



Aqsa Khalid
Lecturer
School of Computing

CS3002 Information Security



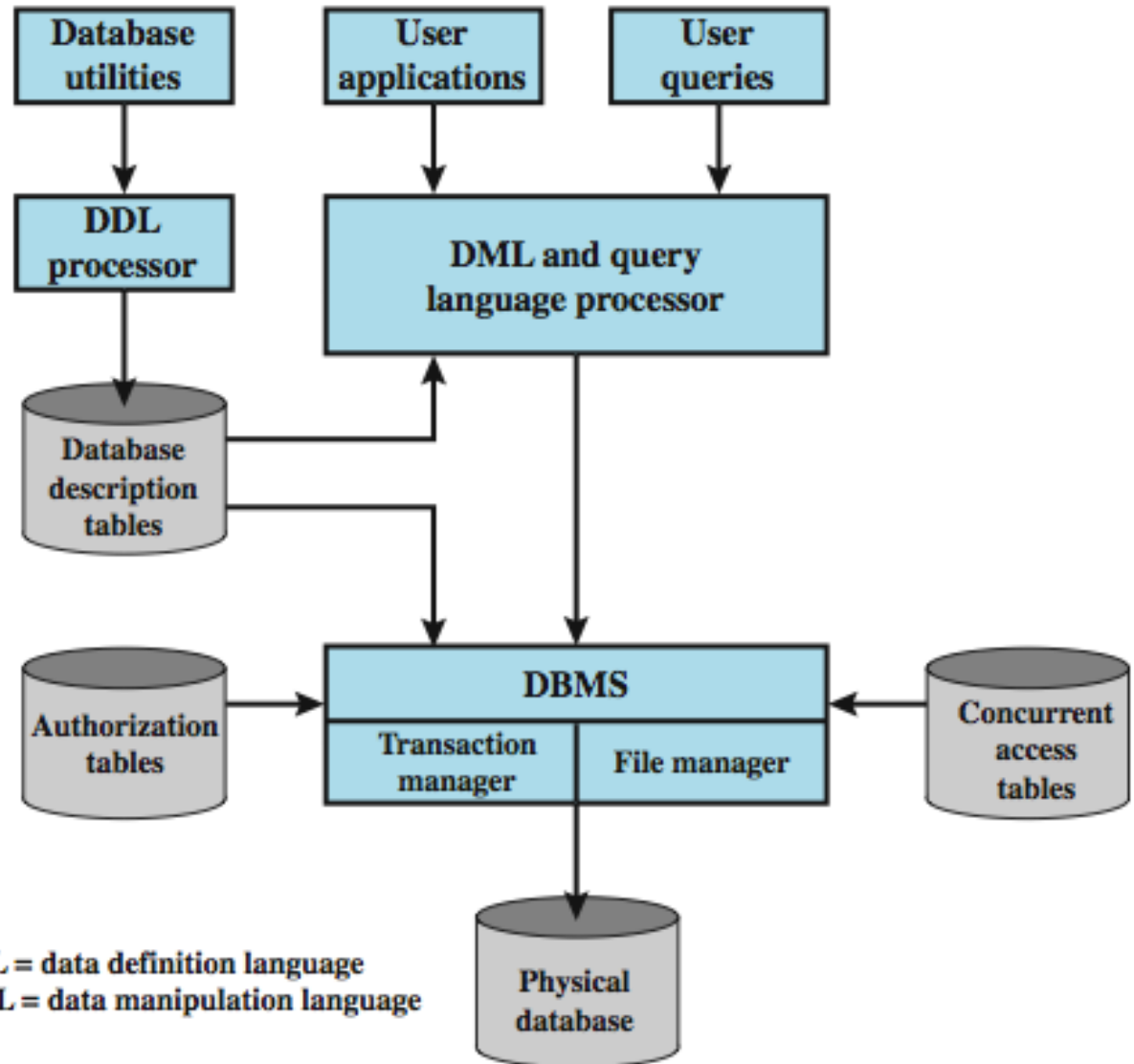
Database Security

Database Systems



- Structured collection of data stored for use by one or more applications
- Contains the relationships between data items and groups of data items
- Can sometimes contain sensitive data that needs to be secured
- Query language: Provides a uniform interface to the database

Database Architecture

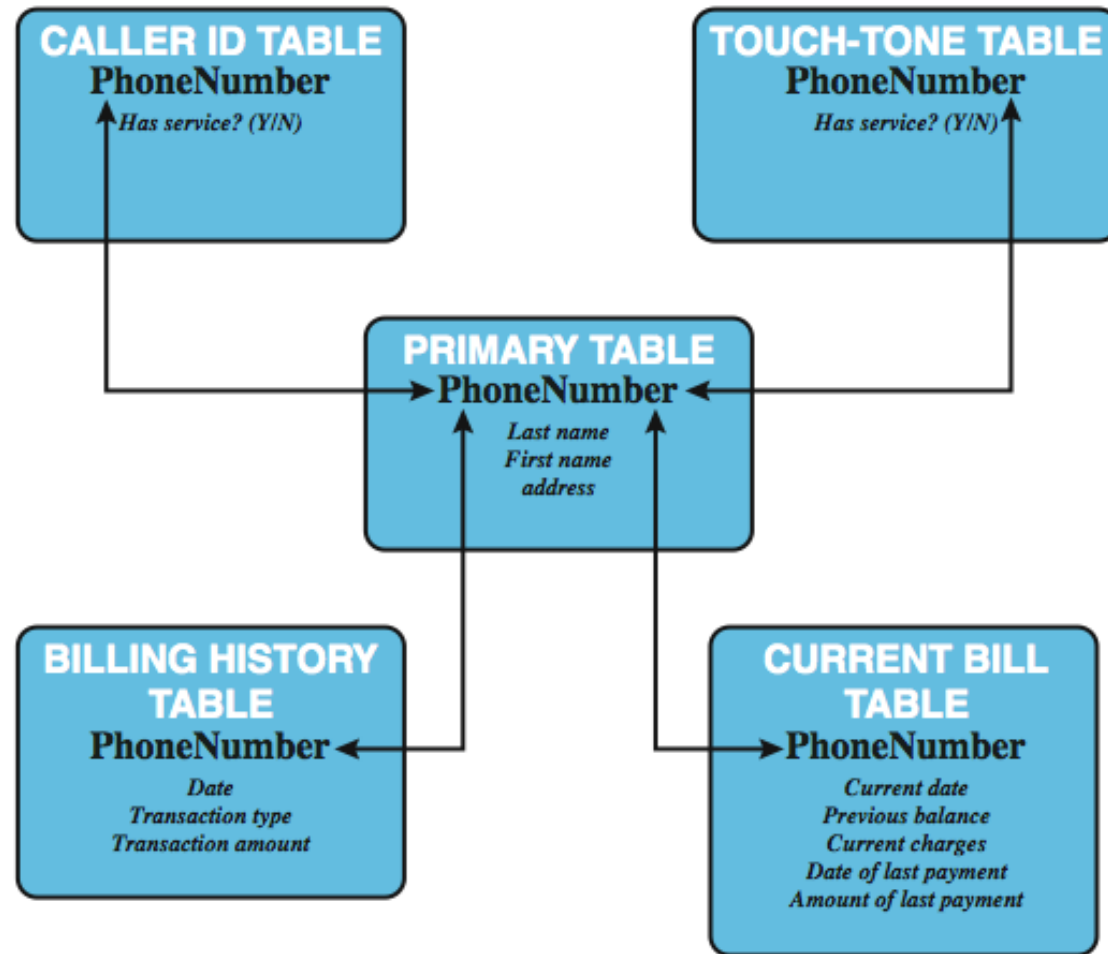


Relational Databases



- Table of data consisting of rows and columns
- Each column holds a particular type of data
- Each row contains a specific value for each column
- Ideally has one column where all values are unique, forming an identifier/key for that row
 - Enables the creation of multiple tables linked together by a unique identifier that is present in all tables
- Use a relational query language to access the database
 - Allows the user to request data that fit a given set of criteria

Relational Database Example



Relational Database Elements

Two tables in a db

Department Table

Did	Dname	Dacctno
4	human resources	528221
8	education	202035
9	accounts	709257
13	public relations	755827
15	services	223945

primary
key

Employee Table

Ename	Did	SalaryCode	Eid	Ephone
Robin	15	23	2345	6127092485
Neil	13	12	5088	6127092246
Jasmine	4	26	7712	6127099348
Cody	15	22	9664	6127093148
Holly	8	23	3054	6127092729
Robin	8	24	2976	6127091945
Smith	9	21	4490	6127099380

foreign
key

primary
key

A view derived from two tables

Dname	Ename	Eid	Ephone
human resources	Jasmine	7712	6127099348
education	Holly	3054	6127092729
education	Robin	2976	6127091945
accounts	Smith	4490	6127099380
public relations	Neil	5088	6127092246
services	Robin	2345	6127092485
services	Cody	9664	6127093148



Structured Query Language



- SQL: originally developed by IBM in the mid-1970s
- standardized language to define, manipulate, and query data in a relational database
- several similar versions of ANSI/ISO standard

SQL Examples

```
CREATE TABLE department (  
    Did INTEGER PRIMARY KEY,  
    Dname CHAR (30),  
    Dacctno CHAR (6) )
```

```
CREATE TABLE employee (  
    Ename CHAR (30),  
    Did INTEGER,  
    SalaryCode INTEGER,  
    Eid INTEGER PRIMARY KEY,  
    Ephone CHAR (10),  
    FOREIGN KEY (Did) REFERENCES department (Did) )
```

```
CREATE VIEW newtable (Dname, Ename, Eid, Ephone)  
AS SELECT D.Dname E.Ename, E.Eid, E.Ephone  
FROM Department D JOIN Employee E ON E.Did = D.Did
```



SQL Injection Attacks

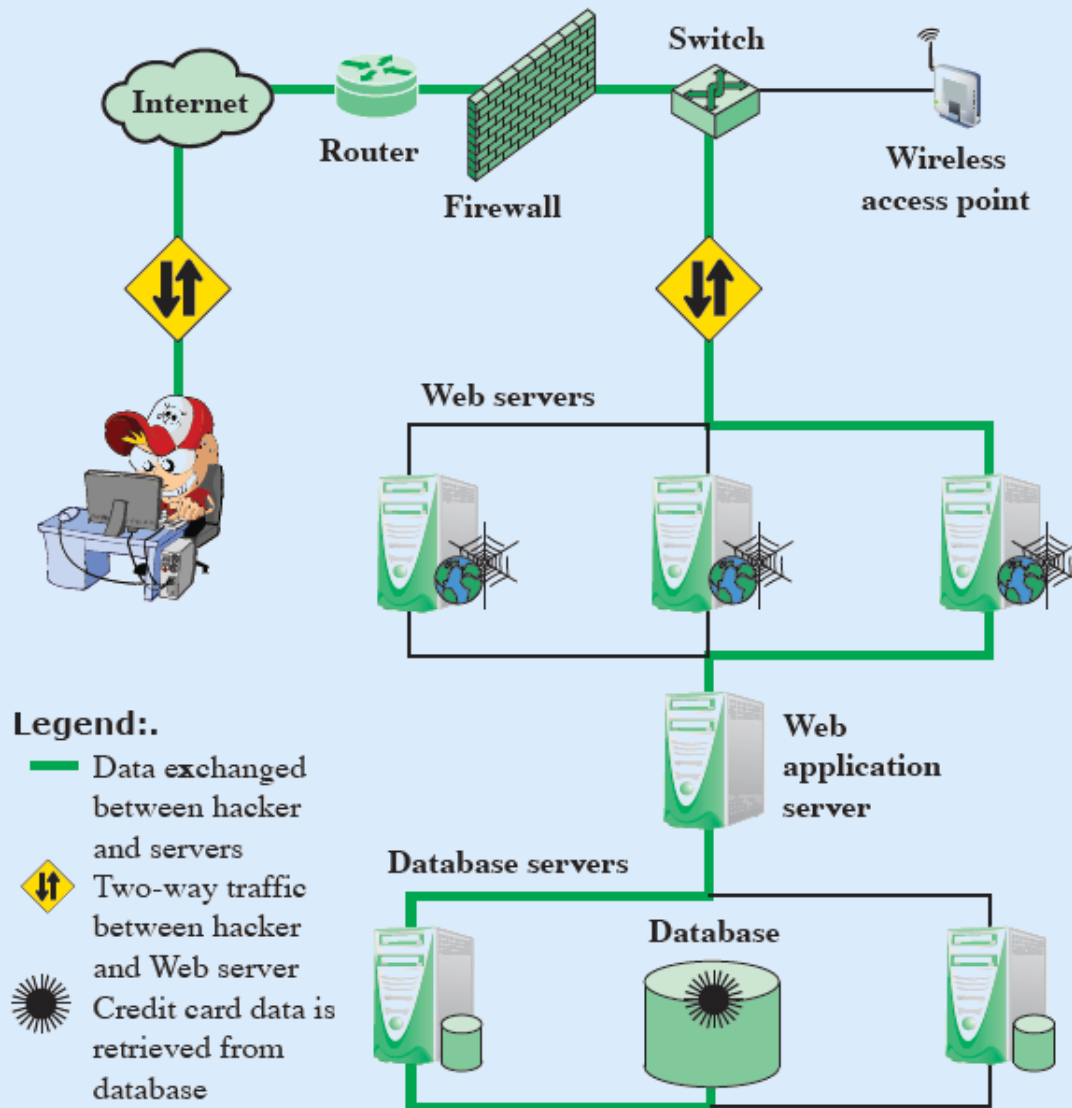


SQL Injection Attacks (SQLi)



- One of the most prevalent and dangerous network-based security threats
- Attacker sends malicious requests — containing SQL commands — to web (or application) server, which forward the commands to the database server
- SQL injection can be exploited to:
 - Steal data in bulk
 - Modify or delete data
 - Execute arbitrary operating system commands
 - Launch denial-of-service (DoS) attacks

A typical injection attack



A typical injection attack (steps)



1. Hacker finds a vulnerability in a custom Web application and injects a SQL command to a database by sending the command to the Web server. The command is injected into traffic that will be accepted by the firewall.
2. The Web server receives the malicious code and sends it to the Web application server.
3. The Web application server receives the malicious code from the Web server and sends it to the database server.
4. The database server executes the malicious code on the database. The database returns data from credit cards table.
5. The Web application server dynamically generates a page with data including credit card details from the database.
6. The Web server sends the credit card details to the hacker

Injection Technique



- The SQLi attack typically works by prematurely terminating a text string and appending a new command.
 - Injected string is terminated with SQL comment mark (--)

```
SELECT fname, lname, phone  
FROM student  
WHERE fname = 'user_input';
```

What if user types the following input?

```
John'; DROP TABLE Course; --
```

Example SQL injection



Server-side C# code to get data from a web form

```
var Shipcity;  
ShipCity = Request.form("ShipCity");  
var sql = "select * from Orders where ShipCity = '"  
          + ShipCity + "'";
```

If user enters **Redmond** in the form, query becomes

```
SELECT * FROM Orders WHERE ShipCity = 'Redmond';
```

Suppose, however, the user enters the following in the form:

```
Redmond'; DROP table Orders; --
```

This results in the following SQL query:

```
SELECT * FROM Orders WHERE ShipCity = 'Redmond'; DROP  
table Orders; --';
```

In-band attacks



In-band: same channel used for injecting SQL code and retrieving results

- **Tautology:** This form of attack injects code in one or more conditional statements so that they always evaluate to true
- **End-of-line comment:** After injecting code into a particular field, legitimate code that follows are nullified through usage of end of line comments
- **Piggybacked queries:** The attacker adds additional queries beyond the intended query, piggy-backing the attack on top of a legitimate request

Tautology Example



Server side PHP script

```
$query= "SELECT info FROM user WHERE name =  
'`$_GET["name"]`' AND pwd = '`$_GET["pwd"]`'";
```

Attacker submits following for the **name** field:

```
' OR 1=1 --
```

Query becomes

```
SELECT info FROM users  
WHERE name = '' OR 1=1 -- AND pwd = ''
```


Inferential Attack (gathering info)



There is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the website/database server

- **Illegal/logically incorrect queries:** lets an attacker gather important information about the type and structure of the backend database of a Web application
 - The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive.

Inferential Attack (gathering info)



There is no actual transfer of data, but the attacker is able to reconstruct the information by sending particular requests and observing the resulting behavior of the website/database server

- **Blind SQL injection:** Allows attackers to infer the data present in a database system even when the system is sufficiently secure to not display any erroneous information back to the attacker
 - The attacker asks the server true/false questions to observe the functionality in each case.

SQLi attempts – Real example



Lee Penkman created a form on his website. It was hidden, so actual users could not see it, only bots (scanning for vulnerabilities) could see and submit it.

Here are some of the things submitted:

- `1 UNION ALL SELECT NULL#`
- `-1 OR 2+456-456-1=0+0+0+1`
- `wrBEIRqX' || DBMS_PIPE.RECEIVE_MESSAGE(CHR(98) || CHR(98) || CHR(98),15) || '`
- `1;WAITFOR DELAY '0:0:5'—`
- `(SELECT (CASE WHEN (2646=4989) THEN 2646 ELSE 2646*(SELECT 2646 FROM INFORMATION_SCHEMA.CHARACTER_SETS) END))`

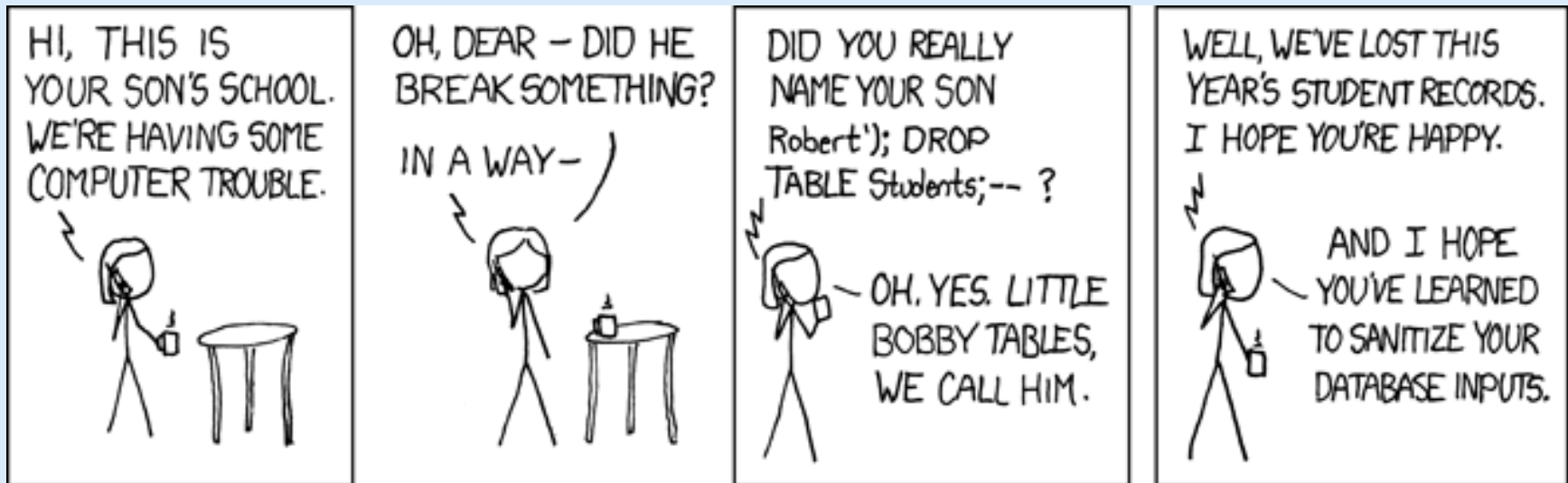
<https://github.com/lee101/hidden-form-on-the-internet>

SQLi countermeasures



- **Defensive coding:**
 - Sanitizing inputs
 - Encoding all unsafe characters (quotes, semicolons, hyphens etc.)
 - Stronger data validation
 - type checking, e.g. to check that inputs that are supposed to be numeric contain no characters other than digits
 - pattern matching to distinguish normal input from abnormal
- **Detection**
 - Signature based
 - Anomaly based
 - Code analysis
- **Runtime prevention:** Check queries at runtime to see if they conform to a model of expected queries

SQLi countermeasures



https://www.explainxkcd.com/wiki/index.php/327:_Exploits_of_a_Mom

Database Access Control



Database Access Control



- DBMS can provide specific access rights to portions of the database
 - e.g. create, insert, delete, update, read, write
 - to entire database, tables, selected rows or columns
 - possibly dependent on contents of a table entry
 - e.g. A teacher can view marks of their own students only
- can support a range of policies:
 - centralized administration
 - ownership-based administration
 - decentralized administration

SQL Access Control



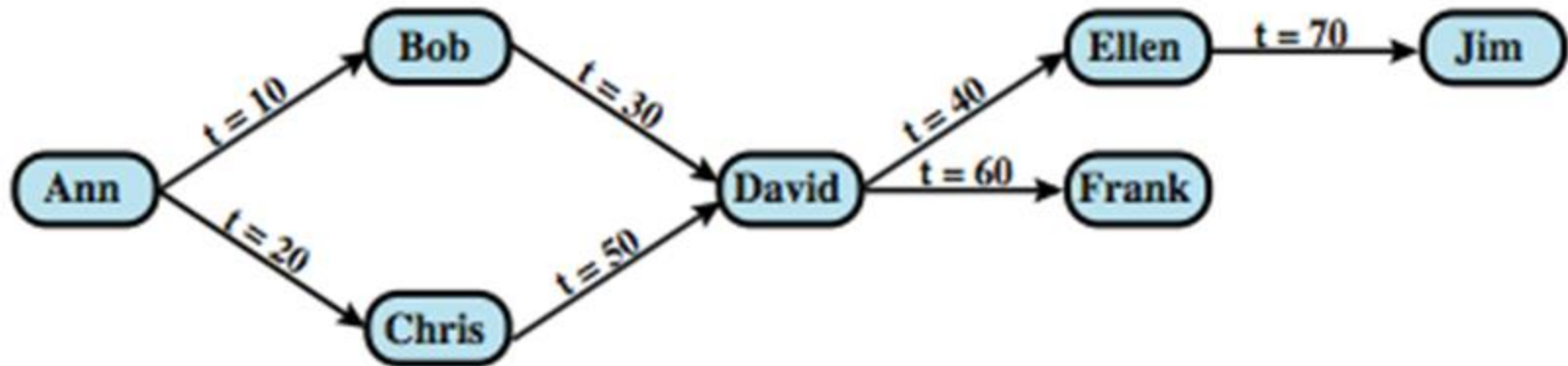
- If the user has access to the entire database or just portions of it
- Two commands:
- GRANT {privileges | role} [ON table] TO {user | role | PUBLIC} [IDENTIFIED BY password] [WITH GRANT OPTION]
 - e.g. **GRANT SELECT ON ANY TABLE TO john**
 - GRANT manager TO sarah** (role assignment)
- REVOKE {privileges | role} [ON table] FROM {user | role | PUBLIC}
 - e.g. **REVOKE SELECT ON ANY TABLE FROM john**
- WITH GRANT OPTION: whether grantee can authorize (GRANT) other users with the same privileges
- Typical access rights (privileges) are: SELECT, INSERT, UPDATE, DELETE, REFERENCES

Cascading Authorizations



Suppose access is granted **WITH GRANT OPTION**.

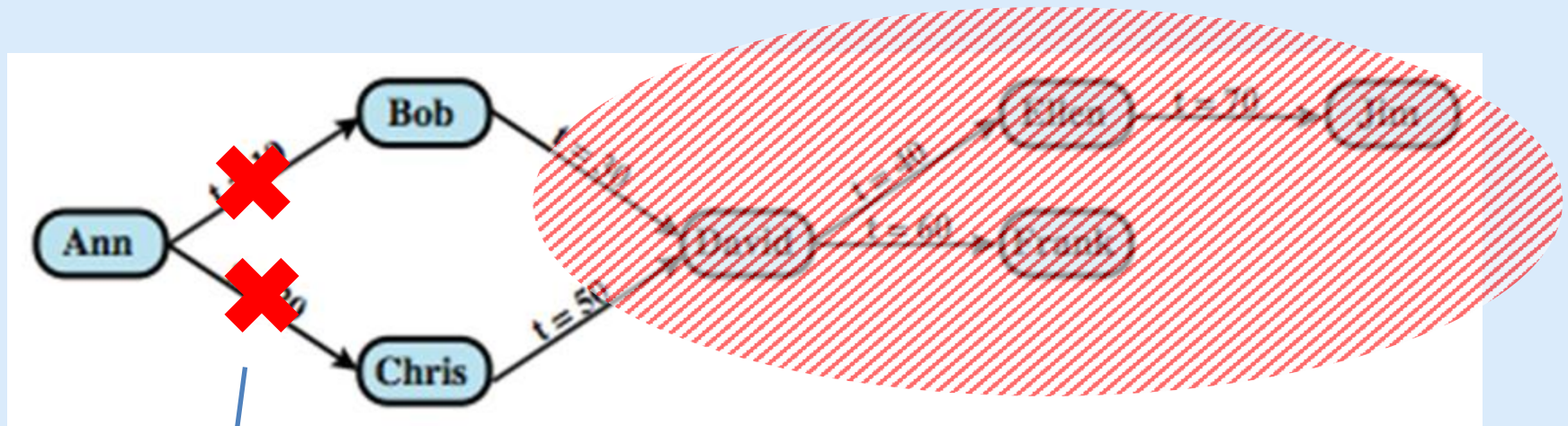
- Access right will cascade through.



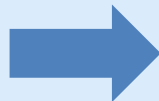
Cascading Authorizations



It works the same way with **revocations**.



If Ann revokes
Bob's and Chris'
access



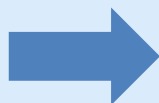
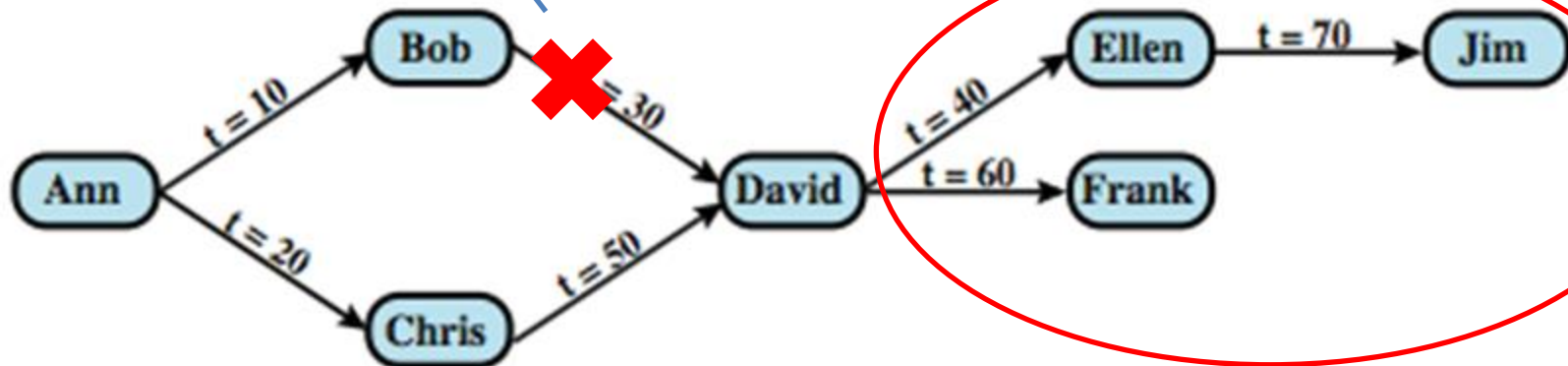
All cascaded access
should also be revoked

Cascading Authorizations



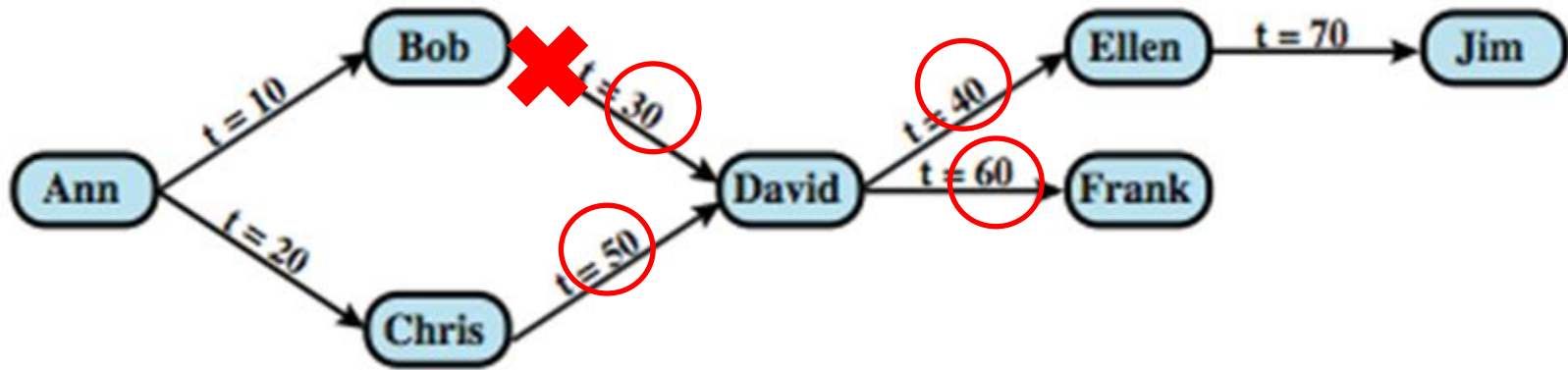
Complications can arise in some cases

Bob revokes
David's access

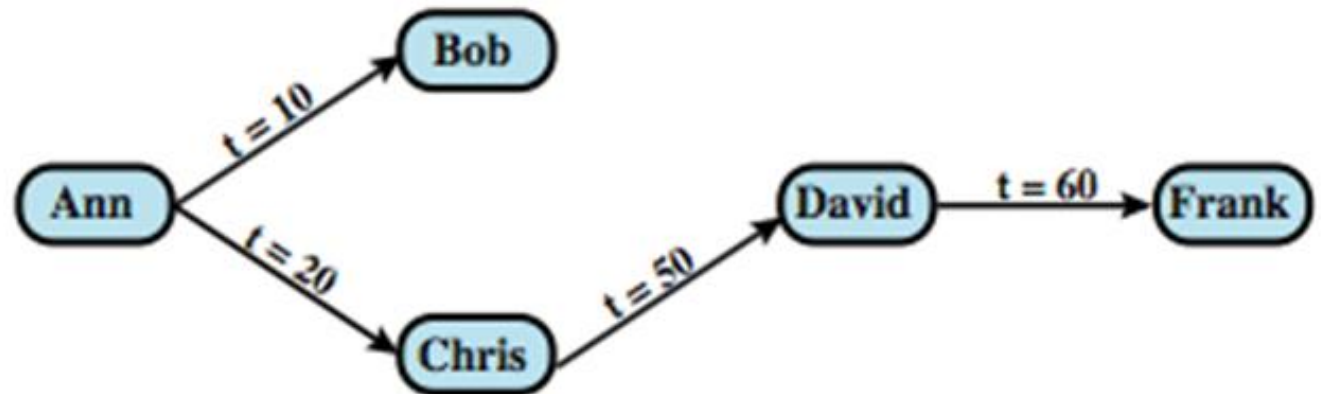


What to do with child accesses?
David's access was also granted by Chris

Cascading Authorizations



One option: Consider the access grant times



Role-Based Access Control (RBAC)



- Role-based access control work well for DBMS
 - eases admin burden, improves security
- Categories of database users:
 - application owner
 - end user
 - administrator
- DB RBAC must manage roles and their users

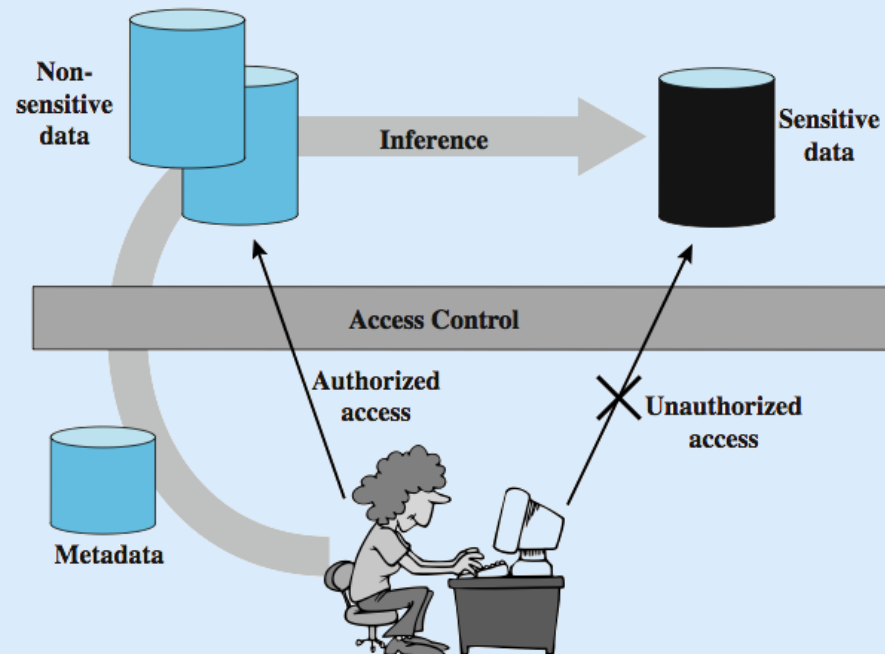
Database Inference



Inference



- The process of performing authorized queries and deducing unauthorized information from the legitimate responses received
- A combination of data items can be used to infer data of a higher sensitivity



Inference Example

Name	Position	Salary (\$)	Department	Dept. Manager
Andy	senior	43,000	strip	Cathy
Calvin	junior	35,000	strip	Cathy
Cathy	senior	48,000	strip	Cathy
Dennis	junior	38,000	panel	Herman
Herman	senior	55,000	panel	Herman
Ziggy	senior	67,000	panel	Herman

(a) Employee table

Position	Salary (\$)
senior	43,000
junior	35,000
senior	48,000

Name	Department
Andy	strip
Calvin	strip
Cathy	strip

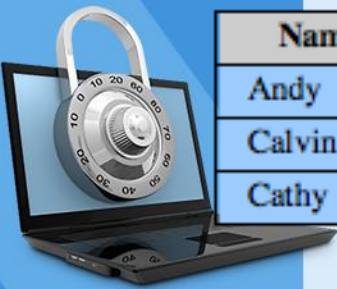
(b) Two views

Suppose a user is not allowed to see the salary of a specific employee, i.e. (Name, Salary) combination can not be in a query

Name	Position	Salary (\$)	Department
Andy	senior	43,000	strip
Calvin	junior	35,000	strip
Cathy	senior	48,000	strip

(c) Table derived from combining query answers

User can still derive the confidential information, because of same data ordering in two views



Inference countermeasures



- Inference prevention during database design
 - Alter database structure or access controls
 - Splitting a table into multiple tables
 - More fine-grained access control roles
- Inference detection at query time
 - by monitoring and altering or rejecting queries

Inference prevention during database design

To protect (Name, Salary) pair information, we can distribute data in three tables:

Employees (Emp#, Name, Department)

Salaries (S#, Salary)

Emp-Salary (Emp#, S#)

Protected Table

Now suppose we want to add a column of employee start dates. Which option is better (for preventing inference)?

Employees (Emp#, Name, Department)

Salaries (S#, Salary, Start-Date)

Emp-Salary (Emp#, S#)

OR

Employees (Emp#, Name, Department, Start-Date)

Salaries (S#, Salary)

Emp-Salary (Emp#, S#)



Statistical Databases



- Only provide data of aggregate/summary or statistical nature
 - e.g. sum, counts, averages
 - So that privacy of individuals is maintained
- Two types:
 - pure statistical database
 - ordinary database with statistical access: some users have normal access, others statistical
- Security problem is one of inference

Statistical Inference Examples



Example 1

- Count (EE.female) = 1
- Sum (EE.Female, GPA) = 2.7

Privacy compromised
(only one student in
group)

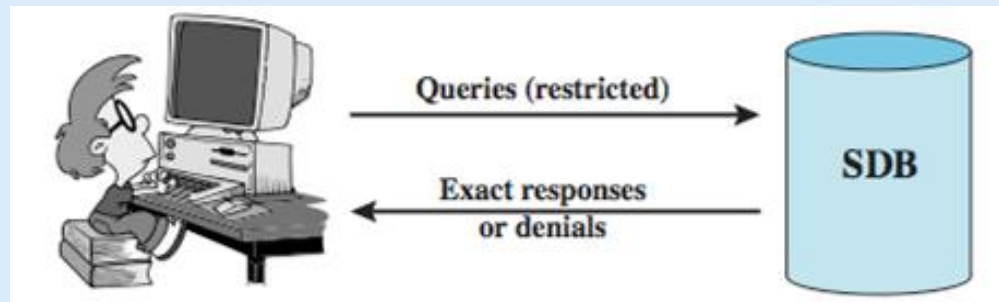
Example 2

- Non secret information:
 - Salary range for system analyst with BS: (50K, 60K)
 - Salary range for system analyst with MS: (65K, 70K)
- An attacker obtains the sum of all employee salaries
- After a new employee joins, same query is run again.
- The difference will reveal new employee salary and qualification

Protecting against Inference



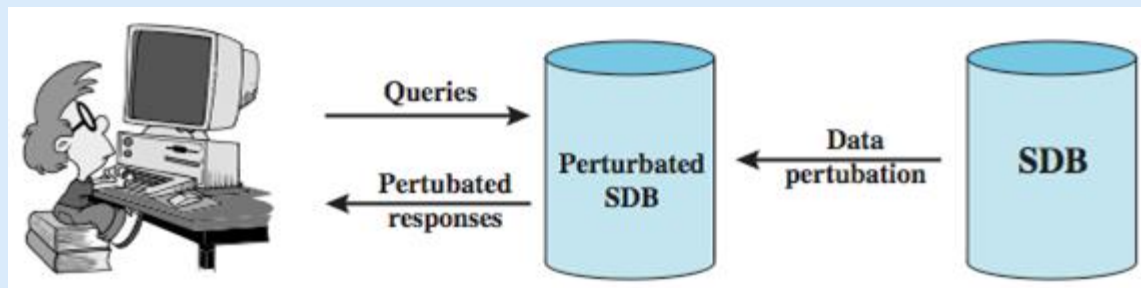
- Query restriction
 - Rejects a query that can lead to a compromise.
 - The answers provided are accurate.
- The simplest form of query restriction is query size restriction.
 - A query is permitted only if the number of records that match the query condition are more than a threshold, k (where $k > 1$).
 - In practice, queries over the whole table are allowed, enabling users to easily access statistics calculated on the entire tables.



Protecting against Inference



- Perturbation
 - Add noise to statistics generated from data
 - will result in differences in statistics
- 1. Data perturbation
 - Swap attribute data between rows so that individual records are not accurate, but overall statistics remain similar

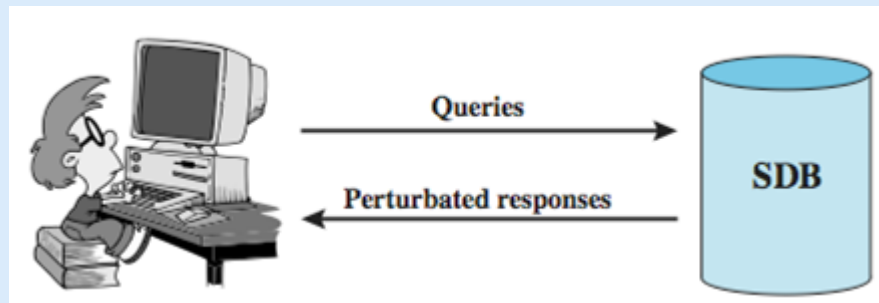


Protecting against Inference



2. Output perturbation

- **Random-sample query:** Calculate the statistics on a properly selected sampled query subset
- **Statistic adjustment:** Adjusting the answer up or down by a given amount in some systematic fashion
- Must minimize loss of accuracy in results



Database Encryption

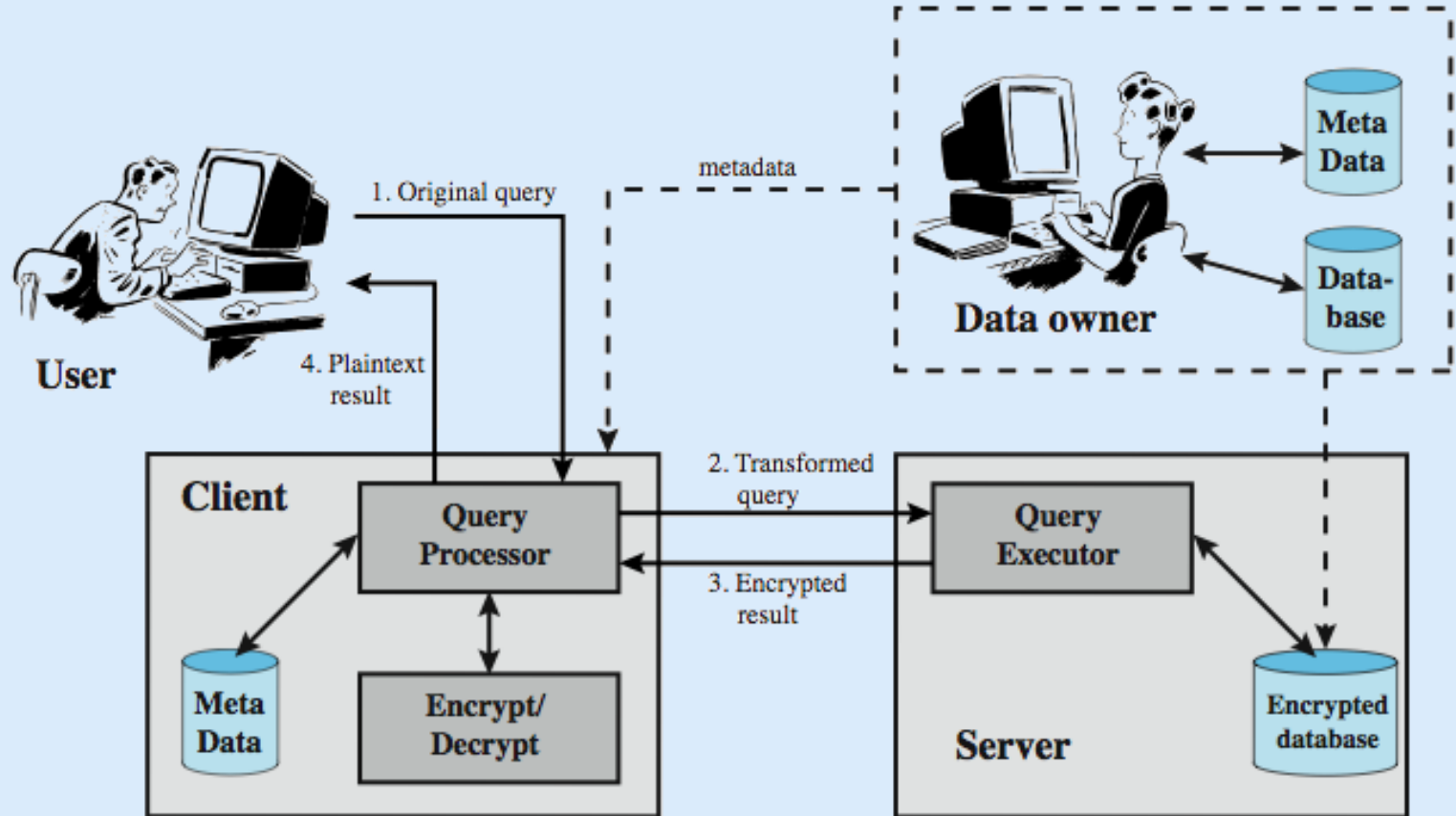


Database Encryption



- Databases typically a valuable info resource
 - protected by multiple layers of security: firewalls, authentication, OS access control systems, DB access control systems
- For particularly sensitive data, DB encryption is the last line of defense
- DB Encryption creates challenges
 - How to distribute decryption keys
 - How to execute queries, i.e. search for records

Querying an Encrypted DB



How to Encrypt



- Can encrypt
 - entire database (tables)
 - very inflexible and inefficient (how to execute queries?)
 - individual fields
 - simpler but still inflexible
 - records (rows)
 - best if attribute indexes are created to help data retrieval
- Varying trade-offs

Field Level Encryption



- Suppose that each individual item in the database is encrypted separately.
- The encrypted database is stored at the server, but the server does not have the key, so that the data is secure at the server. The client system does have a copy of the encryption key.
- A user at the client can retrieve a record from the database with the following sequence:
 1. The user issues SQL query for fields from one or more records with a specific value of the attribute.
 2. The query processor at the client encrypts the attribute parameters, modifies the SQL query accordingly, and transmits the query to the server.
 3. The server processes the query using the encrypted value of the attributes and returns the appropriate record or records.
 4. The query processor decrypts the data and returns the results to user.

Field Level Encryption



Example

```
SELECT Ename, Eid, Ephone  
FROM Employee  
WHERE Did = 15
```

Assume that the encryption key **k** is used and that the encrypted value of the department id 15 is $E(k, 15) = 1000110111001110$.

The query processor at the client could transform the preceding query into

```
SELECT Ename, Eid, Ephone  
FROM Employee  
WHERE Did = 1000110111001110
```

Field Level Encryption



Issues

- This method is certainly straightforward but lacks flexibility.
- For example, suppose the Employee table contains a salary attribute and the user wishes to retrieve all records for salaries less than 70K.
 - There is no obvious way to do this, because the attribute value for salary in each record is encrypted.
 - The set of encrypted values do not preserve the ordering of values in the original attribute.

Record Level Encryption



- Each record (row) of a table in the database is encrypted as a whole, and treated as a contiguous block.
- For each row in the original database, there is one row in the encrypted database.
- To assist in data retrieval, **attribute indexes** are associated with each table. For some or all of the attributes an index value is created.
 - For any attribute, the range of attribute values is divided into a set of non-overlapping partitions that encompass all possible values, and an index value is assigned to each partition.

Record Level Encryption



Example

- Suppose that employee ID (eid) values lie in the range [1, 1000]. We can divide these values into five partitions: [1, 200], [201, 400], [401, 600], [601, 800], and [801, 1000]; and then assign index values 1, 2, 3, 4, and 5, respectively.
- For a text field, we can derive an index from the first letter of the attribute value. For the attribute ename, let us assign index 1 to values starting with A or B, index 2 to values starting with C or D, and so on.
- Similar partitioning schemes can be used for each of the attributes.

Record Level Encryption



Example

(a) Employee Table

eid	ename	salary	addr	did
23	Tom	70K	Maple	45
860	Mary	60K	Main	83
320	John	50K	River	50
875	Jerry	55K	Hopewell	92

The clause

WHERE salary > 60K

gets transformed to

WHERE I(salary) > 2

(b) Encrypted Employee Table with Indexes

$E(k, B)$	I(eid)	I(ename)	I(salary)	I(addr)	I(did)
1100110011001011...	1	10	3	7	4
0111000111001010...	5	7	2	7	8
1100010010001101...	2	5	1	9	5
0011010011111101...	5	5	2	4	9