Security and Privacy

CSEE 4121 — Computer Systems for Data Science | Spring 2026

Logistics

- Homework 4 will be out next week
- Homework 3 due April 14th
- Final exam: April 29th, 4:10 PM – 6:40 PM

Today's Agenda

- Common attacks on data systems
- Authentication and authorization
- Access control models in databases
- Data anonymization and de-identification
- Privacy-preserving data analysis

Recap — Cryptographic Primitives

Operation	What It Does	Keys Used	Provides	Example
Symmetric Encrypt	Transform plaintext into ciphertext; reversible with same key	Shared secret key (same for encrypt & decrypt)	Confidentiality	`AES-256`
Symmetric Decrypt	Recover plaintext from ciphertext	Same shared secret key	Confidentiality	`AES-256`
Asymmetric Encrypt	Encrypt with recipient's public key; only their private key decrypts	Recipient's public key	Confidentiality	`RSA`
Asymmetric Decrypt	Recover plaintext using private key	Recipient's private key (secret)	Confidentiality	`RSA`
Hash	Map arbitrary input to fixed-size digest; one-way (not reversible)	None (keyless)	Integrity	`SHA-256`
Digital Sign	Hash message, then encrypt hash with sender's private key	Sender's private key (secret)	Integrity + Authenticity	`RSA / ECDSA`
Verify Signature	Decrypt signature with sender's public key; compare to hash of message	Sender's public key	Integrity + Authenticity	`RSA / ECDSA`
Key Exchange	Two parties derive a shared secret over an insecure channel	Each party's ephemeral key pair	Confidentiality (enables symmetric encryption)	`Diffie-Hellman`

Symmetric = fast, used for bulk data | Asymmetric = slower, used for key exchange & signatures | Hashing = one-way, used for integrity checks

Recap — What We Covered Last Time

- Four security goals: Confidentiality, Integrity, Availability, Authenticity
- Cryptographic primitives:
  - Symmetric encryption (AES)
  - Asymmetric encryption (RSA)
  - Digital signatures
  - Diffie-Hellman key exchange
  - Certificates and Public Key Infrastructure
- Compliance and frameworks: HIPAA, GDPR, PCI, FERPA

Why Data Scientists Need to Care

- You will build and query systems that store personally identifiable information (PII)
- You will be asked to "anonymize and share" datasets
- You will use APIs that require authentication and access tokens
- Data breaches cost companies an average of ~$4.5M per incident (IBM)
- GDPR fines can reach up to 4% of global annual revenue
- **Ignorance is not a defense — engineers and data scientists share responsibility**

Common Attacks on Data Systems

Some categories of attacks

- Injection attacks — trick the system into executing attacker-controlled code
- Credential-based attacks — compromise user identities to gain unauthorized access
- Denial of service — overwhelm the system to destroy availability
- Supply chain attacks — compromise a dependency to infiltrate upstream systems
- Social engineering — manipulate humans rather than systems (phishing)
- Exploit software vulnerabilities – Heartbleed, Meltdown, Spectre, general buffer overflow, outdated cryptography etc.
- Intentional backdoors – Designed crypto algorithms with builtin vulnerability only known to some actors

SQL Injection

Consider a simple login form that checks credentials against a database

```
# Backend code (Python)
query = "SELECT * FROM users "
        "WHERE username = '" + user_input + "' "
        "AND password = '" + pass_input + "'"
db.execute(query)

# Honest user enters:  username=alice  password=secret123
# Resulting query:
SELECT * FROM users
  WHERE username = 'alice'
  AND password = 'secret123'

# This works fine for honest inputs...
```

SQL Injection

What happens when the user input contains SQL syntax?

```
# Attacker enters:  username = ' OR 1=1 --

# Resulting query:
SELECT * FROM users
  WHERE username = '' OR 1=1 --' AND password = ''

# OR 1=1  →  always true  →  returns ALL rows
# --      →  comments out the rest (password check)

# The attacker is now logged in.
# They never needed to know any password.
```

SQL Injection — It Gets Worse

- SQL injection isn't limited to bypassing login. The attacker can also:
- Exfiltrate data:
  - ' UNION SELECT credit_card, ssn FROM customers --
- Modify data:
  - '; UPDATE accounts SET balance = 1000000 WHERE user = 'attacker' --
- Delete data:
  - '; DROP TABLE users -- (the classic "Bobby Tables")
- Execute system commands (on some database configurations)
- **A single unprotected input field can compromise the entire database**

SQL Injection — The Fix

NEVER construct SQL queries by concatenating user input

```
# PARAMETERIZED QUERIES (prepared statements)
query = "SELECT * FROM users WHERE username = %s AND password = %s"
db.execute(query, (user_input, pass_input))

# The database treats parameters as DATA, never as code
# The string ' OR 1=1 -- is searched for literally
# as a username — it won't match anyone

# Additional defenses:
#   - Input validation and sanitization
#   - Least privilege for DB accounts (no DROP TABLE)
#   - Web application firewalls (WAFs)
```

Beyond SQL Injection

- NoSQL injection: same idea applied to MongoDB, Cassandra, etc.
  - Attacker sends {"username": {"$gt": ""}} to bypass auth in MongoDB
- Command injection: user input passed to a shell command
  - os.system("ping " + user_input) → attacker inputs: google.com; cat /etc/passwd
- Log injection: attacker injects fake log entries to cover tracks
- Cross-Site Scripting (XSS): inject malicious JavaScript into web pages
  - Stored XSS: attacker posts <script>steal_cookies()</script> in a comment
  - Reflected XSS: attacker crafts a URL with script in query parameter
  - Impact: steal session cookies, hijack accounts, redirect to phishing sites
- **Common thread: any time untrusted input is mixed with code, injection is possible**

Credential-Based Attacks

- Credential stuffing:
  - Attackers take leaked username/password pairs from one breach and try them elsewhere
  - Works because people reuse passwords across services
  - Automated tools can try millions of credentials per hour
- Brute force: systematically try every possible password
  - Mitigated by rate limiting, account lockout, CAPTCHAs
- Dictionary attacks: try common passwords (password123, qwerty, etc.)
  - Mitigated by password complexity requirements

Phishing

- Social engineering: trick the user into giving up credentials voluntarily
- Attacker sends email that looks like a legitimate service (bank, university, employer)
- Email contains a link to a fake login page that captures credentials
- Spear phishing: targeted at a specific individual using personal details
  - E.g., "Hi [name], [your manager] asked me to share this document with you"
- **Extremely effective — most major breaches start with phishing**
- Defense: MFA (even if password is stolen, attacker doesn't have second factor), security awareness training, email filtering

Denial of Service (DoS)

- Goal: make a system unavailable to legitimate users — targets the "A" in CIA
- Volumetric attacks: flood the network with traffic to saturate bandwidth
  - E.g., UDP flood, DNS amplification
- Protocol attacks: exploit weaknesses in network protocols
  - E.g., SYN flood — send millions of TCP SYN packets, never complete handshake, exhaust connection table
- Application-layer attacks: send requests expensive for the server to process
  - E.g., complex SQL queries, large file uploads, expensive API calls
  - Harder to distinguish from legitimate traffic
- DNS KeyTrap

DDoS — Distributed Denial of Service

- Same idea as DoS, but traffic comes from thousands or millions of sources
- Botnets: networks of compromised devices (often IoT — cameras, routers)
  - Mirai botnet (2016): ~600K compromised IoT devices
  - Took down DNS provider Dyn → outages at Twitter, Netflix, Reddit, GitHub
  - Read more: https://www.thousandeyes.com/blog/dyn-dns-ddos-attack
- Much harder to defend — you can't just block one IP
- Mitigations:
  - CDNs absorb traffic at the edge (Cloudflare, Akamai)
  - Rate limiting per IP / per region
  - Traffic scrubbing services
  - Auto-scaling (but attacker wins economically — or do they?)

Authentication & Authorization

Authentication vs. Authorization

Authentication (AuthN)

- "Who are you?"
- Verifying identity
- Examples: login page, fingerprint scan, SSH key
- Happens first

Authorization (AuthZ)

- "What are you allowed to do?"
- Verifying permissions
- Examples: can this user read this table?
- Happens after authentication

AuthN and AuthZ Are Independent

- You can be authenticated but unauthorized
  - "I know you're an intern, but you can't access production"
- A system can allow unauthenticated users to be authorized
  - "The S3 bucket is public — anyone can read it without logging in”
  - Not always intentional (vulnerability!)
- Many breaches exploit this gap between authentication and authorization

Factors of Authentication

- Something you know: password, PIN, security question
  - Weakest — can be guessed, phished, stolen, shared
- Something you have: phone (SMS/TOTP codes), hardware key (YubiKey)
  - Stronger — attacker needs physical access
- Something you are: fingerprint, face recognition, iris scan
  - Convenient but can't be changed if compromised
- **Multi-factor authentication (MFA): combine two or more factors**
  - Vastly stronger than any single factor
  - MFA blocks ~99% of automated credential attacks (Microsoft)

Passwords and Their Problems

- Passwords are the most common auth mechanism and also the weakest
- Why passwords are broken:
  - People choose weak passwords (123456, password still in top 10 every year)
  - People reuse passwords across services
  - Phishing is highly effective at stealing them
  - Even strong passwords can be keylogged
- Despite their problems, passwords aren't going away soon
- **Goal: make the password layer as robust as possible, and supplement with other factors**

How to Store Passwords

Use bcrypt, scrypt, or argon2 — intentionally slow hashing

```
# NEVER store plaintext!
passwords_table = {"alice": "secret123"}  # Terrible

# Hashing: store hash(password)
# Problem: identical passwords → identical hashes (rainbow tables)

# Salted hashing: store hash(password + salt)
import bcrypt
hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt(rounds=12))
# Each user gets unique random salt
# Same password → different hash for different users

# bcrypt at cost 12 ≈ ~250ms per hash
# Legitimate: 1 login = negligible delay
# Attacker: billions of tries = centuries
```

Session Management: Cookies vs. Tokens

Session Cookies

- Server creates session ID, stores in session store
- Sends cookie to browser
- Browser sends cookie with every request
- State lives on the server
- Easy to revoke (delete session)
- Requires server-side storage

Token-Based (JWTs)

- Server generates signed token with user info
- Client includes token in Authorization header
- Server verifies signature — no session store
- Stateless: great for distributed systems
- Harder to revoke (valid until expiration)
- Self-contained — no server-side state

What's Inside a JWT?

Self-contained tokens for stateless authentication

```
# Three parts, base64-encoded, separated by dots:
# header.payload.signature

# Header:
{"alg": "HS256", "typ": "JWT"}

# Payload (claims):
{"sub": "alice", "role": "analyst", "exp": 1714000000}

# Signature:
HMAC(base64(header) + "." + base64(payload), secret_key)

# Anyone can READ the payload (it's just base64)
# But nobody can MODIFY it without the secret key
# JWTs are SIGNED, not ENCRYPTED by default
```

API Keys

- Common authentication for programmatic access to services
- A long random string: sk-abc123xyz789...
- Included in HTTP headers: Authorization: Bearer sk-abc123xyz789...
- You'll encounter these everywhere: OpenAI, Google Cloud, AWS, Stripe
- Best practices:
  - Never commit API keys to Git repositories
  - Use environment variables or secrets managers
  - Rotate keys periodically
  - Use different keys for dev and production
  - Apply least-privilege scoping (read-only key for analytics)
- Risk: bots continuously scan GitHub for exposed keys

Single Sign-On (SSO)

- One authentication event grants access to multiple applications
- Example: log into Okta once → access Slack, Jira, AWS, Snowflake without re-authenticating
- How it works: a central Identity Provider (IdP) authenticates the user; each app trusts the IdP
- Protocols: SAML (XML-based, enterprise) and OIDC (built on OAuth 2.0, more modern)
- Why companies use SSO:
  - Centralized control: onboard/offboard in one place
  - Easier revocation: disable one account → all access revoked
  - Better UX: fewer passwords to remember
  - Easier to enforce MFA across all services

Delegated authorization – OAuth

- Scenario: you want a third-party analytics tool to access your Google Drive data
- Without OAuth: give the app your Google password
  - App has full access to your entire Google account
  - Can't revoke without changing your password
  - Trusting the app to store your password securely
- With OAuth: grant limited, revocable access without sharing your password
  - App gets a scoped token — only the permissions you consent to
  - You can revoke access at any time
  - Your password never touches the application
- OAuth is the protocol behind every "Sign in with Google/GitHub" button

OAuth 2.0 — The Flow

1. User clicks "Sign in with Google" on the Application
2. Application redirects user to Google (Authorization Server)
3. User authenticates with Google and consents to requested permissions

   "This app wants to read your Google Drive files"
4. Google redirects back to Application with an authorization code
5. Application exchanges code for an access token (server-to-server)
6. Application uses access token to call Google's API on behalf of user

The user's password never touches the application

The token is scoped, time-limited, and revocable

MFA in Practice

- SMS codes: server sends code via text message
  - Weakest form: vulnerable to SIM swapping attacks
  - Still better than no MFA
- Authenticator apps (Google Authenticator, Authy): time-based one-time passwords (TOTP)
  - App and server share a secret seed, generate same 6-digit code every 30 seconds
  - Not vulnerable to SIM swapping
- Hardware security keys (YubiKey): physical device, cryptographic challenge-response
  - Strongest: resistant to phishing (key checks the domain)
  - Google: zero successful phishing attacks on 85K+ employees after requiring hardware keys
- Passkeys (emerging): FIDO2/WebAuthn, public-key crypto on your device
  - Replaces passwords entirely — supported by Apple, Google, Microsoft

Service-to-Service Authentication

- Not all authentication involves humans — in data pipelines, it's machines authenticating to machines
- Service accounts: dedicated non-human accounts
  - E.g., a GCP service account with a JSON key file
- IAM roles (cloud): assign permissions to compute resources
  - EC2 instance or Lambda inherits permissions from its role
  - No API keys to manage or leak
- Mutual TLS (mTLS): both client and server present certificates
  - Used in zero-trust architectures, service meshes (Istio, Linkerd)
- **Credential leaks common — hardcoded in scripts, committed to repos, in plaintext config files**

Equifax Breach — Overview and Timeline

- Equifax: one of the three major U.S. credit reporting agencies
- Holds financial data on ~800 million individuals worldwide
- **In 2017, attackers breached Equifax and stole personal data of ~147 million Americans**
- Stolen data included: Social Security numbers, birth dates and addresses, driver's license numbers, credit card numbers

- Full report: https://oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf
- March 7, 2017: Apache Struts vulnerability disclosed, patch released
  - Apache Struts: web framework Equifax used for online dispute portal
- March 9: Equifax internal security team notified to patch
- March 15: Equifax runs a scan but fails to identify the vulnerable system
- **May 13: Attackers exploit the unpatched vulnerability, gain initial access**
- May 13 – July 29 (76 days): Attackers move laterally, exfiltrating data
- July 29: Equifax notices suspicious network traffic
- September 7: Equifax publicly discloses the breach

Equifax — What Went Wrong

- Failure 1: Unpatched software
  - Vulnerability had a patch for 2+ months before exploitation
- Failure 2: Failed vulnerability scan
  - Scan didn't cover all internet-facing systems
- Failure 3: Expired SSL certificate on inspection tool
  - Internal device for inspecting encrypted traffic had expired cert for 19 months
  - Data exfiltration over encrypted channels went undetected for 76 days
- Failure 4: Flat network security architecture
  - No segmentation — once inside, attackers moved freely
- Failure 5: Unencrypted PII
  - Stolen data stored without encryption at rest
- Failure 6: Plaintext credentials in config files
  - Attackers found DB passwords in plain text

Equifax — Lessons for Data Systems

- Patch promptly — especially internet-facing systems
- Encrypt data at rest — breached but encrypted data is still protected
- Network segmentation — limit lateral movement
- Defense in depth — no single control is sufficient
- Monitor and alert — expired certs, anomalous traffic, unusual query volumes
- Credential management — use secrets managers (Vault, AWS Secrets Manager)

Access Control Models in Databases

The Principle of Least Privilege

- **Every user and process should have the minimum permissions needed to do their job**
- Nothing more, nothing less
- Why this matters:
  - Limits blast radius of a breach (compromised account only accesses what it was allowed)
  - Limits accidental damage (analyst can't DROP TABLE)
  - Meets compliance requirements (HIPAA, GDPR require access controls)
- Why it's hard in practice:
  - Convenience pressure: "just give me admin access"
  - Permission creep: users accumulate permissions, old ones never revoked
  - Example: the intern who still has broad access 6 months later

Role-Based Access Control (RBAC)

- The most common access control model in databases and cloud systems
- Three concepts:
  - Users: individual accounts (alice, bob, spark_etl_job)
  - Roles: named sets of permissions (analyst, engineer, admin)
  - Permissions: specific actions on specific resources (SELECT on table X)
- Users are assigned to roles; roles are granted permissions
- Benefits:
  - New analyst joins → assign analyst role → correct permissions immediately
  - Analyst leaves → remove role → all permissions revoked
  - Change analyst access → update role → all analysts updated

RBAC in SQL — GRANT and REVOKE

Analysts can query but not modify; engineers can read and write; admins have full control

```
-- Create roles
CREATE ROLE analyst;
CREATE ROLE engineer;
CREATE ROLE admin;

-- Grant permissions to roles
GRANT SELECT ON customers, orders, products TO analyst;
GRANT SELECT, INSERT, UPDATE ON customers, orders TO engineer;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin;

-- Assign users to roles
GRANT analyst TO alice;
GRANT engineer TO bob;
GRANT admin TO charlie;

-- Revoke access
REVOKE engineer FROM bob;
```

Attribute-Based Access Control (ABAC)

- More flexible than RBAC: permissions based on attributes of user, resource, and environment
- Attribute types:
  - User: department, job title, clearance level, location
  - Resource: data classification (public, confidential), owner
  - Environment: time of day, IP address, device type
- Example policy:
  - "Finance department can access confidential financial data, but only from corporate network during business hours"
- RBAC vs ABAC:
  - RBAC: simpler, easier to audit, covers most use cases
  - ABAC: more expressive for complex, context-dependent policies
  - Many real systems use a hybrid approach

Views as Access Control

Simple, widely supported, works in every SQL database

```
-- Create a view that exposes only safe columns
CREATE VIEW customer_analytics AS
SELECT
  customer_id,
  age_group,
  region,
  purchase_count,
  avg_order_value
FROM customers;
-- Note: name, email, SSN, address are EXCLUDED

-- Grant access to the view, not the table
GRANT SELECT ON customer_analytics TO analyst;
REVOKE SELECT ON customers FROM analyst;

-- Analyst queries customer_analytics freely
-- but never sees raw PII
```

Row-Level Security (RLS)

Use cases: multi-tenant SaaS, regional access, hierarchical org access

```
-- Different users see different ROWS of the same table
-- Example: regional analysts only see their region's data

-- PostgreSQL Row-Level Security
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;

CREATE POLICY regional_access ON orders
  FOR SELECT
  USING (region = current_setting('app.user_region'));

-- East Coast analyst: SELECT * FROM orders
-- → only sees rows where region = 'east'

-- Same table, same query, different results
-- depending on who is asking
```

Column-Level Encryption

customer_id	name	ssn	region
1001	Alice Smith	0x7A3F...encrypted...9B2E	east
1002	Bob Jones	0x4C8D...encrypted...1F7A	west
1003	Carol Lee	0x9E2B...encrypted...4D8C	east

Column-Level Encryption — Details

- Encrypt individual columns containing sensitive data
- Even users with full table access see ciphertext unless they have the decryption key
- Advantages:
  - Defense in depth — if access controls fail, data is still protected
- Challenges:
  - Can't easily query or index encrypted columns
  - Key management is complex: who holds keys, how are they rotated?
  - Performance overhead for encryption/decryption
- Used in AWS RDS, Azure SQL, PostgreSQL with pgcrypto extension

Dynamic Data Masking

Database returns masked values to unauthorized users, real values to authorized users

```
-- Define masking rules (SQL Server / Snowflake syntax)
ALTER TABLE customers ALTER COLUMN email
  ADD MASKED WITH (FUNCTION = 'email()');
  -- Result: aXXX@XXXX.com

ALTER TABLE customers ALTER COLUMN ssn
  ADD MASKED WITH (FUNCTION = 'partial(0, "XXX-XX-", 4)');
  -- Result: XXX-XX-1234

-- Analyst sees:           Admin sees:
-- aXXX@XXXX.com           alice@gmail.com
-- XXX-XX-1234             123-45-6789

-- Authorized users granted UNMASK permission
GRANT UNMASK TO admin_role;
```

Database Access Control Mechanisms

Mechanism	Controls	Granularity	Complexity	Strength
Views	Which columns visible	Column	Low	Moderate
Row-Level Security	Which rows visible per user	Row	Medium	Strong
Column Encryption	Who reads sensitive values	Column	High (key mgmt)	Strong
Dynamic Masking	Displayed values	Column	Low	Weak-Moderate

The Human Side of Access Control

- Technical controls are necessary but not sufficient
- Common real-world failures:
  - Overly permissive defaults ("give everyone admin for now")
  - Shared service accounts (no audit trail, no accountability)
  - Permission creep without reviews
  - Shadow IT: analysts export data to personal laptops or Google Sheets
- Best practices:
  - Regular access reviews (quarterly): do these people still need these permissions?
  - Automated provisioning tied to HR systems
  - Principle of least privilege as cultural norm
  - Audit logging: every query on sensitive data is logged

There IS a tradeoff: Excessive security hampers velocity!

Break

Data Anonymization & De-identification

Why Anonymize Data?

- Common scenarios:
  - Research: publishing a dataset for academic use
  - Third-party sharing: giving data to a vendor or partner
  - Internal: analysts accessing production data without seeing raw PII
  - Compliance: GDPR's data minimization principle
  - Open data: government and public health datasets
- **The goal: preserve analytical utility while removing the ability to identify individuals**
- This turns out to be much harder than it sounds

Direct Identifiers vs. Quasi-Identifiers

Direct Identifiers

- Uniquely identify a person on their own
- Name, SSN, email, phone number, driver's license
- Obvious: must be removed before sharing

Quasi-Identifiers

- Don't identify alone, but CAN in combination
- ZIP code, date of birth, gender, occupation
- Less obvious: each seems harmless, but together they form a fingerprint
- This is the crux of why anonymization is hard

Latanya Sweeney's Key Result (1997)

- **87% of Americans can be uniquely identified by just three fields:**
- **5-digit ZIP code + date of birth + gender**
- She proved this by cross-referencing:
  - "Anonymized" medical records from Massachusetts (names removed, but ZIP + DOB + gender kept)
  - Public voter registration rolls (which contain name + ZIP + DOB + gender)
- She identified the medical records of the Governor of Massachusetts (William Weld)
  - Sent his health records to his office to make the point
- **Lesson: removing direct identifiers is not enough. Quasi-identifiers enable linkage attacks.**

Anonymization Techniques

- Suppression: remove the field entirely
  - Drop the name column, drop the SSN column
- Generalization: reduce precision
  - Age 29 → 25-30 | ZIP 10027 → 100** | Date 1995-03-15 → 1995
- Pseudonymization: replace with consistent pseudonyms
  - Alice Smith → User_7392 (consistent so joins still work)
  - Reversible with mapping table — GDPR still considers this personal data
- Masking: replace with realistic but fake values
  - 123-45-6789 → XXX-XX-6789
- Perturbation: add random noise to numerical values
  - Salary $95,000 → $93,200 (preserves aggregate stats)

Netflix Prize — Background

- In 2006, Netflix launched a $1 million competition to improve recommendations by 10%
- Released a dataset of 100 million movie ratings from ~500,000 users
- Netflix "anonymized" the data by:
  - Removing usernames
  - Replacing user IDs with random numbers
  - Perturbing some ratings slightly
  - Perturbing some dates slightly
- Netflix believed this was sufficient

Netflix Prize — Attack

- Researchers Narayanan & Shmatikov (UT Austin) broke the anonymization
- Method: cross-reference Netflix ratings with public IMDb ratings
  - Many Netflix users also rate movies on IMDb under their real name
  - Match on which movies rated + approximate dates
- **Even with perturbation, your pattern of movie ratings is a fingerprint**
- Results:
  - 6 movies rated → uniquely identify 99% of users
  - 2 movies + approximate dates → identify 68% of users

Netflix Prize — Consequences

- Once de-anonymized, attacker knows complete viewing history — may reveal:
  - Political views, religious beliefs, sexual orientation, health conditions, personal struggles
- A Netflix user sued under the Video Privacy Protection Act
  - Plaintiff: closeted lesbian mother — exposure of viewing history could damage her life
- Netflix canceled the planned Netflix Prize 2
- FTC investigated Netflix's privacy practices
- **Lesson for data scientists:**
  - "Anonymized" ≠ anonymous
  - Sparsity kills anonymity — in high-dimensional data, everyone is unique
  - Release is irreversible — get privacy right before publication

Anonymization techniques discussed don’t provide formal privacy guarantees.
We need a framework that mathematically bounds what an attacker can learn.

Privacy-Preserving Data Analysis

k-Anonymity — The Idea

- Proposed by Latanya Sweeney (2002) — formalized her 1997 finding
- **Definition: every record is indistinguishable from at least (k−1) other records on quasi-identifier fields**
- If k=5, any combination of (ZIP, age range, gender etc.) appears at least 5 times
- An attacker can narrow down to a group of k, but can't identify the individual
- To achieve k-anonymity: generalize and suppress until the property holds

k-Anonymity — Before Generalization

ZIP	Age	Gender	Diagnosis
10027	29	M	Heart Disease
10027	31	M	Flu
10028	45	F	Diabetes
10028	47	F	Heart Disease
10029	29	M	Cancer

k-Anonymity — After Generalization (k=3)

ZIP	Age	Gender	Diagnosis
100**	25-35	M	Heart Disease
100**	25-35	M	Flu
100**	25-35	M	Cancer
100**	40-50	F	Diabetes
100**	40-50	F	Heart Disease

k-Anonymity — Limitations

- Homogeneity attack:
  - If all k records in a group have the SAME sensitive value, attacker learns it
  - E.g., 5 people share (100**, 25-35, M) and all have diagnosis = "HIV"
  - Attacker knows the individual has HIV even without identifying which of the 5
- Background knowledge attack:
  - Attacker knows extra info about the target
  - E.g., "my neighbor is 28, male, in 100** and doesn't have the flu"
  - This eliminates possibilities and narrows the set
- **k-Anonymity provides no guarantee about what the attacker learns about the sensitive attribute**

l-Diversity

- Addresses the homogeneity attack in k-anonymity
- **Definition: each equivalence class must contain at least l distinct sensitive values**
- If l=3, each group must have at least 3 different diagnoses
- Example:
  - Group (100**, 25-35, M): {Heart Disease, Flu, Cancer} → l=3 ✓
  - Group (100**, 40-50, F): {Diabetes, Heart Disease} → l=2 ✓ if l≤2
- Limitation: doesn't consider distribution of values
  - {Cancer, Cancer, Cancer, Cancer, Flu, Diabetes} → technically l=3
  - But attacker is 67% confident the target has Cancer

t-Closeness

- **t-Closeness: distribution of sensitive values in each group must be within distance t of the overall distribution**
- Prevents both homogeneity and skewed-distribution attacks
- The cat-and-mouse problem:
  - k-anonymity → broken by homogeneity
  - l-diversity → broken by distribution skew
  - t-closeness → stronger, but requires heavy generalization, reducing utility
  - Each fix patches one hole and reduces usefulness
- **These are all syntactic guarantees — they define structural properties, not bounds on what an attacker can learn**
- We want a semantic guarantee: a mathematical bound regardless of auxiliary information