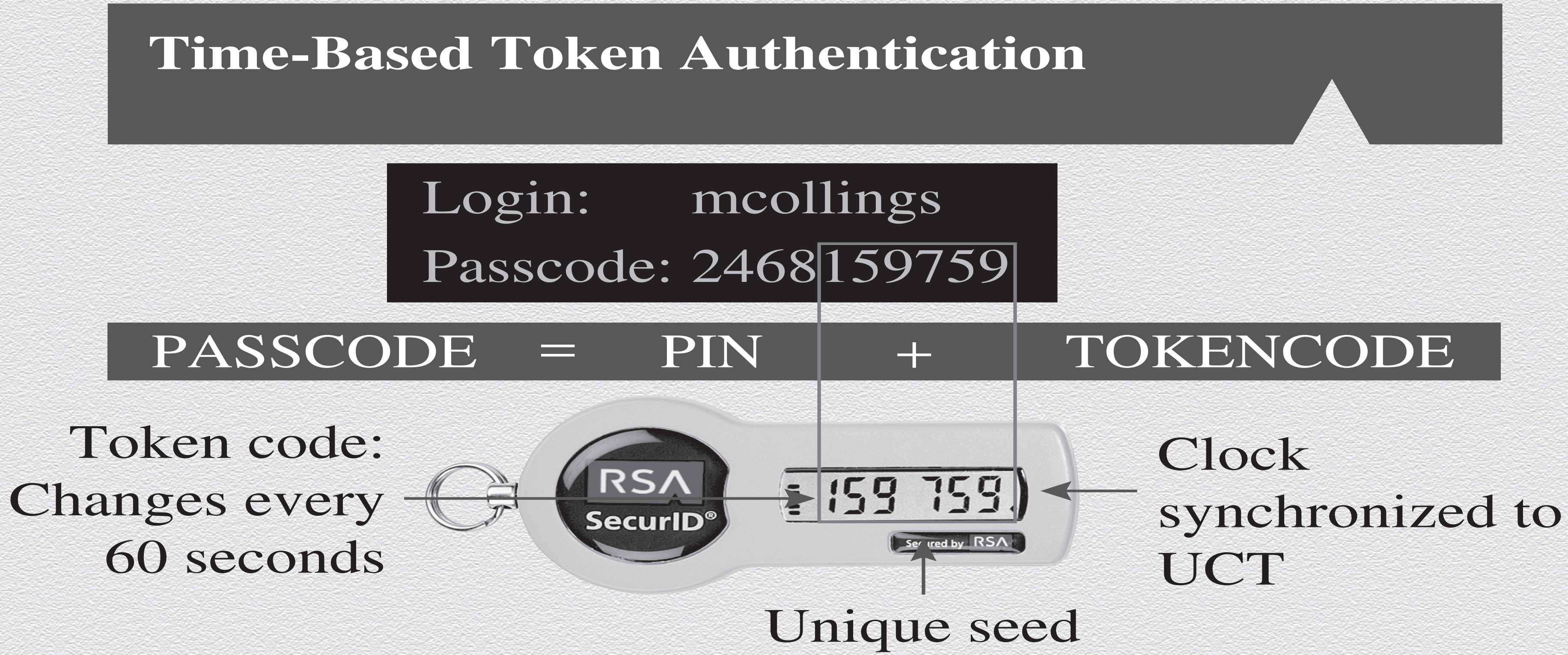
- you are leaving your fingerprints in many places
- rubber fingers have defeated commercial fingerprint-recognition
- minor issue if authentication takes place in the presence of security personnel
  - when authenticating remote users additional precautions have to be taken
- user acceptance: so far fingerprints have been used for tracing criminals

9.1.3 Something you have – authentication tokens

Something you have

- user presents a physical token to be authenticated
  - keys, cards or identity tags (access to buildings), smart cards
- limitations
  - physical tokens can be lost or stolen
  - anybody in possession of token has the same rights as legitimate owner
- physical tokens are often used in combination with something you know
  - e.g. bank cards come with a PIN or with a photo of the user
  - this is called: **2nd-factor authentication or multi-factor authentication**

Tokens: something you have



Time-Based Token Authentication

Login: mcollings

Passcode: 2468159759

PASSCODE = PIN + TOKENCODE

Token code: Changes every 60 seconds

Clock synchronized to UCT

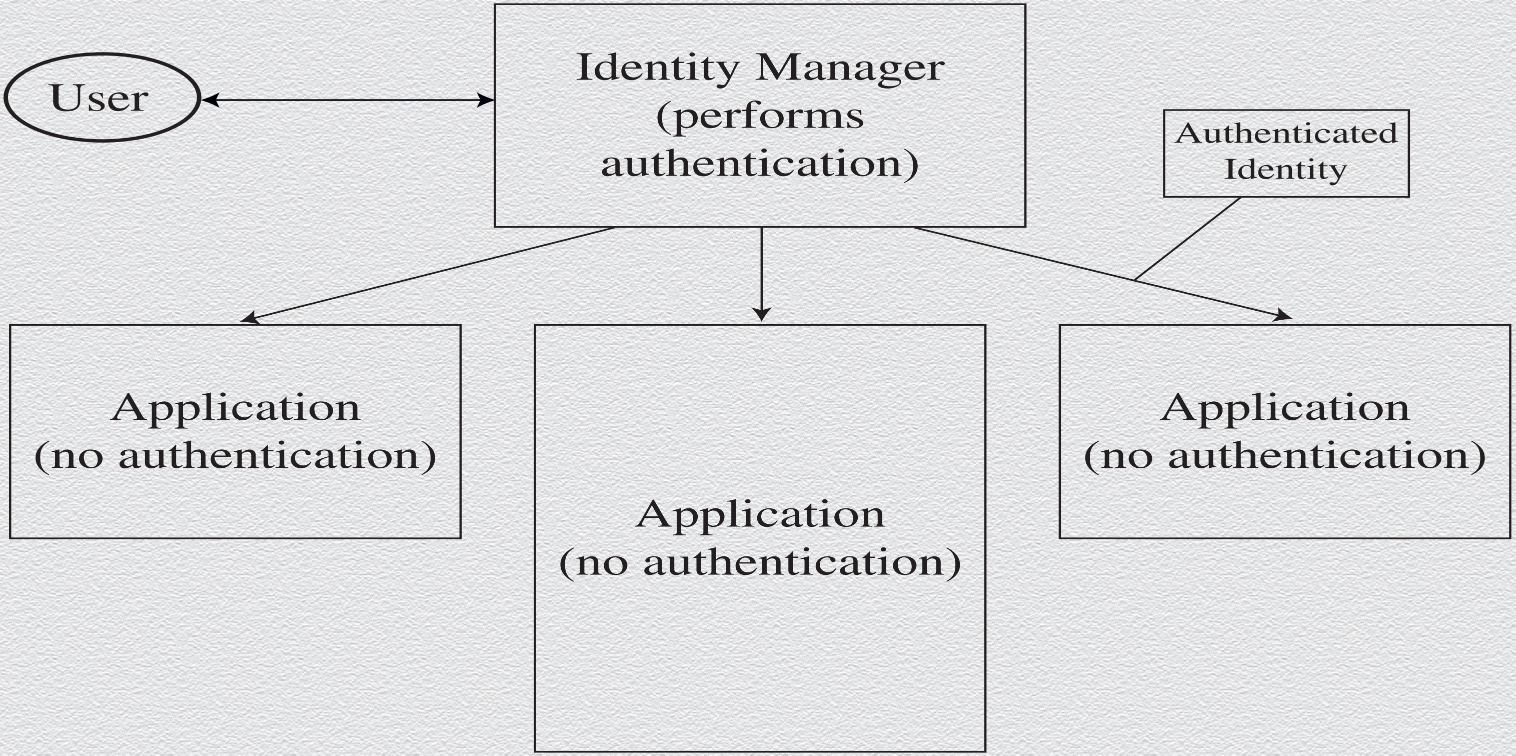
Unique seed

Problems with tokens

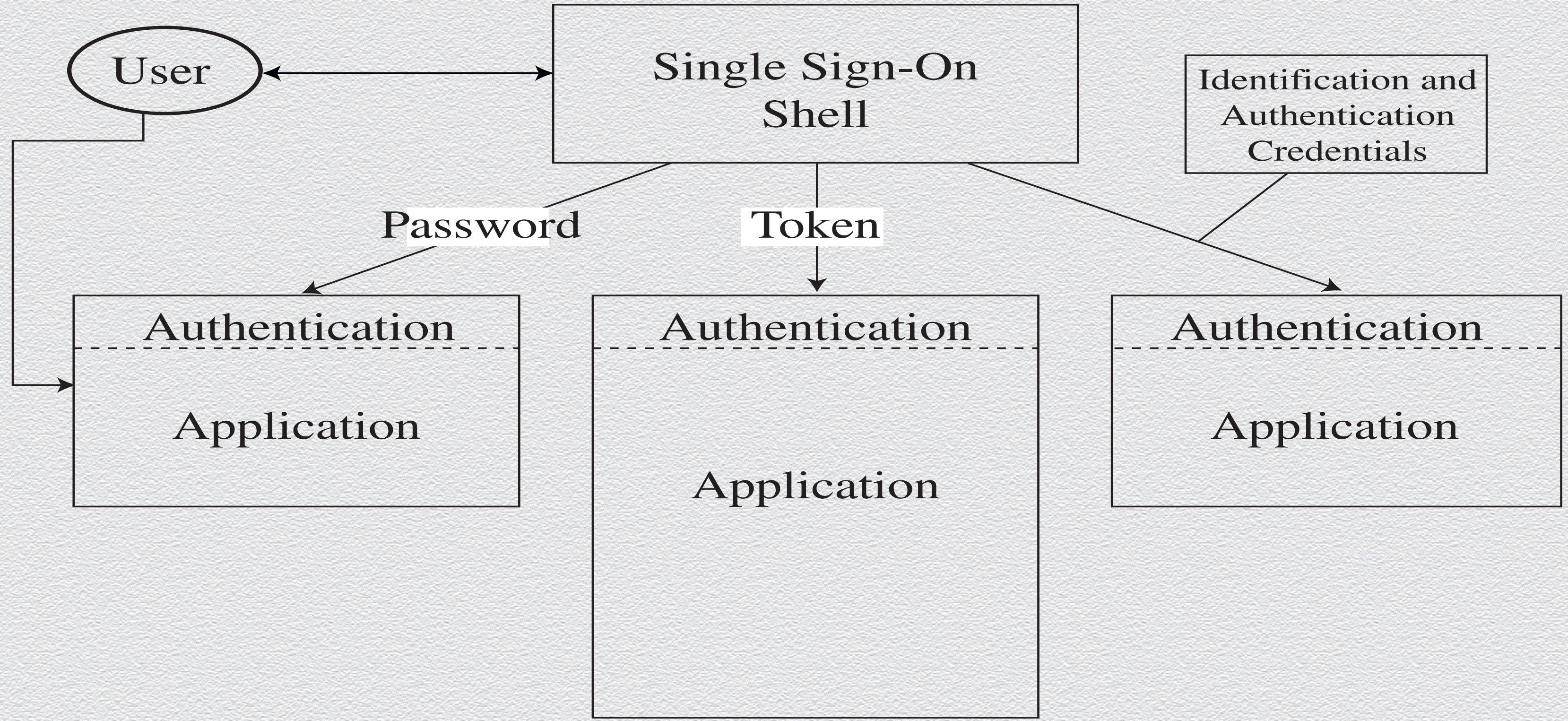
- Inconvenience
- Lost token
- Stolen token
- Cloned token
- Side-channel attacks (for key exfiltration)

9.1.4 Other methods

Federated identity management


User
Identity Manager (performs authentication)
Authenticated Identity
Application (no authentication)
Application (no authentication)
Application (no authentication)

SSO: Single Sign-On



More details on SSO

- Having to remember many passwords for different services is a nuisance
  - with a single sign-on service, you have to enter your password only once
  - an alternative solution: password managers
- A simplistic single-sign on service could store your password and do the job for you whenever you have to authenticate yourself
  - such a service adds to your convenience but it also raises new security concerns
- System designers have to balance convenience and security
  - ease-of-use is an important factor in making IT systems really useful
  - but many practices which are convenient also introduce new vulnerabilities

More on authentication

If dissatisfied with security level provided by passwords?

- you can be authenticated on the basis of
  - something you know
  - something you have
  - something you are
  - **what you do – behavioural**
  - **where you are – location based**

What you do

- people perform mechanical tasks in a way that is both repeatable and specific to the individual
- experts look at the dynamics of handwriting to detect forgeries
- users could sign on a special pad that measures attributes like writing speed and writing pressure
- on a keyboard, typing speed and key strokes intervals can be used to authenticate individual users
- more recently behaviours from one's mobile phone have been studied

Where you are

- some OSs grant access only if you log on from a certain terminal
  - a system administration may only log on from an operator console but not from an arbitrary user terminal
  - users may be only allowed to log on from a workstation in their office
- common method in mobile and distributed computing
- Global Positioning System (GPS) might be used to established the precise geographical location of a user during authentication

9.1.5* More on password cracking

Password cracking methods

- Brute force
  - Try all passwords (in a search space) for inverting a specific password hash
  - Eventually succeeds given enough time & CPU power
- Dictionary
  - Precompute & store by hash (hash, password) pairs of a set of likely passwords
  - Fast look up for password given the hash
  - Large storage & preprocessing time
- Rainbow table
  - Partial dictionary of hashes
  - More storage, shorter cracking time

Brute force cracking: Method

- Try all passwords (for a given password space)
- Parallelizable
- Eventually succeeds given enough time & computing power
- Best done with GPUs and specialized hardware (e.g., FPGAs or ASIC)
- Large computational effort for each password cracked

Brute force cracking: Search space

Assume a standard keyboard with 94 characters

Password length	Number of passwords
5	94^5 = 7,339,040,224
6	94^6 = 689,869,781,056
7	94^7 = 64,847,759,419,264
8	94^8 = 6,095,689,385,410,816
9	94^9 = 572,994,802,228,616,704

Brute force cracking: Computational effort

Say, the attacker has 60 days to crack a password by exhaustive search assuming a standard keyboard of 94 characters.

How many hash computations per second are needed?

5 characters:	1,415
6 characters:	133,076
7 characters:	12,509,214
8 characters:	1,175,866,008
9 characters:	110,531,404,750

Dictionary attack: Method

- Precompute hashes of a set of likely passwords
- Parallelizable
- Store (hash, password) pairs sorted by hash
- Fast look up for password given the hash
- Requires large storage and preprocessing time

Dictionary attack: Example

STEP 1: Make a plaintext password file of bad passwords (called `wordlist`):

```
triandop12345
letmein
zaq1zaq1
```

STEP 2: Generate MD5 hashes:

```
for i in $(cat wordlist); do
    echo -n "$i" | md5 | tr -d " *-"; done > hashes
```

STEP 3: Get a dictionary file.

E.g., using rockyou.txt which lists most common passwords from the RockYou hack in 2009.

Dictionary attack: Intelligent guessing

Try the top N most common passwords

- e.g., check out several lists of passwords on known repositories

Try passwords generated by

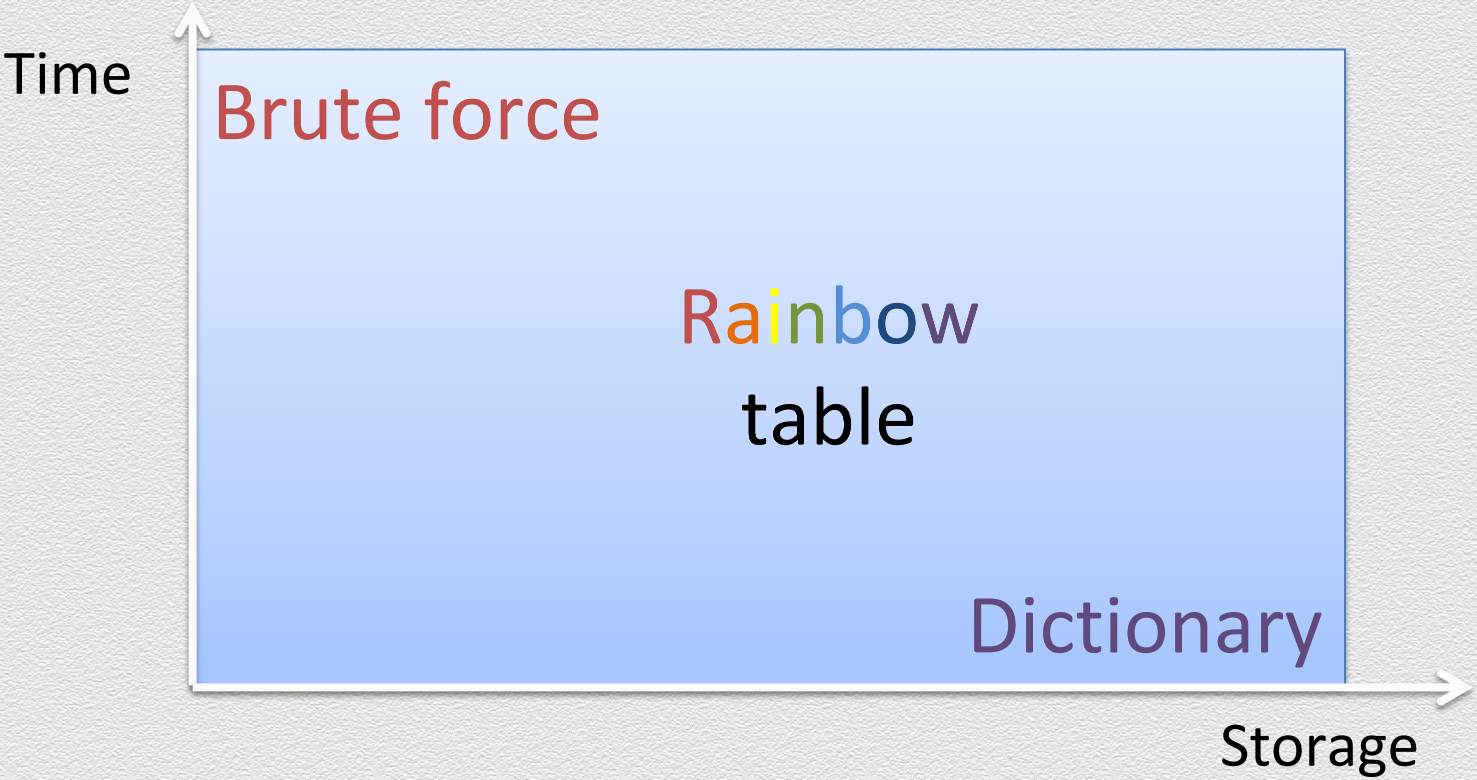
- a dictionary of words, names, places, notable dates along with
    - combinations of words & replacement/interspersion of digits, symbols, etc.
- a syntax model
    - e.g., 2 words with some letters replaced by numbers: elitenoob, e1iten00b, …
- a Markov chain model or a trained neural network

Password tracking tradeoffs

1980 - Martin Hellman

- Achieves (possibly useful) time Vs. memory tradeoffs
- Idea: Reduce time needed to crack a password by using a large amount of memory
    - **Benefits**
        - Better efficiency than brute-forcing methods
    - **Flaws**
        - This kind of database takes tens of memory’s terabytes

Password cracking tradeoffs (cont.)



Password cracking tradeoffs (cont.)

Brute-force: no preprocessing, no storage, very slow cracking

Dictionary: very slow preprocessing, huge storage, very fast cracking

Rainbow tables: **tunable** tradeoff between storage space & cracking time

- Trade more storage for faster cracking

Password space of **size n**

Method	Storage	Preprocessing	Cracking
Brute-force	~ 0	~ 0	n
Dictionary	n	n	~ 0
Rainbow table, $mt^2 = n$	mt	mt^2	$t^2/2$

All costs relate to **hashing**

Rainbow tables

- Use data-structuring techniques to get desirable time Vs. memory tradeoffs
- Main challenge
  - Cryptographic hashing is random and exhibits no patterns
  - E.g., no ordering can be exploited to allow for an efficient search data structure
- Main idea
  - Establish a type of “ordering” by randomly mapping hash values to passwords
  - E.g., via a “reduction” function that produces password “chains”

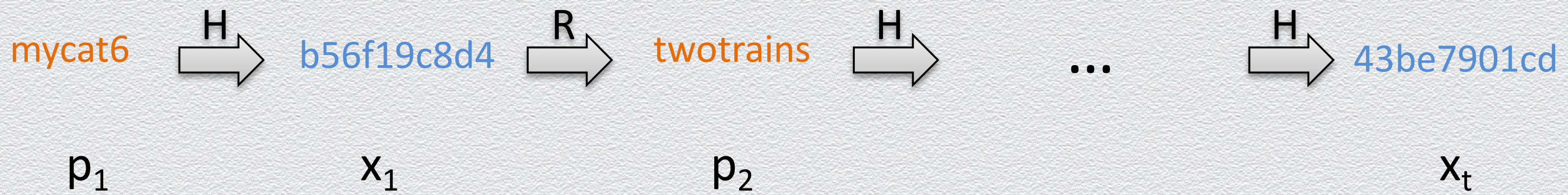
Reduction function

Maps a hash value to a pseudorandom password from a given password space

- E.g., reduction function p = R(x) for 256-bit hashes & 8-character passwords from a 64-symbol alphabet a_1, a_2, …, a_{64}
  - Split hash x into 48-bit blocks x_1, $x_{2,}$ …, x_5 and one 16-bit block x_6
  - Compute $y = x_1 \oplus x_2 \ldots \oplus x_5$
  - Split y into 6-bit blocks y_1, $y_{2,}$ …, y_8
  - Let $p = a_{y_1}, a_{y_{2,}} \ldots, a_{y_8}$
- This method can be generalized to arbitrary password spaces

Password chain

- Sequence (of size t) alternating **passwords** & **hashes**
  - Start with a random password p_1
  - Alternate using cryptographic hash function H & reduction function R
    - $x_i = H(p_i)$, $p_{i+1} = R(x_i)$
  - End with a hash value x_t

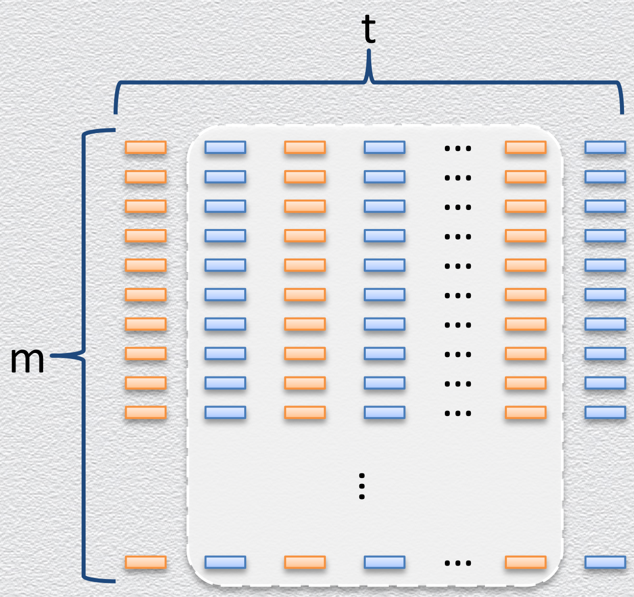


mycat6 H b56f19c8d4 R twotrains H … H 43be7901cd

p_1 x_1 p_2 x_t

Hellman’s method

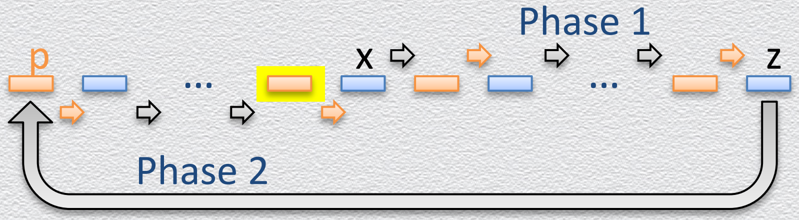
- Starting from m random passwords, build a table of m password chains, each of length t
- The expected number of distinct passwords in a table is $\Omega(mt)$
- Compressed storage:
  - For each chain, keep only the first password, p, and the last hash value, z
  - Store pairs (z, p) in a dictionary D indexed by hash value z



Classic password recovery

Recovery of password with hash value x

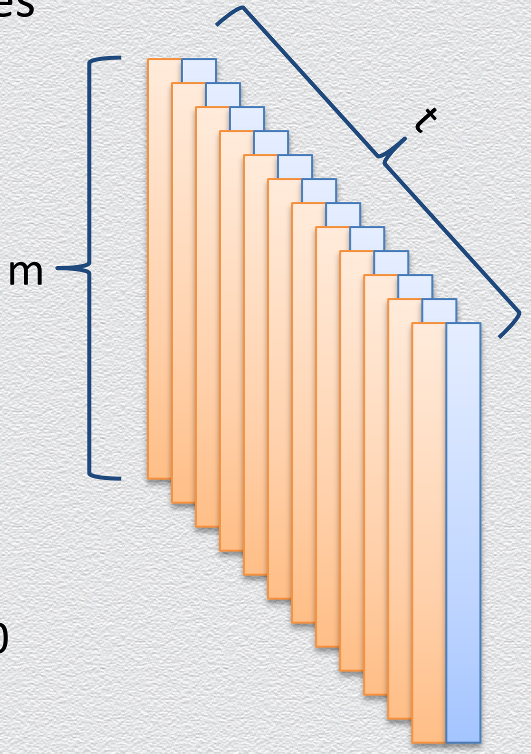
- Step 1: traverse the suffix of the chain starting at x
  - y = x;
  - while p = D.get(y) is null
    - y = H(R(y)) //advance
    - if i++ > t return “failure” //x not in the table
- Step 2: traverse the prefix of the chain ending at x
  - while y = H(p) $\neq$ x
    - p = R(y) //advance
    - if j++ > t return “failure” //x not in the table
  - return p //password recovered



High-probability recovery

Collisions in the reduction function result in recovery issues

- Mitigate the impact of collisions, using t tables with <u>distinct</u> reduction functions R
- If $m \cdot t^2 = O(n)$, n passwords are covered with high probability



Performance

- Storage: mt cryptographic hash values
- Recovery: t^2 hash computations & t^2 dictionary lookups
- E.g., n = 1,000,000,000, $m = t = n^{1/3}$, $mt = t^2 = n^{2/3}$ = 1,000,000

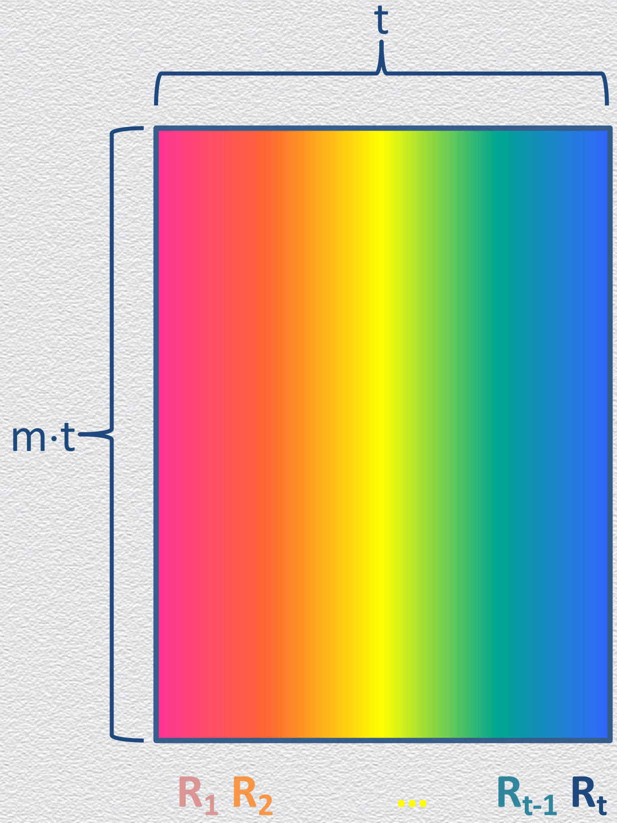
Rainbow table

Instead of t different tables, use a single table with

- O(m·t) chains of length t
- Distinct reduction function at each step
- Visualizing the reduction functions with a gradient of colors yields a **rainbow**

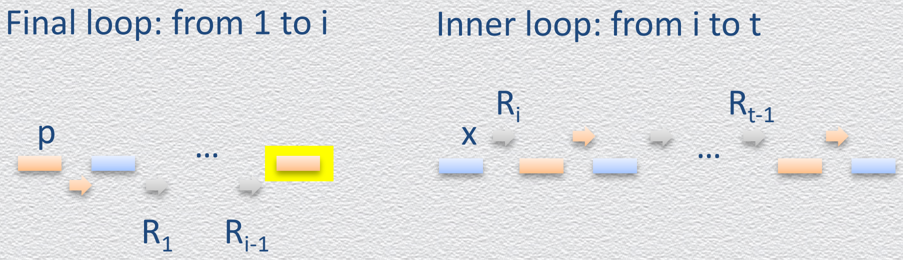
Performance

- Storage : mt hash values (as before)
- Recovery : $t^2/2$ hash computations & t dictionary lookups (lower than before)



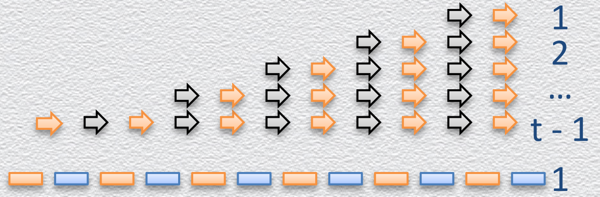
Rainbow-table password recovery

```
for i = t, (t – 1), … , 1
        y = x //x is password hash we want to crack
        for j = i, …, t - 1 //traverse from i to t
                y = H(R_j(y)) //advance
        if p = D.get(y) is not null //candidate position i
                for j = 1 … i - 1 //traverse from 1 to i
                        p = R_j(H(p)) //advance
                if H(p) = x   return p //password recovered
                else return “failure” //x not in the table
return “failure” //x not in the table
```



Worst-case # of hashing

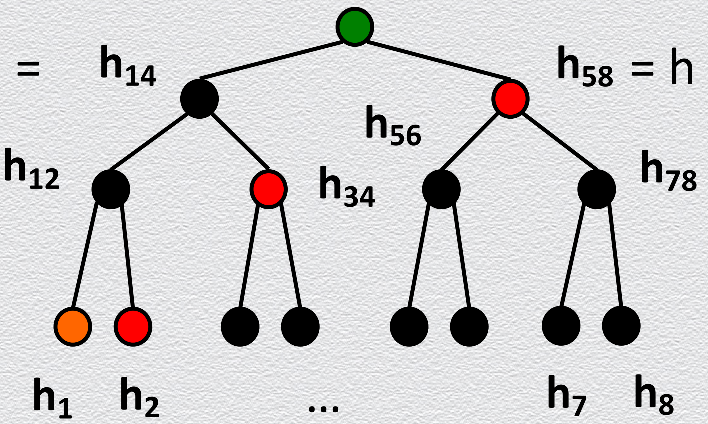
$1 + 2 + … + (t -1) + 1 ≈ t^2/2$



9.2 Data authentication: The Merkle tree

Application 6: The Merkle tree

An alternative (to Merkle-Damgård) method to achieve domain extension



d = h (h_{14} || h_{48})

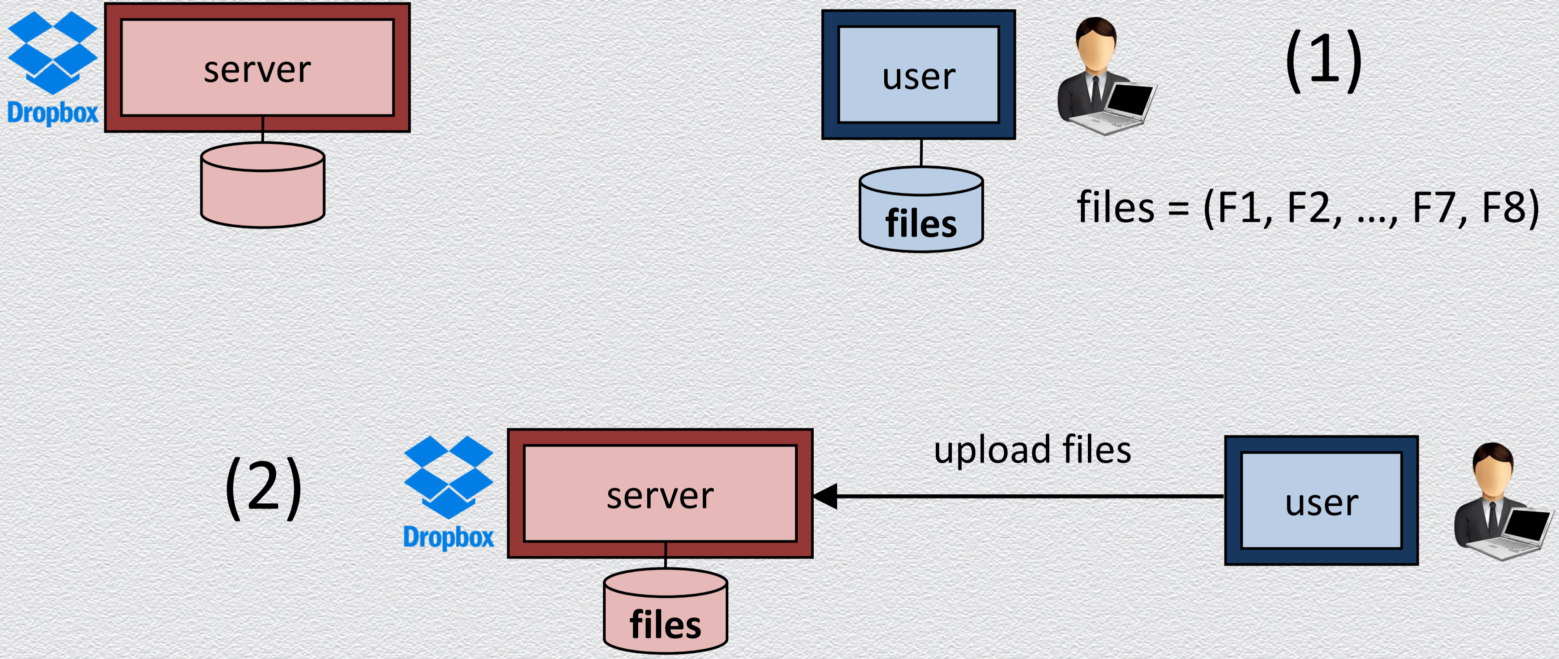
h (h_{12} || h_{34}) = **h_{14}**

h_{58} = h (h_{56} || h_{78})

h (h_1 || h_2) = **h_{12}**

let $h_i = h(F_i)$, $1 \leq i \leq n$

Example 6: Secure cloud storage



Dropbox

server

user

(1)

files

files = (F1, F2, …, F7, F8)

(2)

Dropbox

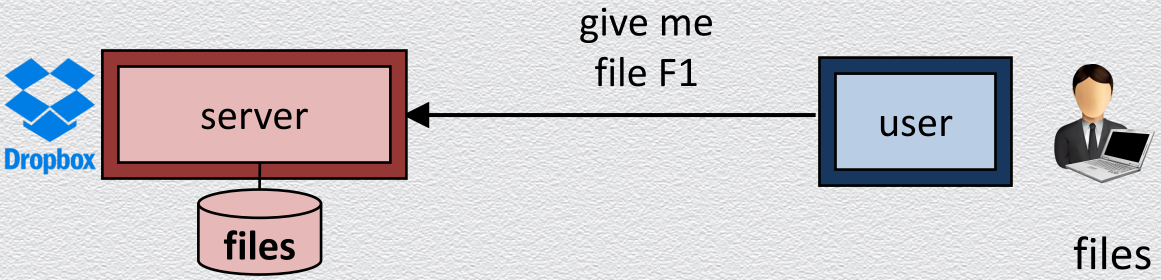
server

upload files

user

files

Example 6: Secure cloud storage



give me file F1

Dropbox

server

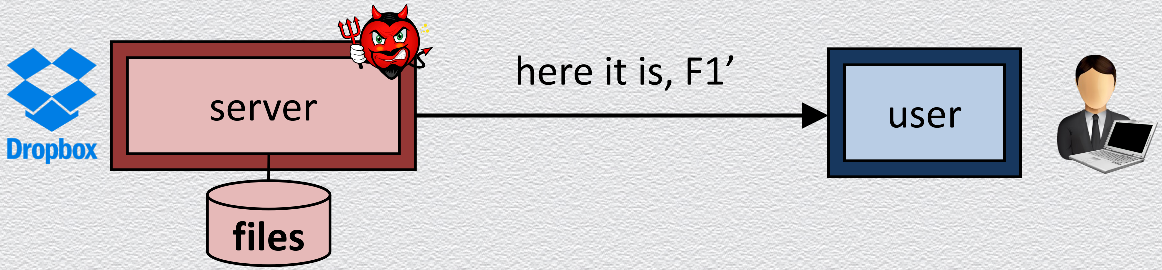
files

user

(3)

files = (F1, F2, …, F7, F8)

(4)



Dropbox

server

files

here it is, F1’

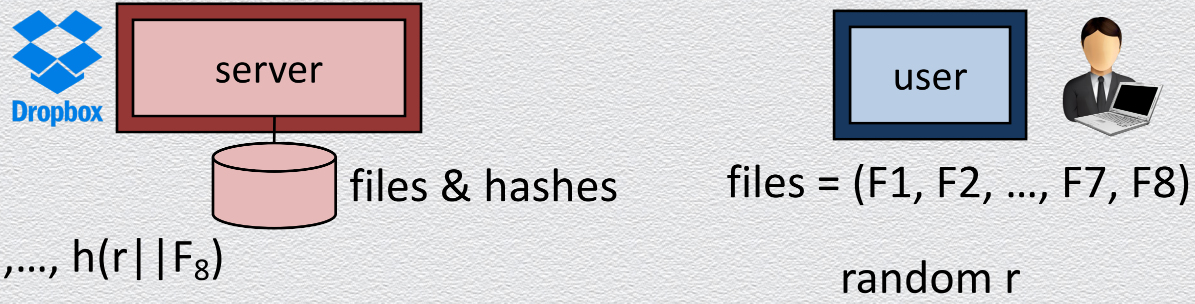
user

F1’ = #@$@!#^@$^…   (altered)

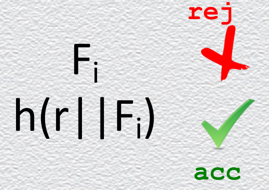
Example 6: Secure cloud storage – per-file hashing

Bob wants to outsource storage of files $F_1, F_2,\ldots,F_8$ to Dropbox & check their integrity

- Bob stores random r
  (& keeps it secret)
- Bob sends to Dropbox
  - files $F_1, F_2,\ldots,F_8$
  - hashes $h(r||F_1), h(r||F_2),\ldots, h(r||F_8)$



Every time Bob **reads** a file F_i, he also reads $h(r||F_i)$ to verify F_i's integrity



- any problems with **writes**?

Example 6: Secure cloud storage – per-file-set hashing

Dropbox

server

user

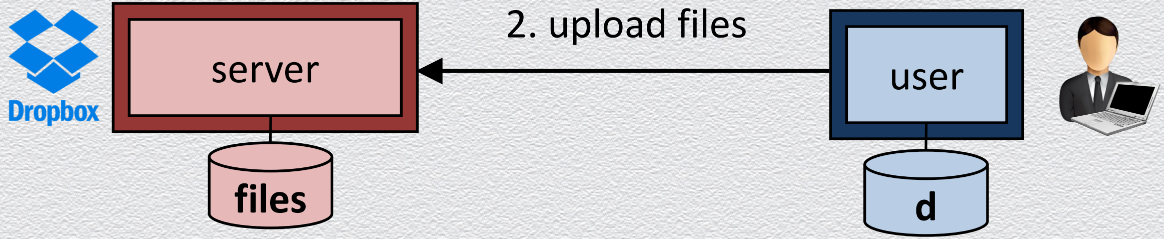
files

(1)

files = (F1, F2, …, F7, F8)

1. use CR hash function h to compute over all files a digest d, |d| << |F|

(2)



Dropbox

server

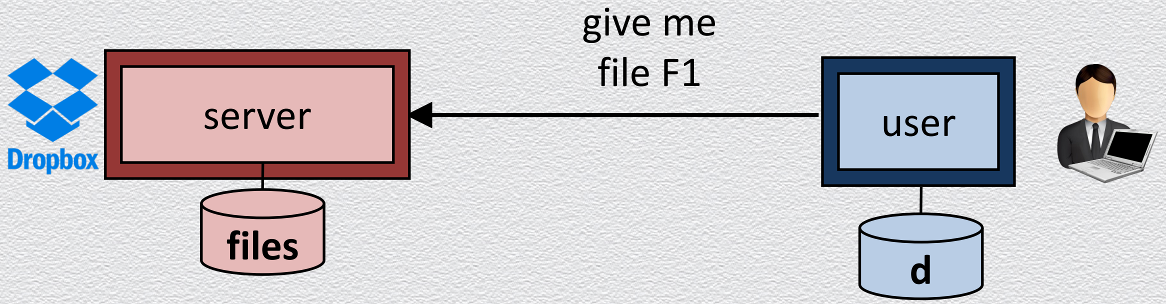
files

2. upload files

user

d

Example 6: Secure cloud storage – integrity checking



give me file F1

server

files

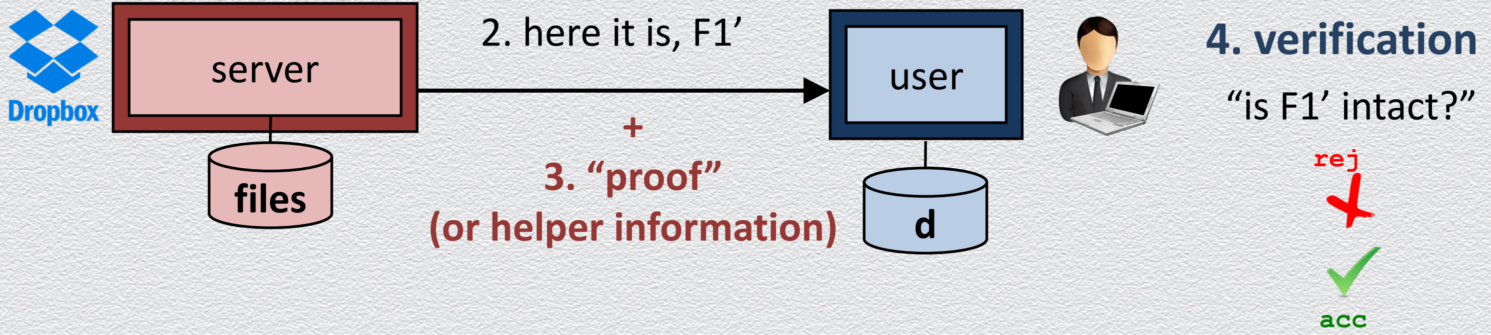
user

d

Dropbox

(3)

(4)



server

files

Dropbox

2. here it is, F1’

+

3. “proof”
(or helper information)

user

d

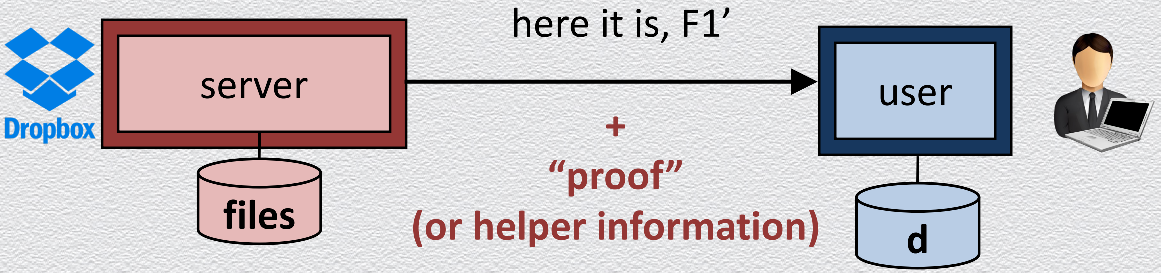
4. verification

“is F1’ intact?”

rej ✗

✓ acc

Example 6: Secure cloud storage – verification



Dropbox

server

files

here it is, F1’

+

“proof”

(or helper information)

user

d

(4)

verification

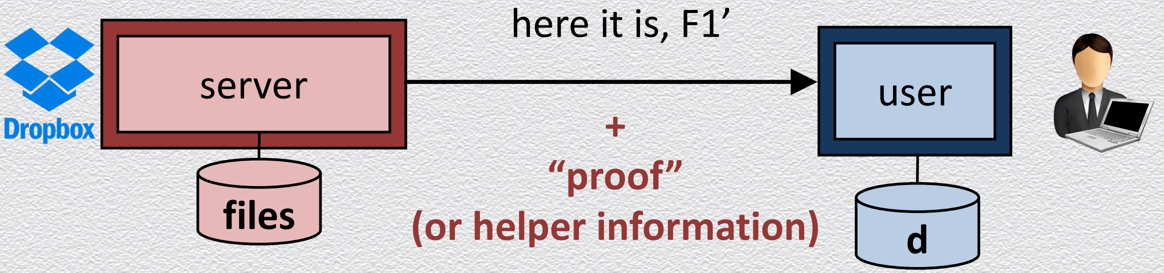
“is F1’ intact?”

- user has
  - authentic digest d (locally stored)
  - file F1’ (to be checked/verified as it can be altered)
  - **proof** (to help checking integrity, but it can be maliciously chosen)
- user locally verifies received answer
  - combine the file F1’ with the proof to re-compute candidate digest d’
  - check if d’ = d
  - if yes, then F1 is intact; otherwise tampering is detected!

rej ✗

✓ acc

Example 6: Data authentication via the Merkle tree



(4)

verification

“is F1’ intact?”

d’ ≠ d ✗ rej

d’ = d ✓ acc

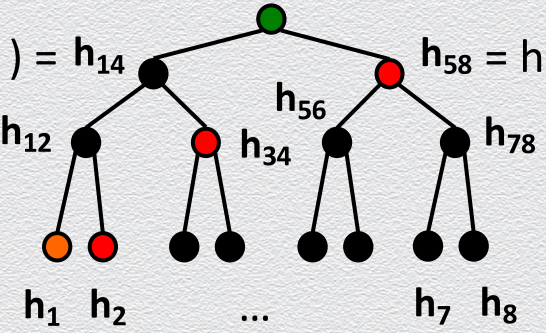
digest is the **green** root hash

d = h (h_{14} || h_{48})

h (h_{12} || h_{34}) = $\mathbf{h_{14}}$

$\mathbf{h_{58}}$ = h (h_{56} || h_{78})

h (h_1 || h_2) = $\mathbf{h_{12}}$



compute candidate d’ based on **answer** & **proof**

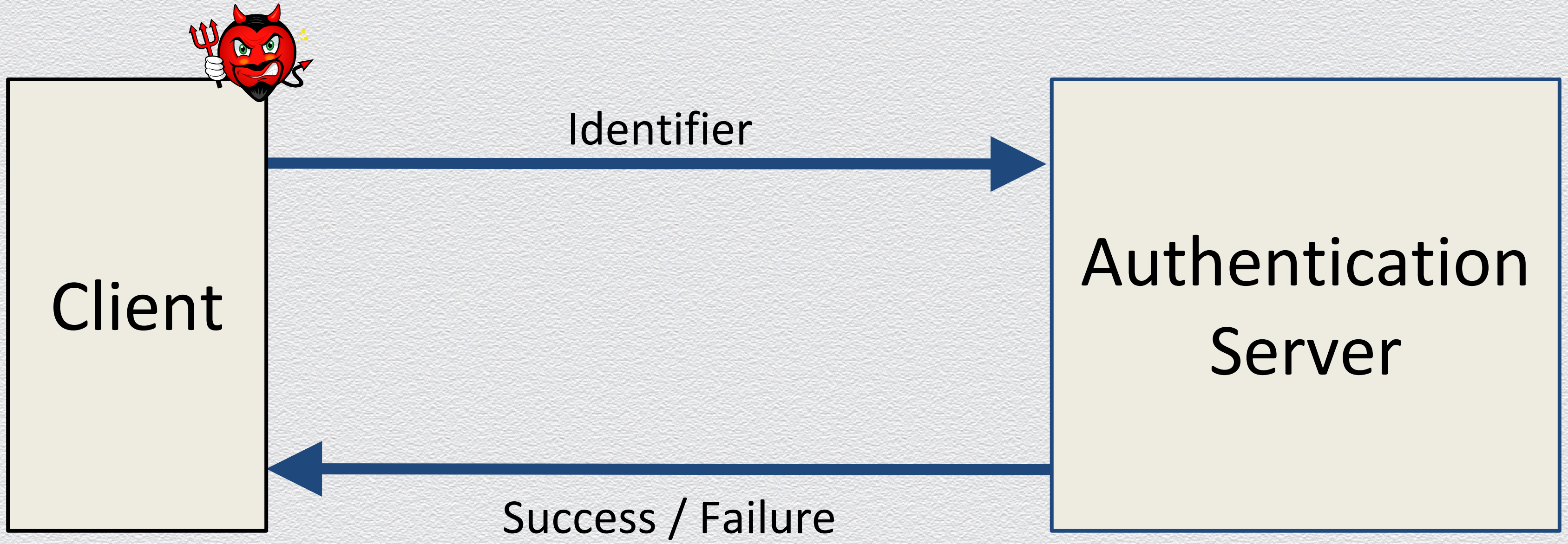
answer is **orange** hash

proof is **red** hash path

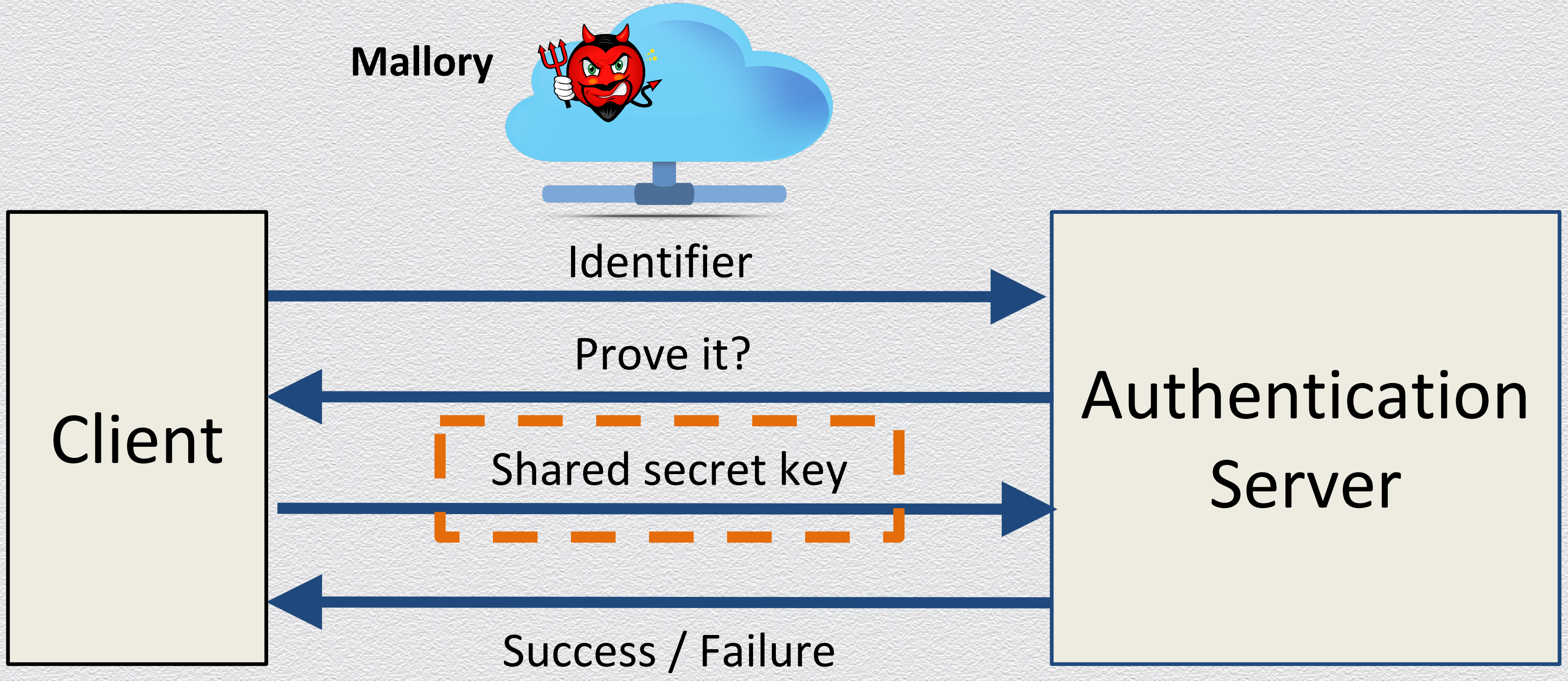
let $h_i = h(F_i)$, $1 \leq i \leq 8$

9.3 Other authentication protocols

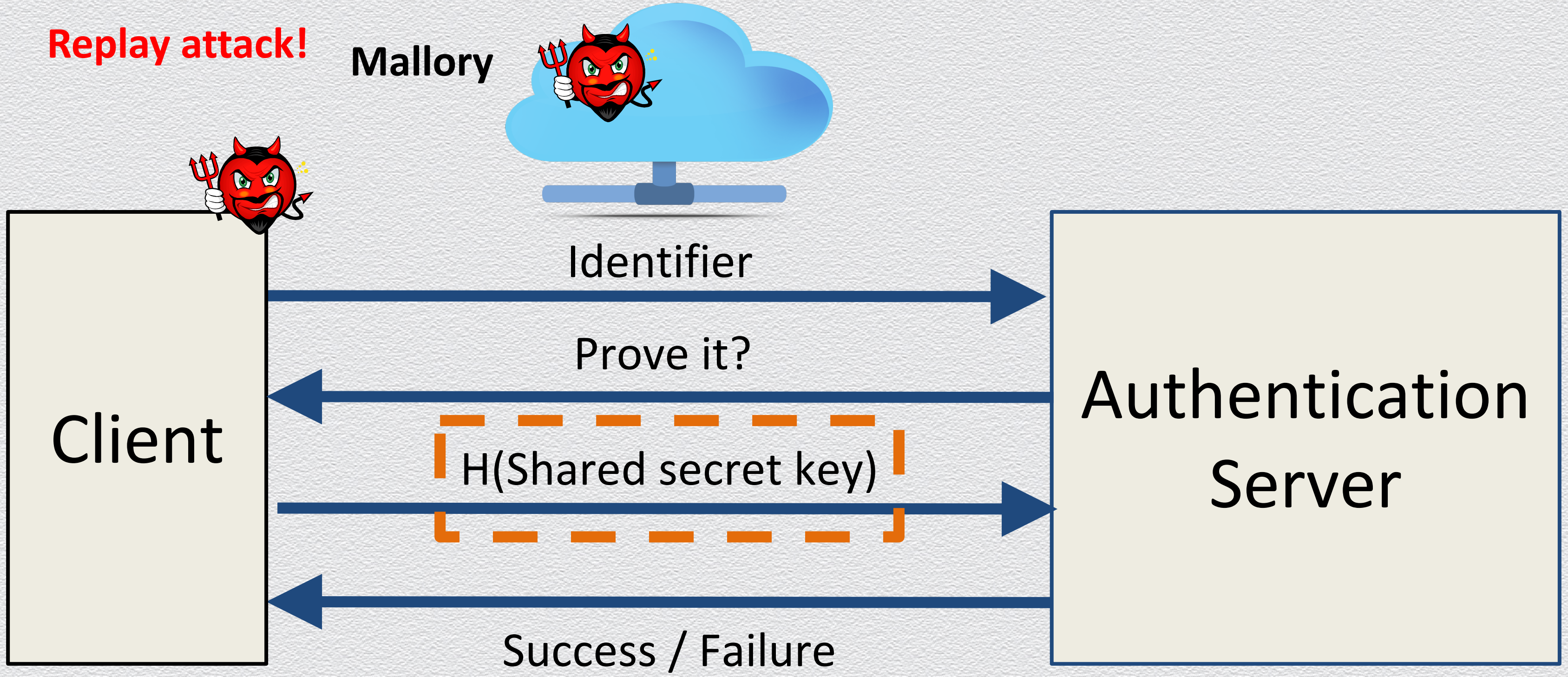
How to authenticate two systems?



But…


Mallory
Identifier
Prove it?
Shared secret key
Success / Failure
Client
Authentication Server

Even better method…



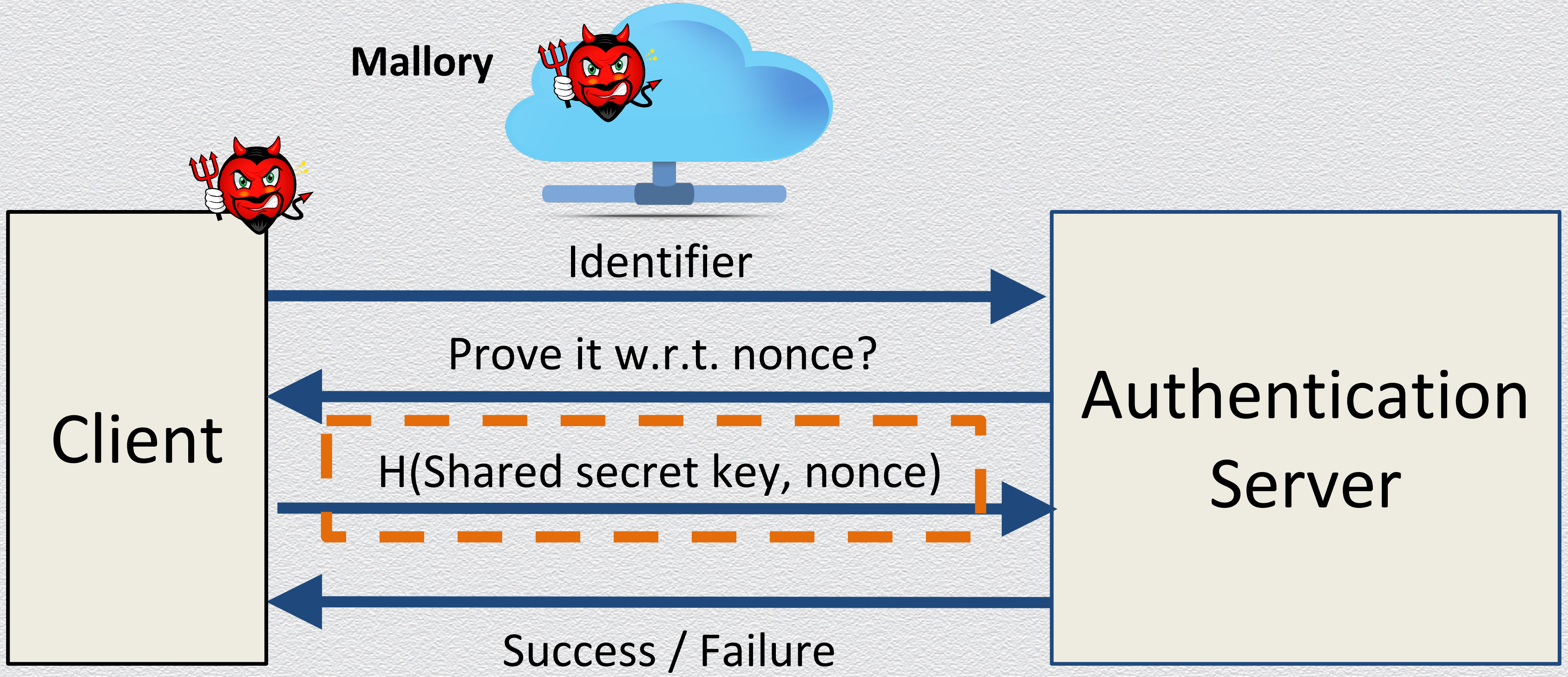
Challenge-response

- Use **challenge-response**, to prevent replay attack
    - Goal is to avoid the reuse of the same credential
- Suppose Client wants to authenticate Server
    - **Challenge** sent from Server to Client
- Challenge is chosen so that…
    - Replay is not possible
    - Only Client can provide the correct **Response**
    - Server can verify the response

Nonces

- To ensure “freshness”, can employ a **nonce**
    - Nonce == **n**umber used **once**
- What to use for nonces?
    - A **unique** random string
- What should the Client do with the nonce?
    - Transform the nonce using the shared secret
- How can the Server verify the response?
    - Server knows the shared secret and the nonce, so can check if the response is correct

Challenge-Response authentication method



Authentication protocols

- Challenge response mainly relies on nonce
- What if nonce wasn’t random?
- Harder to authenticate humans, more on that later…

9.4 Entropy

Entropy

- Amount of uncertainty in a situation

- Fair Coin Flip
  - Maximum uncertainty
- Biased Coin Flip
  - More bias → Less uncertainty

Entropy (cont.)

- Computers need a source of uncertainty (entropy) to generate random numbers.
  - Cryptographic keys.
  - Protocols that need coin flips.
- Which are sources of entropy in a computer?
  - Mouse and keyboard movements or thermal noise of processor.
  - Unix like operating systems use dev/random and dev/urandom as randomness collector

Random numbers in practice

- We need random numbers but...

  "*Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin.*" - John von Neumann

- Bootup state is predictable and entropy from the environment may be limited:
  - Temperature is relatively stable
  - Oftentimes the mouse/keyboard motions are predictable
- Routers often use network traffic
  - Eavesdroppers.
- Electromagnetic noise from an antenna outside of a building
- Radioactive decay of a 'pellet' of uranium
- Lava lamps…

Lava lamps

- Cloudflare company uses lava lamps as an entropy source

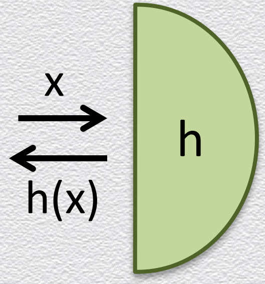
Provable security: Idealized models

- challenge in proving security of scheme S that employs scheme S’
  - no reasonable assumption on S’ or $\mathcal{A}$ can provide a security proof for S
- naïve approach: look for other schemes or use scheme S (if S’ looks “secure”)
- middle-ground approach: fully rigorous proof Vs. heuristic proofs
  - employ **idealized** models that **impose** assumptions on S’, $\mathcal{A}$
  - formally prove security of S in this idealized model
  - better than nothing…
- **canonical example**: employ the **random-oracle model** when using hashing
  - a cryptographic hash function h is treated as a **truly random** function

The random-oracle model

treats a cryptographic hash function h as a “black box” realizing a **random** function

- models h as a “secret service” that is publicly available for querying
  - anyone can provide input x and get output h(x)
  - nobody knows the exact functionality of the “box”
  - queries are assumed to be private
- interpretation of internal processing
  - if query x is new, then record and return a **random** value h(x) in the hash range
  - otherwise, answer **consistently** with previous queries on x



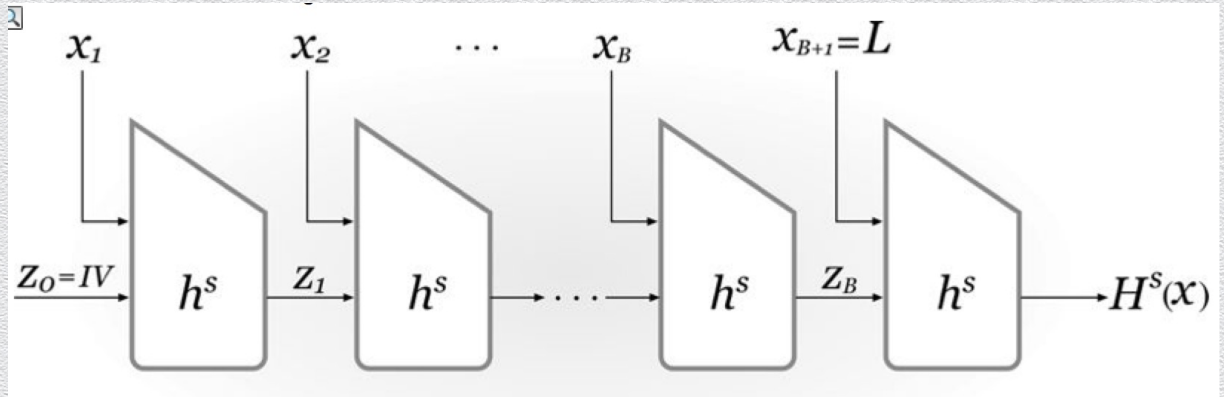
Random-oracle methodology

1. design & analyze using random oracle h; 2. instantiate h with specific function h'

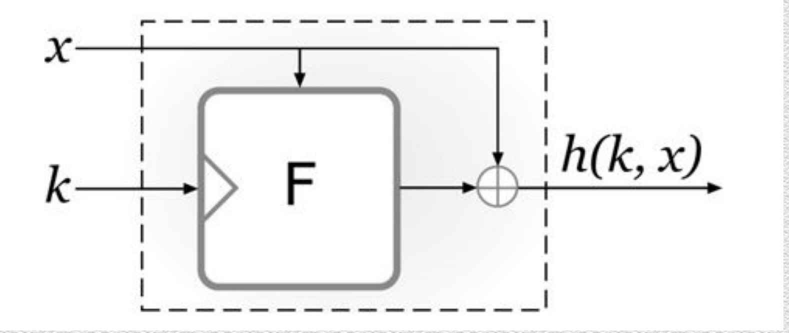
- how sound is such an approach? on-going debate in cryptographic community
- pros (proof in random-oracle model better than no proof at all)
  - leads to significantly **more efficient** (thus practical) schemes
  - design is **sound**, subject to limitations in instantiating h to h'
  - at present, only **contrived** attacks against schemes proved in this model are known
- cons (proofs in the standard model are preferable)
  - random oracles **may not exist** (cannot deterministically realize random functions)
  - real-life $\mathcal{A}$s see the code of h' (e.g., may find a shortcut for some hash values)
  - can construct scheme S, s.t. S is proven secure using h, but is insecure using h'
  - note: "h' is CR" Vs. "h' is a random oracle"

Recall: Constructing hash functions in practice

Merkle-Damgård transform



The Davies-Meyer scheme

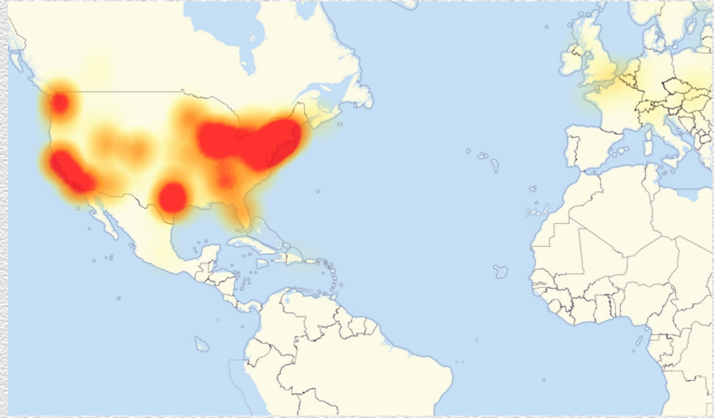


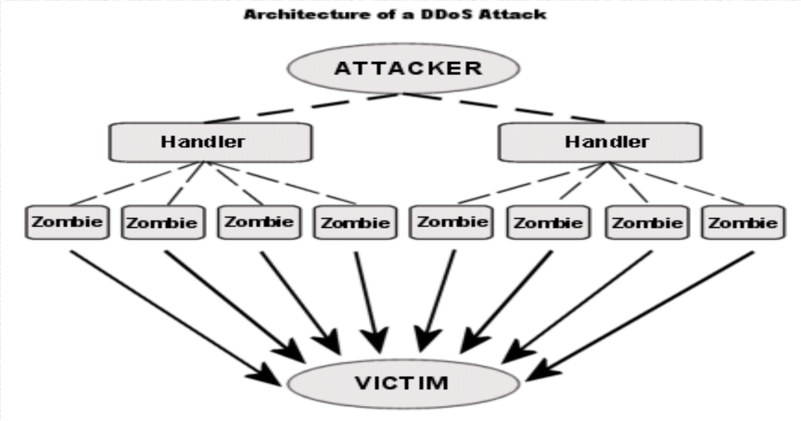
9.5* The Dyn DDoS attack

It’s unfair! – I had no class but couldn’t watch my Netflix series!

On October 21, 2016, a large-scale cyber was launched

- it affected globally the entire Internet but particularly hit U.S. east coast
- during most of the day, no one could access a long list of major Internet platforms and services, e.g., Netflix, CNN, Airbnb, PayPal, Zillow, …
- this was a **Distributed Denial-of-Service (DDoS)** attack





DoS: A threat (mainly) against availability

Which main security property does a Denial-of-Service (DoS) attack attempt to defeat?

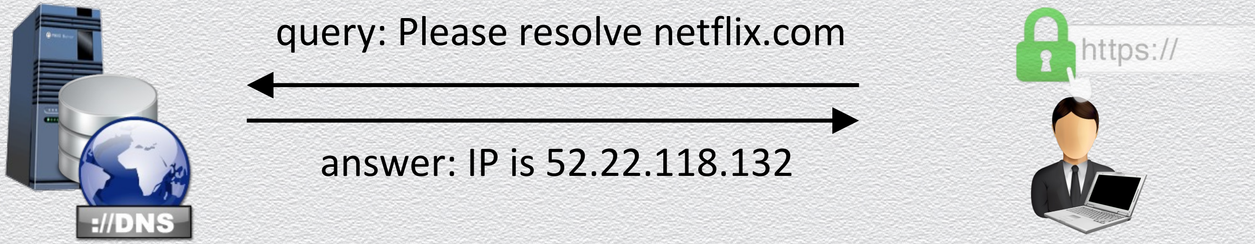
- availability; a user is denied access to authorized services or data
  - availability is concerned with preserving authorized access to assets
  - a DoS attack aims against this property; its name itself implies its main goal
- integrity & confidentiality; services or data are modified or accessed by an unauthorized user
  - elements of a DoS attack may include breaching the integrity or confidentiality of a system
  - but the end goal is disruption of a service or data flow; not the manipulation, fabrication or interception of data and services

DNS

The Domain Name Service (DNS) protocol

Resolving domain names to IP addresses

- when you type a URL in your Web browser, its IP address must be found
  - e.g., domain name “netflix.com” has IP address “52.22.118.132”
  - larger websites have multiple IP responses for redundancy to distributing load
- at the heart of Internet addressing is a protocol called DNS
  - a database translating Internet names to addresses



query: Please resolve netflix.com

answer: IP is 52.22.118.132

DNS name resolution is a critical asset – a target itself!

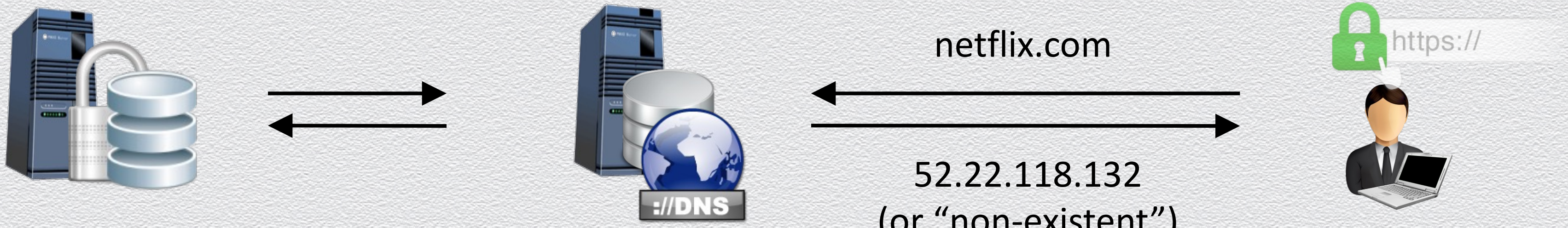
What main security properties must be preserved in such an important service?

- all properties in CIA triad are relevant!
- resolving domain names to IP addresses is a service that
  - must critically be available during all times – availability
    - or else your browser does not know how to connect to Netflix…
  - must critically be trustworthy – integrity
    - or else connections to malicious sites may occur (e.g., DNS-spoofing attacks)
  - must also protect database entries that are not queried – confidentiality
    - or else an attacker may find out about the structure of a target organization (e.g., zone-enumeration attacks)

Recursive name resolution: hierarchical search

Search is performed recursively and hierarchically across different type of DNS resolvers

- application-level (e.g., Web browser), OS-level (e.g., stub resolver): locally managed
- recursive DNS servers: query other resolvers and cache recent results



DNS entries:
<netflix.com, 52.22.118.132>

primary

subset of cached queried entries
(or information of other resolvers)

secondary

locally cached IP addresses
(at Web browser and OS)

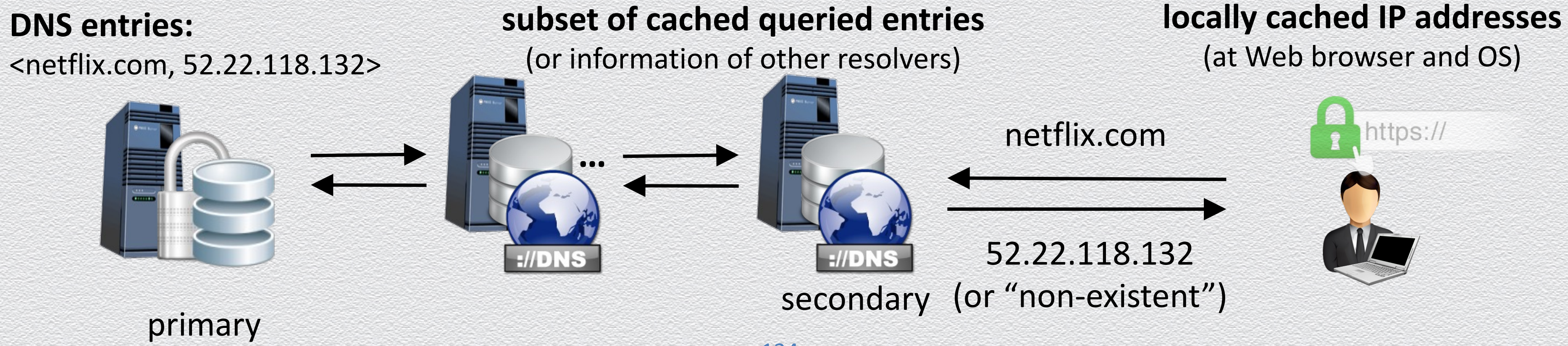
netflix.com

52.22.118.132
(or “non-existent”)

Recursive name resolution: hierarchical search

Search is performed recursively and hierarchically across different type of DNS resolvers

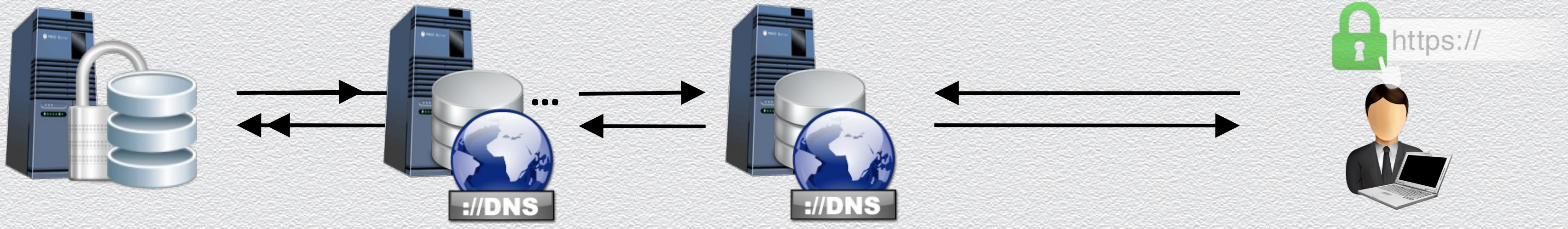
- application-level (e.g., Web browser), OS-level (e.g., stub resolver): locally managed
- recursive DNS servers: query other resolvers and cache recent results
- root name servers: refer to appropriate TLD (top-level domain) server
- TLD servers: control TLD zones such as .com, .org, .net, etc.


DNS entries:
<netflix.com, 52.22.118.132>
primary
subset of cached queried entries
(or information of other resolvers)
...
://DNS
://DNS
secondary
locally cached IP addresses
(at Web browser and OS)
https://
netflix.com
52.22.118.132
(or "non-existent")

Recursive name resolution: flexibility

Infrastructure allows for different configurations

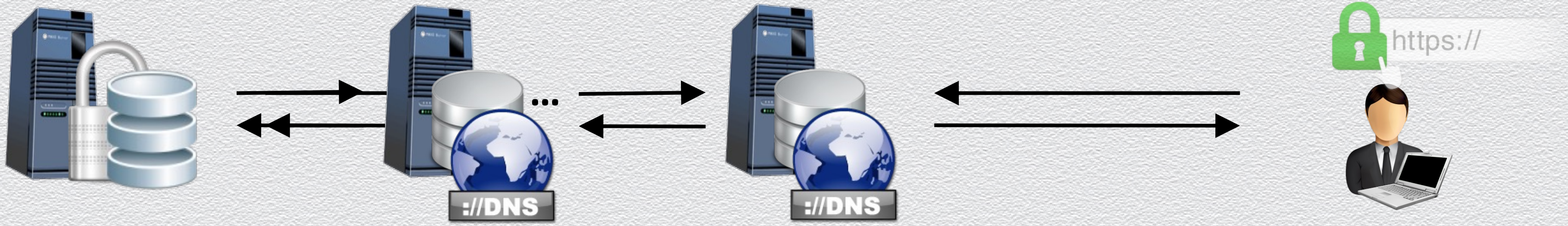
- authoritative-only servers: answer queries on zones they are responsible for
  - fast resolution, no forwarding, no cache
- caching / forwarding servers: answer queries on any public domain name
  - recursive search / request forwarding, caching for speed, first-hop resolvers
- primary / secondary servers: authoritative servers replicating DNS data of their domains
- public / private servers: control access to protected resources within an organization



Recursive name resolution: benefits

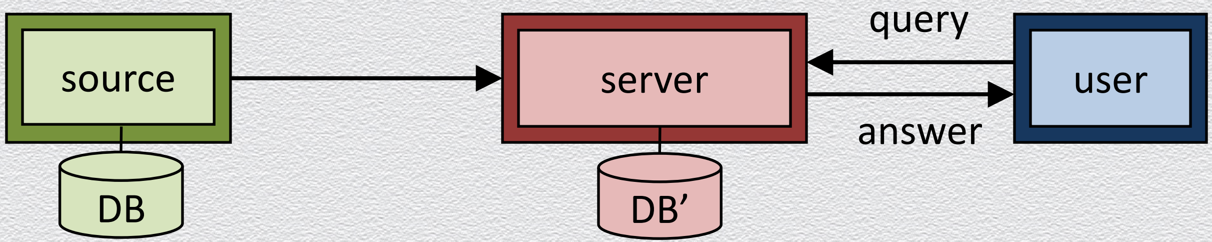
Why DNS uses non-authoritative name servers (that is, recursive resolution)?

- for more scalability & locality
  - high query loads can saturate the response capacity of primary servers
  - secondary do not have to store large volumes of DNS entries
  - cached recently queried domain names speed up searches due to locality of queries
- for added security / locality / scalability alone – not quite
  - e.g., non-authoritative name servers are untrusted and thus possibly compromised



DNS integrity: Protocols DNSSEC & NSEC

DNS as a (distributed) database-as-a-service



source

DB

server

DB’

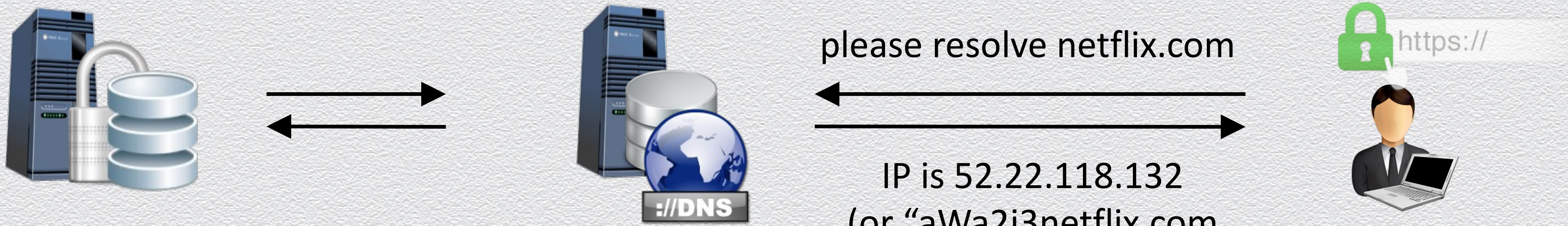
query

answer

user

DNS entries:
<netflix.com, 52.22.118.132>

subset of cached queried entries
(or information of other resolvers)



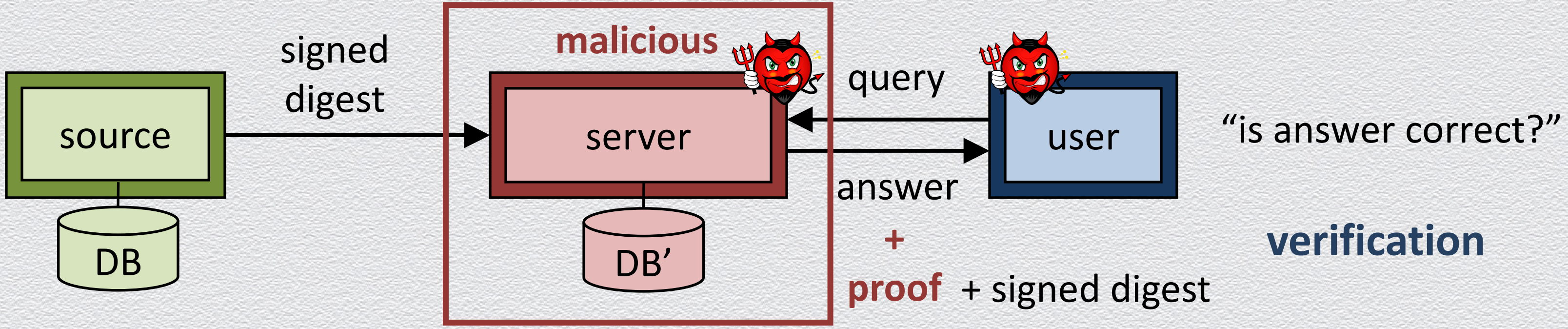
please resolve netflix.com

IP is 52.22.118.132
(or “aWa2j3netflix.com
is a non-existent domain”)

“primary”
name server

“secondary”
name server

A critical asset prone to attacks



signed digest

malicious

source

DB

server

DB’

query

user

answer

+

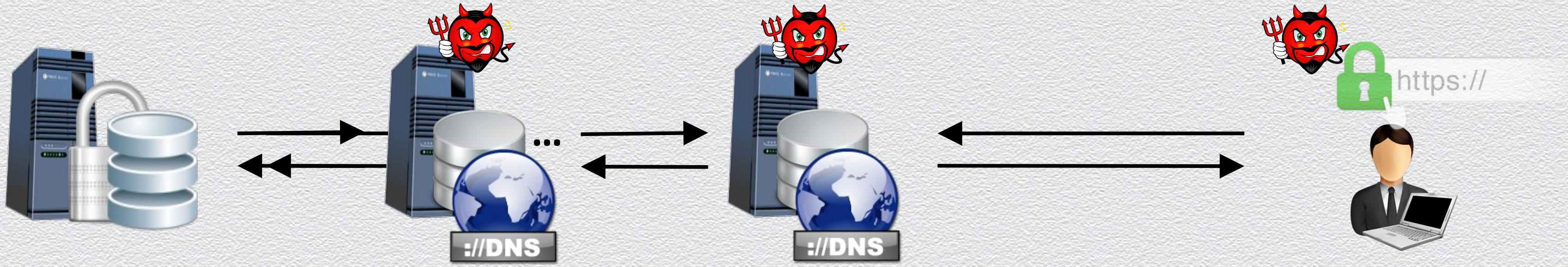
proof + signed digest

“is answer correct?”

verification

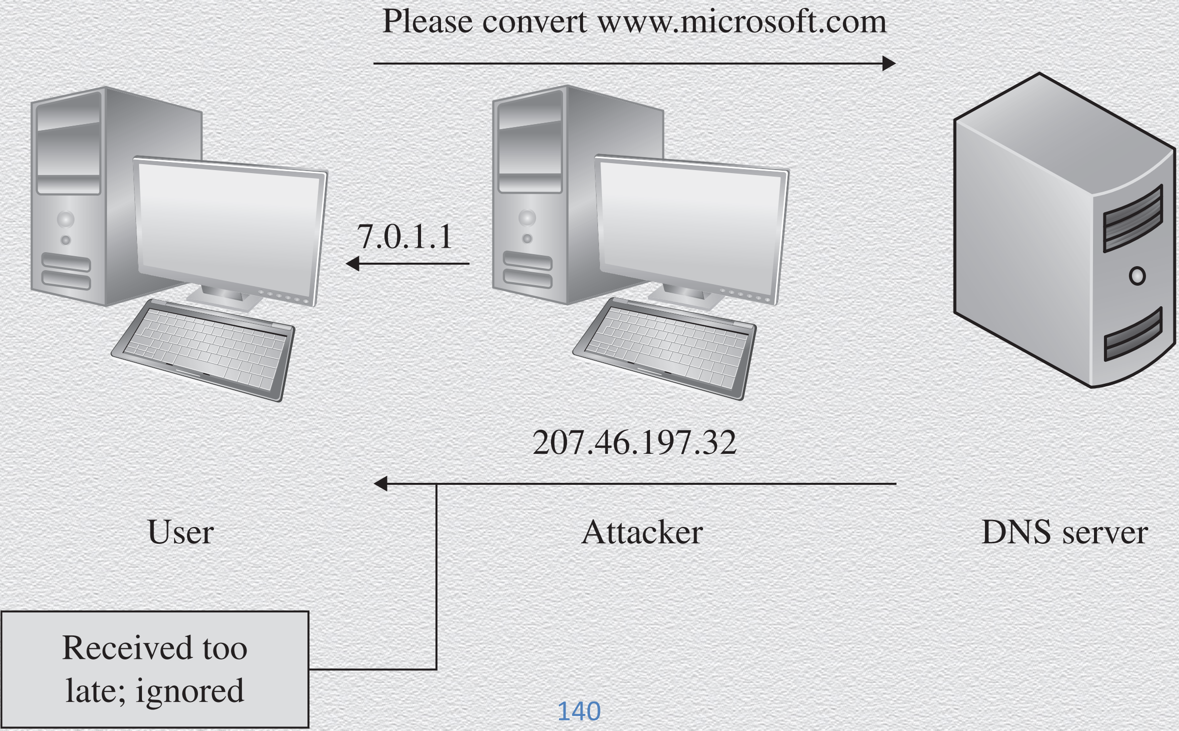
integrity

availability / confidentiality



DNS spoofing (or cache poisoning)

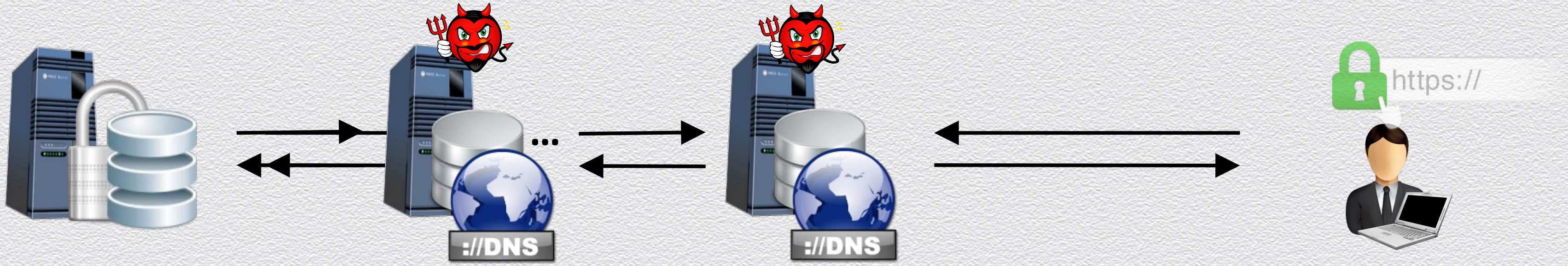
The attacker acts as the DNS server in order to redirect the user to malicious sites



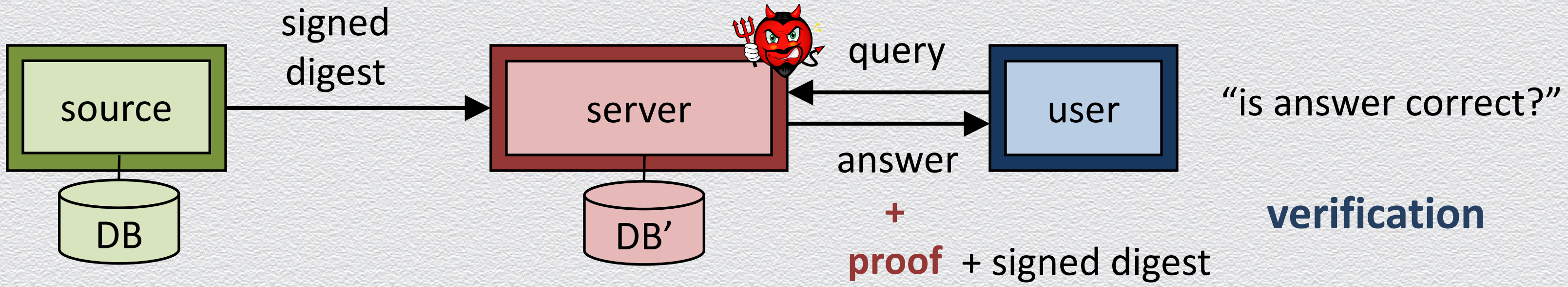
DNSSEC & NSEC

Security extension of DNS protocol to protect integrity of DNS data

- correct resolution, origin authentication, authenticated denial of existence
- specifications made by Internet Engineering Task Force (IETF) via RFCs
  - an RFC (request for comments) is a suggested solution under peer review
- challenges: backward-compatible, simplicity, confidentiality, who signs
  - NSEC (next secure record): extension that provides proofs of denial of existence



DNSSEC & NSEC: core idea



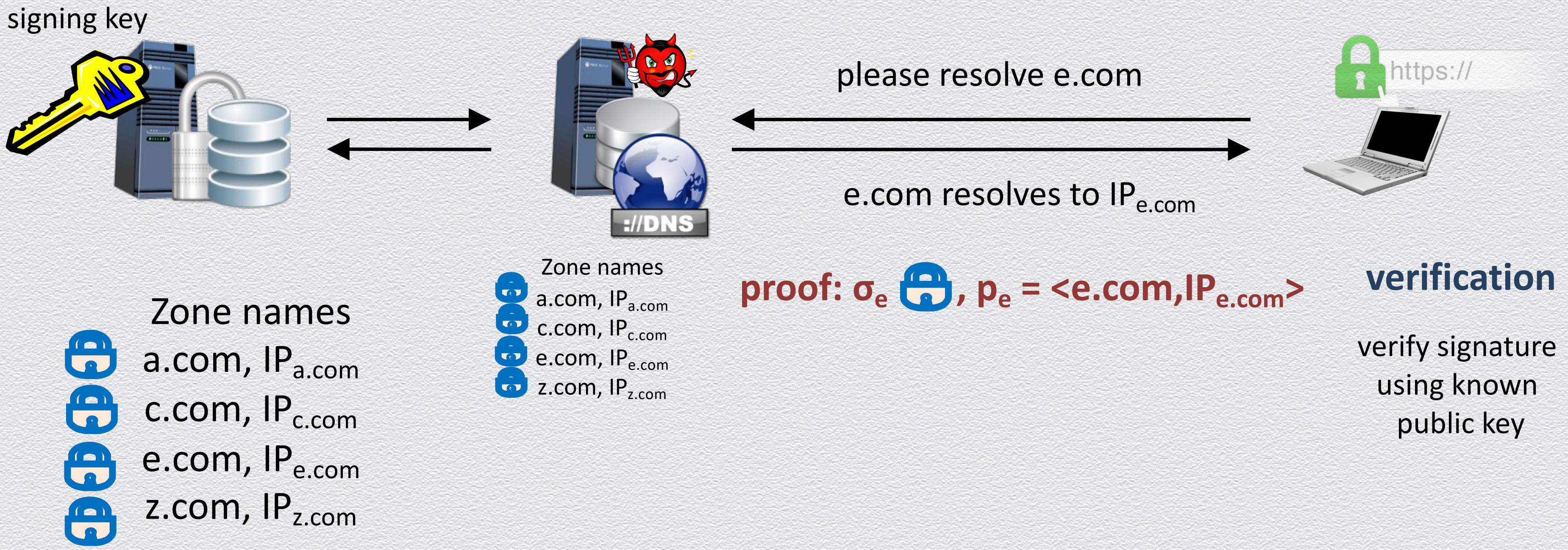
DNSSEC protocol: each DNS entry is pre-signed by primary name server

NSEC protocol:

- domain names are lexicographically ordered and then each pair of neighboring existing domain names is pre-signed by the primary name server
- non-existing names, e.g., aWa2j3netflix.com are proved by providing this pair “containing” missed query name, e.g., <awa.com, awb.com>

DNSSEC: example

Each entry <domain name, IP address> in the database is individually signed by a primary DNS server and uploaded to secondary DNS servers in signed form



signing key

Zone names

🔒 a.com, $IP_{a.com}$

🔒 c.com, $IP_{c.com}$

🔒 e.com, $IP_{e.com}$

🔒 z.com, $IP_{z.com}$

Zone names

🔒 a.com, $IP_{a.com}$

🔒 c.com, $IP_{c.com}$

🔒 e.com, $IP_{e.com}$

🔒 z.com, $IP_{z.com}$

please resolve e.com

e.com resolves to $IP_{e.com}$

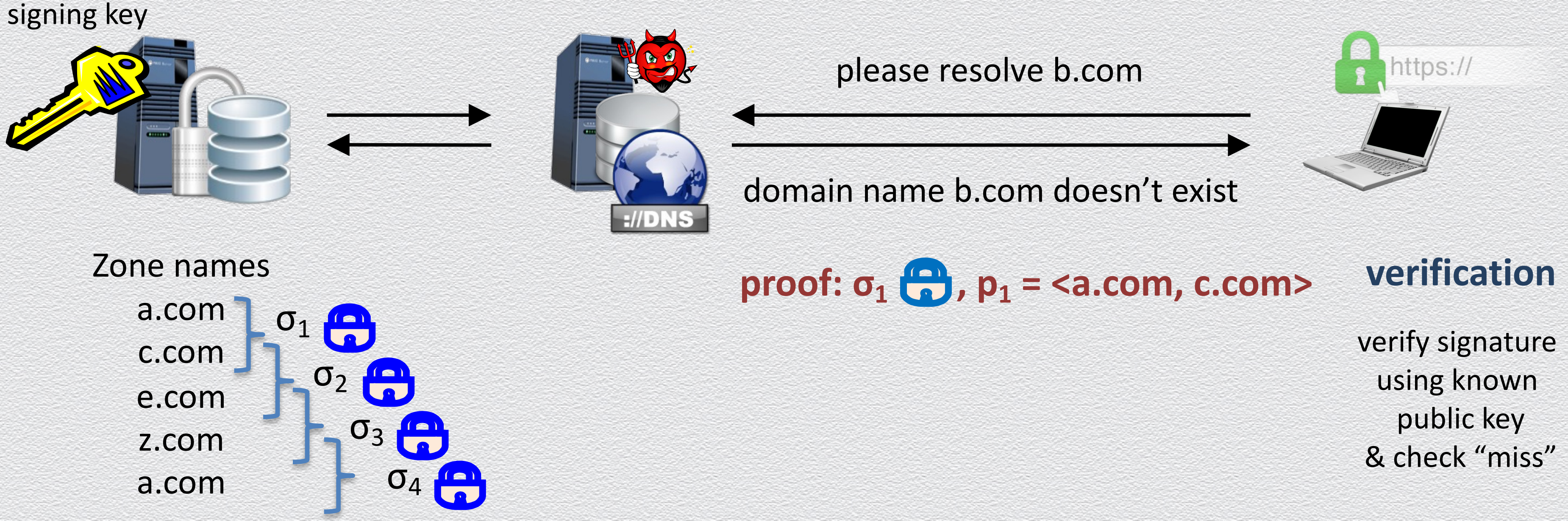
proof: σ_e 🔒, p_e = <e.com,$IP_{e.com}$>

verification

verify signature using known public key

NSEC: example

Additionally, pairs of consecutive (in alphabetical order) domain names are individually signed by a primary DNS server and uploaded to secondary DNS servers in signed form



signing key

please resolve b.com

domain name b.com doesn't exist

Zone names

a.com
c.com
e.com
z.com
a.com

σ_1 σ_2 σ_3 σ_4

proof: σ_1, p_1 = <a.com, c.com>

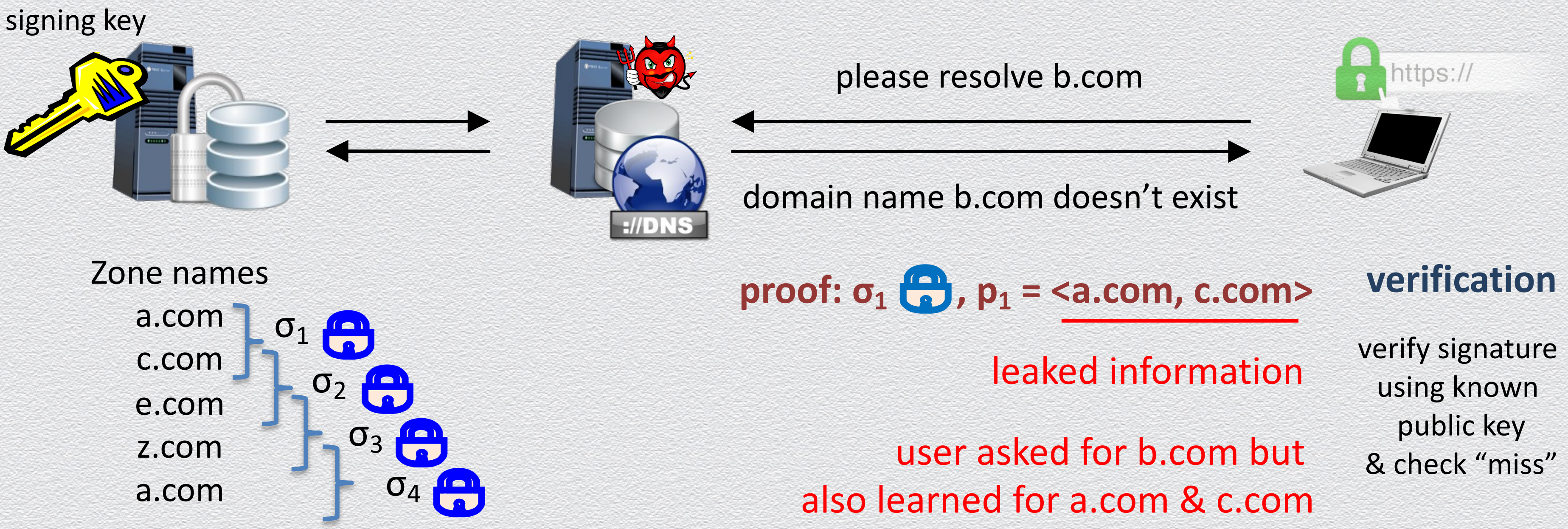
verification

verify signature
using known
public key
& check "miss"

NSEC vulnerability: Protocols NSEC3 & NSEC5

The problem

Proofs of non-existing names leak information about other unknown domain names



signing key

please resolve b.com

domain name b.com doesn’t exist

Zone names

a.com
c.com
e.com
z.com
a.com

σ_1 🔒 σ_2 🔒 σ_3 🔒 σ_4 🔒

proof: σ_1 🔒, p_1 = <a.com, c.com>

leaked information

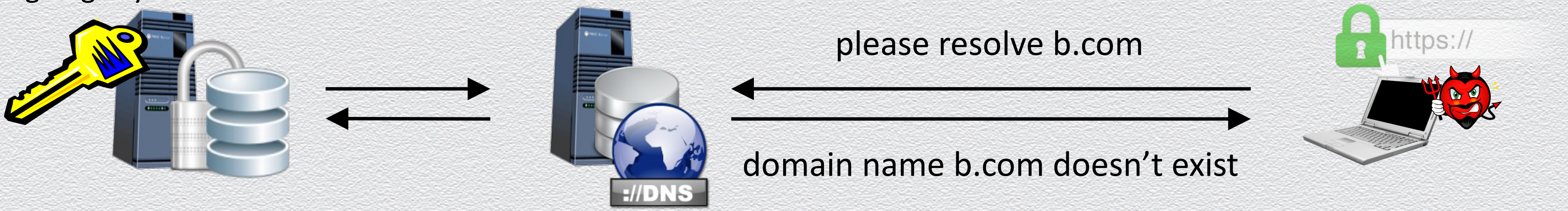
user asked for b.com but
also learned for a.com & c.com

verification

verify signature
using known
public key
& check “miss”

Zone enumeration attack: Main idea

An attacker can simply act as a “querier” to learn target organization’s network structure!

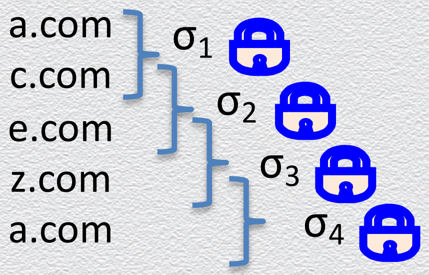


signing key

please resolve b.com

domain name b.com doesn’t exist

Zone names



a.com

c.com

e.com

z.com

a.com

σ_1 🔒

σ_2 🔒

σ_3 🔒

σ_4 🔒

proof: σ_1 🔒, p_1 = <u><a.com, c.com></u>

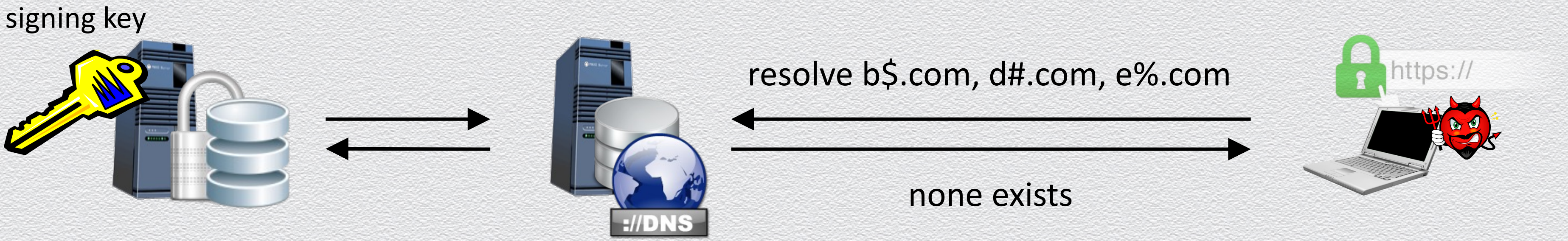
exploit the “leak-domain-names” vulnerability of NSEC to learn the domain names of an entire zone

verification

verify signature using known public key & check “miss”

Zone enumeration attack: Example

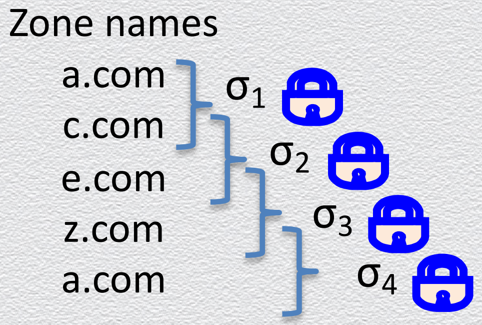
An attacker can simply act as a “querier” to learn target organization’s network structure!



signing key

resolve b$.com, d#.com, e%.com

none exists



Zone names

a.com
c.com
e.com
z.com
a.com

σ_1 🔒
σ_2 🔒
σ_3 🔒
σ_4 🔒

proof: σ_1 🔒, p_1 = <a.com, c.com>
proof: σ_2 🔒, p_2 = <c.com, e.com>

proof: σ_3 🔒, p_3 = <e.com, z.com>

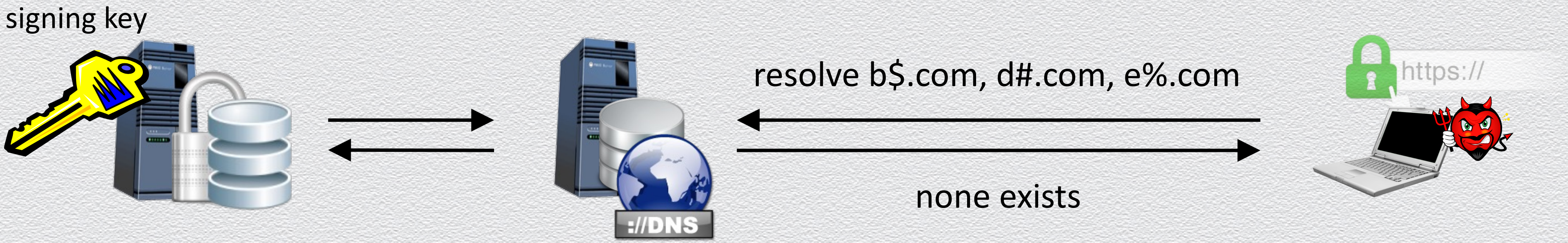
ask for non-existing names
to get all possible proofs

verification

verify signature
using known
public key
& check “miss”

Zone enumeration attack: Result

An attacker can simply act as a “querier” to learn target organization’s network structure!



Zone names

a.com

c.com

e.com

z.com

a.com

σ_1

σ_2

σ_3

σ_4

ask for non-existing names to get all possible proofs

This attack may expose private device names (e.g., IoT devices which can be toehold for other attacks) or reveal other private data that many registries may have legal obligations to protect

Zone names

a.com

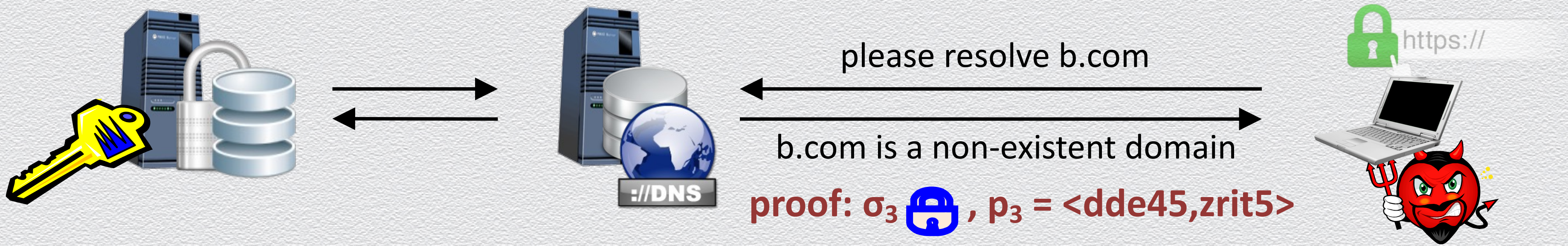
c.com

e.com

z.com

a.com

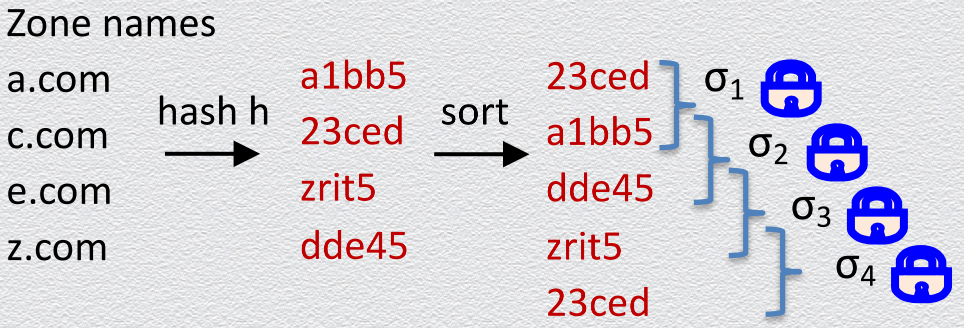
NSEC3: NSEC in the hash domain



please resolve b.com

b.com is a non-existent domain

proof: σ_3 🔒 , p_3 = <dde45,zrit5>



Zone names

a.com
c.com
e.com
z.com

hash h

a1bb5
23ced
zrit5
dde45

sort

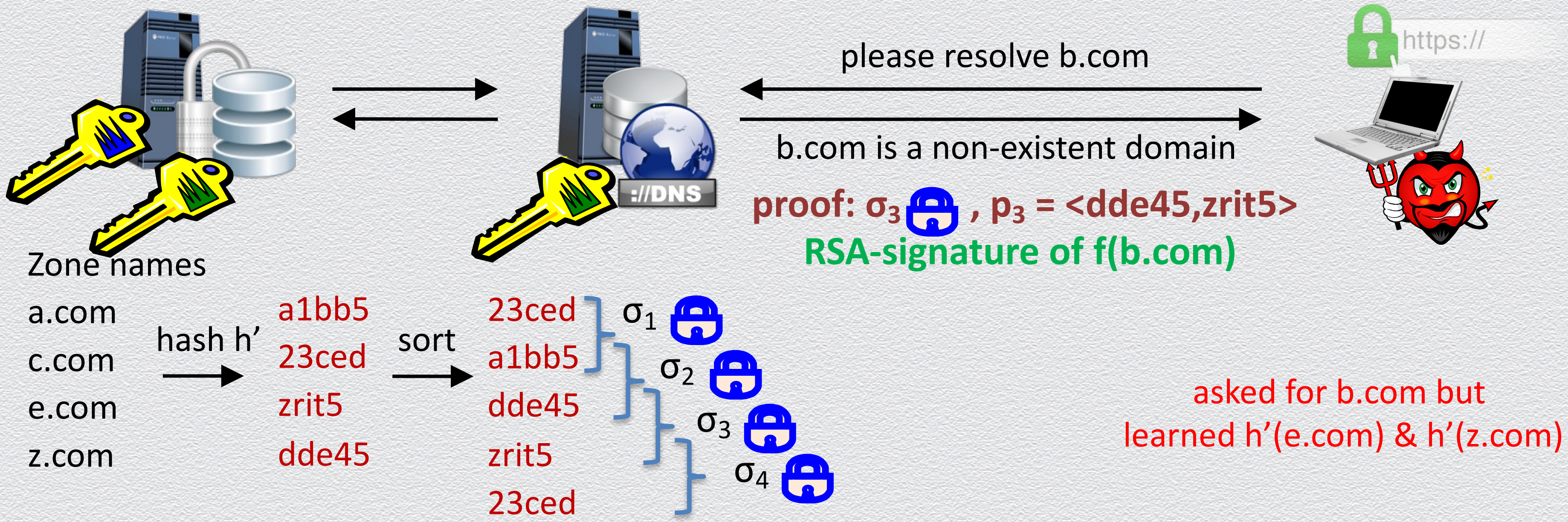
23ced
a1bb5
dde45
zrit5
23ced

σ_1 🔒
σ_2 🔒
σ_3 🔒
σ_4 🔒

asked for b.com but
learned h(e.com) & h(z.com)

h(b.com) = ntwo4
e.g., h is SHA-256

NSEC5: A secure solution



please resolve b.com

b.com is a non-existent domain

proof: σ_3 🔒 , p_3 = <dde45,zrit5>

RSA-signature of f(b.com)

Zone names

a.com

c.com

e.com

z.com

hash h' →

a1bb5

23ced

zrit5

dde45

sort →

23ced

a1bb5

dde45

zrit5

23ced

σ_1 🔒

σ_2 🔒

σ_3 🔒

σ_4 🔒

asked for b.com but
learned h'(e.com) & h'(z.com)

h'(b.com) = ntwo4

h: as in NSEC3

f: "message transformation" hash

h'(x) = h(RSA-Sign(🔑, f(x)))