Hardware and Software Data Security

Sabrina De Capitani di Vimercati, *Dip. Elettronica per l'Automazione, Università di Brescia, 25123 Brescia, Italy*
Pierangela Samarati, *Dip. di Tecnologie dell'Informazione, Università di Milano, 20163 Crema, Italy*
Sushil Jajodia, *Dept. of Information & Software Engineering, George Mason University, Fairfax, VA 22030-4444*

Keywords:

Access Control Policies and Mechanisms, Access Control List, Asymmetric-key cipher, Audit, Authentication, Authorization, Availability, Biometric characteristics, Capability, Confidentiality, Cryptography, Data Security, Digital cash, Discretionary Access Control, Double-spending problem, Inference, Mandatory Access Control, Memory card, Non-repudiation, Public Key Infrastructure, Replay attacks, Role-based Access Control, Secrecy, Symmetric-key cipher, Smart token, Spoofing attacks, Suffing attacks, Trojan horse.

Contents

Glossary

AES: Advanced Encryption Standard.

Asymmetric-key cipher: An encryption algorithm based on methods involving a public key and a private key.

Authentication: Means of establishing the validity of a claimed identity.

Auditing: It is the monitoring and recording of events to investigate suspicious activity and/or to monitor and gather data about specific activities.

Audit log: A file including records showing who has accessed a system and what operations he/she has performed during a given period of time.

Authorization: The right granted to a user to exercise an action (e.g., read, write, create, delete, and execute) on certain objects.

Availability: A requirement intended to guarantee that information and system resources are accessible to authorized users when needed.

Biometric: Any specific and uniquely identifiable physical human characteristic (e.g., retina, fingerprints) that may be used to authenticate an individual.

Block ciphers: A symmetric-key cipher that encrypts a message by breaking it into blocks and encrypting each block.

Challenge-response: Common authentication technique whereby a user receives a random number (the challenge) and then provides some private information (the response) related to the received challenge.

Cipher: A cryptographic algorithm used to encrypt and decrypt messages.

Ciphertext: The result of encryption. A ciphertext contains the same information as the original plaintext, but makes it unintelligible to unauthorized parties.

Confidentiality: The assurance that private or confidential information not be disclosed to unauthorized users.

Cryptography: the art or science encompassing the principles and methods of transforming an intelligible message into one that is unintelligible, and then retransforming that message back to its original form.

DAC: Discretionary Access Control.

DES: Data Encryption Standard.

Data integrity: A requirement that information is not modified improperly.

Decryption: Any process to convert ciphertext back into plaintext.

Denial-of-service: Prevention of legitimate users of a service from using that service.

Digest: Commonly used to refer to the output of a hash function.

Double-spending: Double-spending refers to fraudulently spending the same digital money twice.

Encryption: Any process to convert plaintext into ciphertext.

Group: A set of users.

Hash function: A hash function h is a transformation that takes an input m and returns a fixed-size string, which is called the hash value.

IDEA: International Data Encryption Algorithm.

Identification: Means by which a user provides a claimed identity to the system.

Inference: An inference problem arises whenever some data can be used to derive partial or complete information about some other more sensitive data.

Integrity: Information has integrity when it is accurate, complete, and consistent. (See *data integrity* and *system integrity*.)

MAC: Mandatory Access Control.

NIST: National Institute of Standard and Technology.

Non-repudiation: A requirement intended to guarantee that users cannot deny actions they performed.

PIN: Personal Identifier Number.

Plaintext: The data to be encrypted.

Public Key Infrastructure: The framework and services that provide for the generation, production, distribution, control, accounting and destruction of public key certificates.

RBAC: Role Based Access Control.

Replay attacks: Attacks based on intercepting and recording messages between parties for their subsequent (illegitimate) replaying in a different context.

Role: A job function within an organization that describes the authority and responsibility related to the execution of an activity.

Secrecy: A requirement that released information be protected from improper or unauthorized release.

Security: The combination of integrity, availability, and secrecy.

Security mechanism: Low-level software and/or hardware functions that implement security policies.

Security policy: High-level guidelines establishing rules that regulate access to resources.

Smartcard: A small electronic device that contains electronic memory and is equipped with processing capabilities.

Stream cipher: A symmetric-key encryption algorithm that operates on a bit at a time.

Subject: An active entity that can exercise access to the resources of the system.

Substitution cipher: A cipher in which each letter of a message is replaced with another character, but preserves its position within the message.

Symmetric-key cipher: An encryption algorithm where the same key is used for encryption as decryption.

System integrity: A requirement that a system performs its intended functions while preventing deliberate or inadvertent unauthorized manipulation of the resources.

Token: A small device typically used by users to authenticate them to the system.

Transposition cipher: A cipher in which the original letters of a plaintext message are rearranged into a different, unintelligible sequence according to a fixed rule.

Trojan horse: A malicious program containing hidden instructions allowing the unauthorized collection of information and that masquerades as benign applications.

User: A person who interacts directly with a system.

Summary

Data security refers to the protection of information against possible violations that can compromise its secrecy, confidentiality, integrity, or availability. Secrecy is compromised if information is disclosed to unauthorized subjects. Integrity is compromised if information is modified in an unauthorized or improper way. Availability is compromised if users are prevented from exercising authorized access (denial-of-service).
Guaranteeing data security requires the establishment and enforcement of different kinds of controls, including the identification and authentication of the different parties in a system (e.g., users and machines), the enforcement of rules regulating access to the system and its resources, the use of encryption techniques to protect information in storage or in transit over the network, and the post-facto examination of all the activities in a system to point out vulnerabilities or violations.
This chapter discusses issues involved in establishing security restrictions to regulate access to data and resources in a system.

1 Introduction

Governments, commercial businesses, and individuals are all storing information in electronic form. This medium provides a number of advantages over previous physical storage: storage is more compact, transfer is almost instantaneous, and accessing via databases is simpler. The ability to use information more efficiently has resulted in a rapid increase in the value of information; many organizations today recognize information as their most valuable asset. However, with the electronic revolution, information faces new, and potentially more damaging, *security threats*. Unlike information printed on paper, electronic information can be copied leaving the original unaltered. Also, information in electronic form can potentially be stolen from a remote location and it is vulnerable from interceptions and alterations during communication.

Data security describes all measures taken to prevent unauthorized or improper access to electronic data - whether unlegitimate access can take the form of disclosure, alteration, substitution, or destruction of the data concerned. Data security can be classified as the provision of the following services:

- *Secrecy* (Confidentiality) Information that is stored on a system or transmitted over a network should be released, directly or indirectly, only to users authorized to access it.
- *Integrity* Information should be protected from unauthorized or improper alteration, that is, information must not be improperly modified, deleted, or tampered.
- *Availability* Users should not be prevented from accessing data for which they have the necessary permissions (*denials-of-service*).

Ensuring security requires the application of different protection measures at both the organizational level (organizational practices and user training) and the technical level. Technical services crucial to the protection of data include *Authentication*, *Access Control*, *Audit*, and *Encryption*.

Authentication Authentication establishes the validity of one party to another, where parties can be human users or computers. Authentication can also be employed in communication system to ensure the validity of transmitted messages.

Access Control Access control is concerned with evaluating every request, submitted by users who have entered the system, to access data and resources to determine whether the request should be allowed or denied based on a specified policy.

Audit Audit is an independent review and examination of records and activities in the system to assess the adequacy of system controls, to ensure compliance with established policies and operational procedures, and to recommend necessary changes in controls, policies, or procedures.

Encryption Allows the coding of information so to make it unintelligible to parties not authorized to access it. It also allows to signal possible improper alterations.

In the remainder of this chapter we describe each of these services in more details.

2 Authentication

Authentication is a means of establishing identities. Generally speaking, authentication allows the establishment of the identity of one party to another, where parties can be computers or human users. The most popular form of authentication is the authentication of a user to a computer, by which a machine ensures the correctness of the identity of users requesting access to its resources. In a computer to computer interaction, authentication can be required to be performed in both directions, as in the case of peer-to-peer communication. Mutual authentication can also be used in a client-server scenario (although typically only client authentication is enforced). Also in a user-to-computer interaction, authentication of a computer can be used, to ensure the user of the identity of the machine with which he/she is interacting and thus preventing

against spoofing attacks. In spoofing attacks a system masquerades as another system, tricking the user into disclosing information.

Authentication can be certainly seen as the most primary security service on which other security services depend. As a matter of fact, good authentication is a prerequisite for correct access control and auditing: if a user's identity is incorrect, so will be the privileges granted (or denied) to the user by the access control mechanism and the accountability attribution of the auditing controls.[1]

The most common ways to enforce user to computer authentication are based on the use of:

- something the user *knows* (e.g., a username and password);
- something the user *possesses* (e.g., a smartcard);
- something the user *is* (e.g., fingerprints and retinal scan).

These techniques can be used in alternative or in combination; thus providing a stronger protection. For instance, a smartcard may require that a password be entered to unlock it.

Authentication based on something the user knows The most common form of authentication is based on the assignment to each user of an identifier (e.g., a computer login) and an associated password (string, PIN, or passphrase). While the identifier can be public, the password is assumed to be known only to the legitimate user. The password is stored in the system in encrypted form. To access the system, a user provides his identifier (declaration of identity) and the corresponding password (proof of identity). Since only the user is assumed to know the password, matching of the encrypted version of the password provided with that stored at the system for the declared identifier successfully authenticates the user. The benefits of password-based authentication are that it is very simple, cheap, and easy to enforce. All these characteristics make password-based control the most commonly used authentication measure. However, strength is traded off for such simplest and low cost, and password-based techniques are the weakest form of authentication: security strictly depends on maintaining the secrecy of the password, which however can be compromised. For instance, passwords can be *sniffed* (i.e., observed by unlegitimate users) when in transit over the network, *snooped* by people observing the legitimate users when they key them in, or simply *guessed* by intruders. A common bad practice that makes password vulnerable to guessing attacks is the users tendency to choose passwords that are easy for them to remember, such as their birthdate, the name of their relatives or pets, or their favorite sport. Other diffused bad practices that put password's secrecy at risk are writing the password down, to not forget it, or pass it to colleagues, as a quick and dirty solution to a file sharing problem. Instead, passwords should always remain private: giving away our password means giving someone else the ability of masquerading as ourselves to the system, being retained accountable for all the actions they will execute (which are recorded as associated with our identifier).

As user bad practices are one of the major cause of password vulnerability, a primary aspect for the success of password-based techniques is proper user training and awareness. Also, password security can be improved by the enforcement of additional controls. For instance, sniffing can be prevented by encrypting the communication between the user and the computer. Snooping can be prevented by protecting the keyboard from the view of others and by not echoing it on the screen. Guessability of passwords can be reduced by adopting password generators or dictionary controls. In the first case, the computer chooses the password for the users. In the second case, the computer simply controls the strength of the passwords chosen by the users, rejecting those passwords considered too easy and therefore guessable by an adversary. A good password should be at least 8 character long, should use a reasonably large character set (possibly mixing alphanumeric characters with special characters), and still should be easy to remember (otherwise users would be tempted to write it down or they would suffer denial-of-service in case of forgotten passwords). Also, passwords should not correspond to real words (or a slight variation of them), since this would make them vulnerable to dictionary attacks, by which an adversary automatically attempts all words in a dictionary. It is also a good practice to change passwords frequently (e.g., once a month). Automatic controls can be applied to enforce periodical password changes: passwords are assigned a limited lifetime after which they become invalid; users are therefore forced to change their password upon expiration. These controls can also be coupled with history checks, to avoid users reusing the same password before a specified time frame.

[1] Accountability is the ability to link all the activities to the users who exercised them.

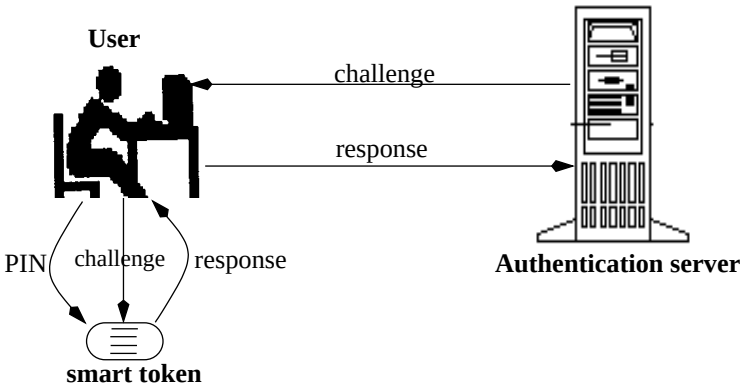

Figure 1: An example of challenge-response handshake

Authentication based on something the user possesses Authentication is based on possession by users of objects, called *tokens*. Each token has a unique private cryptographic key stored within it, used to establish the token's identity to the computer. Token-based authentication is stronger than password, as by keeping control on the token the user maintains control on the use of his identity. However, token authentication proves only the identity of the token, not of the user presenting it. The main weakness of such an approach is that tokens can be forged, lost, or stolen; anybody gaining possess of a token would be able to masquerade as the legitimate owner. To solve this problem, token-based authentication is often combined with the request of a proof of knowledge by the user. As an example, think of the Automatic Teller Machine (ATM), where a card is used together with a PIN (Personal Identification Number). The PIN is usually a string of four numeric digits, and works like a password. The combination of token and password clearly provides better security than each of the measures singularly taken. Indeed, to enter a system, an intruder needs both to present the token and to enter the PIN.

The simplest form of token is a *memory card*. Memory cards have storing capabilities, but do not have any processing ability. They cannot therefore perform any check on the PIN or encrypt it for transmission. This requires sending the PIN to the authentication server in the clear, exposing the PIN to sniffing attacks and requiring trust in the authentication server. More sophisticated tokens, called *smart tokens*, are equipped with processing capabilities (e.g., tokens incorporating one or more integrated circuits). For instance, the ATM cards are provided with processing power that allows the checking and encrypting of the PIN before its transmission to the authentication server. Smart tokens can use different types of authentication protocols, which can be classified as *static password exchange*, *dynamic password generators*, and *challenge-response*. With static password exchange, the user authenticates himself to the token and the token authenticates the user to the system. In the dynamic password generator approach, a token dynamically changes its key, by periodically generating a new key to be used. To authenticate the token to the system, the user reads the current key for the token and types it into the system. Alternatively, the key can be communicated to the system by the token. The challenge-response approach is the one most commonly used. It works on a challenge-response handshake as follows (see Figure 1). The party establishing the authentication issues a challenge (e.g., a random string number). The token generates a response to the challenge using the token's private key. Like for the dynamic case, communication between the token and the system can be enforced with or without the user intervention. In the first case, the challenge is keyed into the token by the user and the response displayed by the token is again keyed by the user into the workstation and communicated to the authenticating party. In the second case, the workstation is equipped with a reader that can directly interact with the token eliminating the need for the user to key in the challenge and response. An example of smart token is represented by smart cards, sophisticated token devices that have both processing power and direct connection to the system. Each smart card has a unique private key stored within. To authenticate the user to the system, the smart card verifies the PIN. It then enciphers the user's identifier, the PIN, and additional information like date and time, and sends the resulting ciphertext to the authentication server. Authentication succeeds if the authentication server can decipher the message properly.

Authentication based on something the user is Authentication techniques in this category exploit *biometric* characteristics of the users. These can be *physical characteristics*, such as fingerprints, hand shape, and characteristics of the eyes and face, or *behavioral characteristics*, like signatures, voiceprint, handwriting, and keystroke dynamic. The first step in the application of biometric techniques is the measurement of the interested characteristic to the purpose of defining a template for it. This step, called *enrollment phase*, generally comprises of several measurements of the characteristic (e.g., to define the voiceprint for an individual several inputs need to be considered). Based on the different measurements, a template is computed and stored for authentication. When a user presents himself to the system, the relevant characteristic is measured and matched with the stored template. Notice that, unlike for password and token-based techniques, biometric-based authentication cannot require an exact match. While a password either matches the one stored at the authentication server or it does not, no two signatures of a person are an exact copy one of the other. The authentication result is therefore based on how closely the measured characteristic matches the stored template. Authentication succeeds if the difference is within an acceptable predefined threshold; it fails otherwise. An important, and not easy, task is therefore the definition of the acceptable threshold, which must guarantee a high rate of successes (correct authentication of legitimate users and rejection of attackers) and low rate of insuccesses.

Although they can be less accurate, biometric techniques are stronger than either password or token based techniques. Indeed, they eliminate the weaknesses due to the possibility of the identity proof (password or token) being acquired by unlegitimate users. However, the use of biometric techniques is still limited because of the high cost and expensive equipment needed. Moreover, their intrusive nature limits user acceptance and large scale use. For instance, retinal scanners, which are one of the most accurate biometric methods of authentication, have raised concerns about possible harms that the infrared beams sent to the eye by the scanner can cause. Also, deployment of biometric technology in a large scale is certain to raise social and political debates, since unforgeable biometric authentication could result in significant loss of privacy for individuals.

From a strictly technical point of view, the best authentication solution would be the combination of user-to-token biometric authentication, followed by mutual cryptographic authentication between the token and system services.

2.1 Attacks

We have already mentioned some of the most popular attacks (e.g., password spoofing or dictionary attacks) to fool authentication mechanisms, and possible defenses against these attacks. Another popular class of attacks to password secrecy is represented by *replay* attacks. Replay attacks are based on intercepting and recording messages between parties for their subsequent (illegitimate) replaying in a different context. When replayed, messages can be redirected to recipients other than the one originally intended, or they can be repeated in different protocols or protocol runs. A way to combat replay attacks is to ensure that the information to be exchanged across the network be different each time. Methods which prevent replay attacks are known as *strong authentication* and can be divided into three classes: *shared sequence*, *challenge-response*, and *asymmetric-key*. In shared sequence methods, the user and the service share a sequence of one-time passwords that the user can present to authenticate himself at the service. A one-time password can be used only for one connection, and once used it becomes invalid. Even if the password is sniffed all replay attacks trying to use it will therefore fail. In challenge-response methods, the service generates a challenge string, which must be different for each transaction, and sends it to the user. The user computes a response to the challenge with a function dependent on both the challenge and the user's key. The function used to produce the answer must be such that it must practically impossible (meaning infeasible from a computational point of view), given the challenge and the response to it, to reconstruct the password. Since the challenge is different every time, replay attacks cannot succeed (as possibly intercepted responses cannot be reused). In asymmetric key methods, the user possesses a pair of keys: a public key k, which is widely publicized; and a private key k^{-1}, which is kept secret. Whenever a user wants to authenticate himself to a service, he/she sends a message signed (i.e., encrypted) with his/her private key. If the service can decrypt the message correctly by using the corresponding public key, it can be certain that the message was encrypted by using the user's private key (which should be known only to the user). Analogously, a service which encrypts its replies with the user's public key can be confident that they can only be read by using

the corresponding private key. We will illustrate asymmetric-key techniques in more details in Section 5.

3 Access control

Access control evaluates the requests to access resources and determines whether to grant or deny them. Typically, access control operates after authentication has taken place, evaluating all requests to access resources of users who have successfully entered the system. The ability of users to access resources usually depend on their identity, which must be therefore properly authenticated. In studying access control, it is useful separate security *policies* from *mechanisms*. A security policy defines high-level guidelines establishing rules that regulate access to resources. Mechanisms are low-level software and/or hardware functions that implement the policies. The design of an access control system is usually performed with a multiphase approach, from the analysis of the security requirements, to the definition and formalization of the policies, to their final implementation in a security mechanism. The formalization of security policies introduces the concept of an *access control model*, that formally defines the entities that part of the system (e.g., users and resources), the accesses to be controlled (operations), and the rules regulating access. The definition of a formal model allows to reason about the properties that the resulting system will have and prove security results. By proving properties on the formalized model and by proving that the mechanism correctly implements the model, we can claim that the mechanism enjoys those properties. Among the properties that a security model must satisfy there are the basic *completeness* and *consistency*. Completeness ensures that *all* the input security requirements to be addressed are satisfied. Consistency requires that the model be free of contradictions: an access cannot be simultaneously granted and denied. Many mechanisms have been developed and they vary in terms of precision, sophistication, and cost. Access control mechanisms are generally based on the definition of a *reference monitor* that intercepts every access request to objects in the system and examines whether the request should be granted or not, according to the access control policy to be enforced. The reference monitor must be:

- *tamper-proof*: the reference monitor cannot be altered;
- *non-bypassable*: each access request must be filtered by the reference monitor;
- *kernel-based*: the reference monitor should be confined in a limited part of the system (splitting the security functions all over the system would require to verify of all the code);
- *small*: the reference monitor should be enough small so to make formal verification possible.

Obviously, the reference monitor that enforces a certain access control policy should be trusted by the authority that specifies the policy. Multiple reference monitors can be involved in specific access decisions; as in the case of distributed systems where policies specified by different authorities govern the access to certain objects.

The separation between policies and mechanisms has many advantages: access requirements can be discussed independently of their implementation to reason about their correctness and properties; different access control policies as well as different mechanisms enforcing the same policy can be compared; and it is possible to design mechanisms that enforce multiple policies.

Access control policies can be divided in three major categories: *discretionary access control* (DAC), *mandatory access control* (MAC), and the most recent *role-based access control* (RBAC).

3.1 Discretionary Access Control

The Trusted Computer System Evaluation Criteria (TCSEC) defines discretionary access control as

> "A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)"

Discretionary access controls therefore base on (discretionary) rules that state who can exercise given accesses. The simplest form of rule is an authorization tuple, usually of the form $\langle u, o, a, p \rangle$, where u is the user to whom the access rule is granted, o is the object, a is the action and represents the type of access that the user can exercise on the object, and p is a predicate expressing conditions over the access. Authorization tuple $\langle u, o, a, p \rangle$ states that user u can execute action a on object o provided that predicate p is satisfied. The simplest form of predicates are system conditions, that is, conditions that can be evaluated by checking the state of the system, like the time or the location of a request. For instance, an authorization can state that "Bob can write file accounts but only between 8am and 5pm and from machines in the bank building". Predicates can also express content-dependent conditions, which restrict the authorization validity depending on the content of the object, or history-based conditions, which restrict the validity of the authorization depending on previous accesses. Specific subjects, objects, and actions to which authorizations can be referred may be different from system to system. For instance, in an operating system, objects will be files and directories, while in database systems, tables and records within them might be considered as objects. Actions for which authorizations can be specified include the following access modes: *read*, to provide users with the capability to view information; *write*, to allow users to modify or delete information; *execute*, to allow users to run programs; *delete*, to allow users to delete system resources; and *create*, to allow users to create new resources within the system. Authorization tuples can be referred to single users, actions, and objects, or to sets of them. The reference of authorizations to sets of entities is usually done by defining named *groups* of these entities. Usually groups need not be disjoint (i.e., a user can belong to several groups) and can be nested (i.e., groups can be defined as members of other groups). Groups together with the membership relationship form a hierarchy usually depicted as a directed acyclic graph. Figure 2 illustrates an example of users, actions, and objects hierarchies. Authorizations assigned to a group can be enjoyed by all its members. Whenever a user requests an access to an object, his/her request is checked against the specified authorizations. If there exists an authorization applicable to the request, the access will be granted; it will be denied otherwise.

Access control policies based on the (positive) authorization tuples granting privileges are called *closed*: an access is granted only if there is an authorization for it. Alternatively, an *open* policy can be applied which is based on negative authorizations. In an open policy, authorization tuples state accesses that must be denied, and an access is granted only if it is not denied by any authorization. Recent proposals adopt a hybrid approach, combining in a single model the use of both positive and negative authorizations. The combination of positive and negative authorizations provides more flexibility and control in the specification of authorizations. For instance, the owner of an object who is delegating administration to others, can specify a negative authorization for a specific user to ensure that the user will never be able to access the object, even if others grant him/her a positive permission for it. Negative authorizations can also be used to specify exceptions. For instance, suppose we wish to grant an authorization to all members of a group, except Bob. In the absence of negative authorizations, we would have to express the above requirement by specifying a positive authorization for each member of the group except Bob. If negative authorizations are supported, the same requirement can be expressed by granting a positive authorization to the group and a negative authorization to member Bob. While increasing flexibility, negative authorizations introduce the possibility of conflicts, meaning the presence of both a negative and a positive authorization for an access. Different policies can be applied to resolve conflicts. Among them, the most intuitive and natural is the *most specific takes precedence* according to which authorizations specified for an entity (user, object, or action) take precedence over authorizations specified for groups in which the entity belong. With reference to the example just illustrated, this conflict resolution policy would consider, in evaluating Bob's access, the negative authorization granted to Bob personally as prevailing over the authorization granted to the group in which he belongs. The most specific takes precedence criteria is intuitive and natural, as it expresses the concept of "exception". However, it does not solve all possible conflicts. For instance, Alice can belong to two groups (which are not in a membership relationship) holding conflicting authorizations. Alternative or additional conflict resolution policies that ensure complete conflict resolution include: the *denials-take-precedence* policy (negative authorizations always win); the *permissions-take-precedence* policy (positive authorizations always win); explicit assignments of priorities to the authorizations; grantor-dependent resolution (the strength of an authorization depend on who granted it); or considering the sign of the authorizations that are in larger number. More recent access control models further extend the authorization tuple allowing the specification of more expressive access control rules, usually based on some logic language. Goal of these proposals is

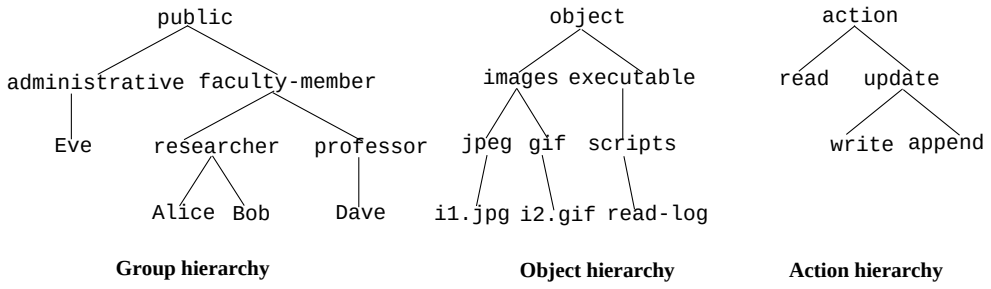

Figure 2: An example of group, object, and action hierarchies

the development of flexible and powerful access control models that provide multiple policy features, that is, that can capture within a single model (and therefore mechanism) different access control and conflict resolution policies.

As their name suggests, discretionary access policies give users discretion in the specification of accesses that can (or cannot) be granted. The specification of authorizations can be regulated by different administrative policies. The most elementary administrative policy is the *centralized* policy in which a central authority (e.g., superuser, database administrator, or security administrator), which can correspond to one or more privileged users, has the privilege of specifying authorizations. Another basic and highly applied administrative policy is the *ownership* policy, in which the creator of an object is considered its owner and as such he/she is granted administrative authority on it. The two approaches can be enriched with decentralized administration, in which the administrator of an object (superuser or owner) can grant to other users the privilege of administering accesses on the objects. The delegated authority can be limited to the specification of access authorizations or it can include administrative authorizations (i.e., the delegated authority can pass on the administrative privilege to others). Decentralized administration introduces the problem of revocation. In particular, the question is what should happen to the privileges granted by a user once his/her administrative privileges are revoked. It is easy to imagine that while there are cases in which we would like these privileges to be deleted as well (as in the case where we are revoking administrative privileges because we do not trust the user anymore), there are other cases in which these privileges should be maintained (as in the case where we are revoking administrative privileges since the user has been promoted and would like to retain all his/her administrative work). Different revocation strategies have been proposed, which include recursive (or *cascade*) and non recursive revocation. In recursive revocation, authorizations granted by the revokee are recursively deleted. In the non recursive revocation approach, either the revoke operation can be rejected if "pending" authorizations (those granted by the revokee) would remain or it can be enforced giving the revoker authority over these authorizations.

3.1.1 DAC Mechanisms

A simple way to represent a set of authorizations for their enforcement consists in using an *access control matrix*. An access control matrix has authorization subjects represented on the rows, and protected objects on the columns. Entry (s, o) in the matrix reports the actions that subject s can exercise on object o. Figure 3 illustrates an example of access matrix. Since the access matrix is usually large and sparse, its storage implies a waste of memory space. There are three basic approaches of implementing the access matrix in a practical way:

- *Authorization table* Store a table of non-null triples of the form (s, a, o). It is especially used in database management systems (DBMSs), where authorizations are stored as catalogs.
- *Access control lists* (ACLs) Each object is associated with an ACL which specifies which users have which access modes on it.
- *Capability lists* (tickets) Each user is associated with a capability lists that specifies the objects that the user can access and the access modes the user can exercise on them.

	fileA	fileB	programC
Ann	own write read	write	execute
Bob	read	read	execute read write
Eve	read	read	

Figure 3: An example of access matrix

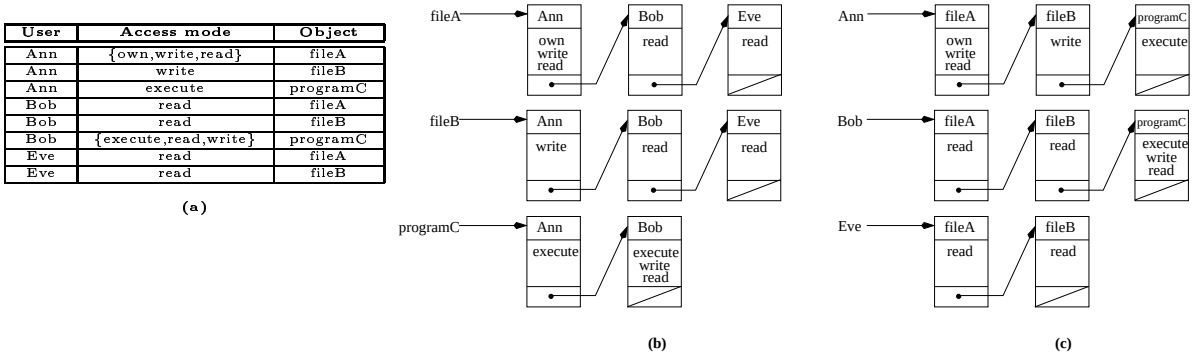

Figure 4: Authorization table (a), access control lists (b), and capability lists (c) corresponding to the access matrix in Figure 3

Intuitively, an entry in the authorization table corresponds to a cell in the matrix, an ACL corresponds to a column of the matrix, and a capability corresponds to a row of the matrix. Figures 4 illustrates the authorization table, ACLs, and capability lists corresponding to the access matrix in Figure 3. ACLs and capabilities have dual advantages and disadvantages: the ACL approach provides efficient per-object access, while the capability approach provides efficient per-subject access. In particular, in the ACL approach, by looking at an object's ACL, it is easy to determine which actions subjects are currently authorized for that object. Determining all accesses for which a subject is authorized would require instead the examination of all the ACLs. Conversely, in a capability based approach it is easy to review all accesses that a subject is authorized to perform, by simply examining the subject's capability list. However, determination of all subjects who can access a particular object requires examination of each and every subject's capability list. A number of capability-based computer systems were developed in the 1970s, but did not prove to be commercially successful. Modern operating systems typically take the ACL-based approach. An example of primitive form of ACL is that implemented in the UNIX operating system, where each file is associated with a list of bits. Access privileges are represented as 9 bits where: bits 1 through 3 reflect the privileges of the file owner, bits 4 through 6 those of the user group to which the file belongs, and bits 7 through 9 those of all the other users. The three bits correspond to the read, write, and execute privilege, respectively. For instance, a list *rwxr-x–x* associated with a file states that the file can be read, written, and executed by its owner, read and executed by the users belonging to the group associated with the file, and executed by all the other users.

3.1.2 Weaknesses of discretionary access control

Discretionary access control policies restrict access to objects based only on the identity of users who are requesting access. Although each access request is originated by users for the purpose of performing some action, it is useful, for a better understanding of the access control problem, to make a distinction between *user* and *subject*. A user is a "passive" entity for whom authorizations are defined and who can connect to the system. A subject is an "active" process operating on behalf of a user that accesses system resources, while users are trusted, subjects that operate on their behalf are not. Discretionary access control policies make

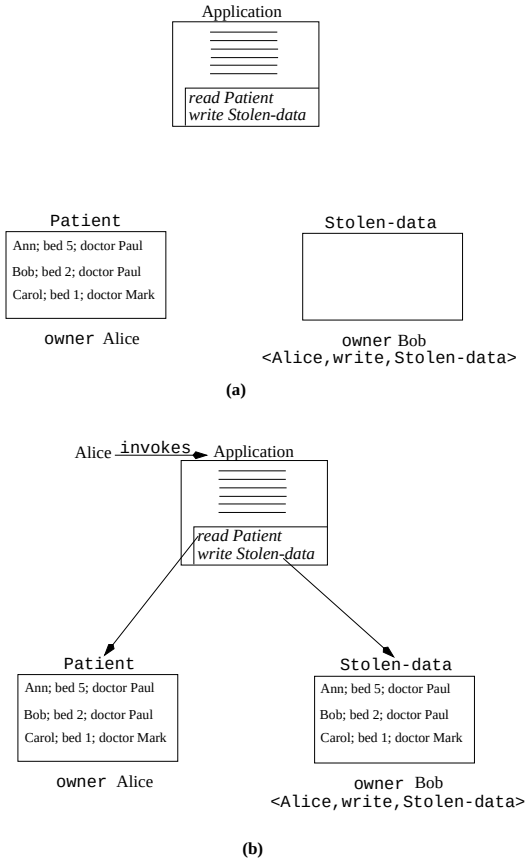

Figure 5: An example of Trojan horse

no distinction between these two concepts: any process (subject) running on behalf of a user can exercise the access privileges granted to the user. DAC policies are vulnerable to *Trojan horses* (i.e., a software containing hidden malicious code) improperly leaking information to unauthorized users. The reason for this weakness is that no control is enforced on the use or dissemination of the information once this information is released to a process. It is thus possible for a process to write information into objects accessible by users not authorized to access the objects from which the information has been read. To see how a Trojan horse can be used to leak information despite the enforcement of discretionary access controls, consider the following example. A user Alice (the victim) creates a file, called `Patient`, and writes sensitive data in it. Alice is the only one allowed to access the file; no one else has any authorization on it. Consider now user Bob who wants to acquire the sensitive information in file `Patient` but is denied access by discretionary access control. To acquire information, Bob creates a file `Stolen-data` and grants Alice the authorization to write it. Bob also writes an appealing application (e.g., an electronic agenda) that provides useful functionalities, but that also has two hidden instructions: a read operation on `Patient`, and a write operation of the read data in `Stolen-data` (see Figure 5(a)). He then gives this application to Alice. Alice ignores both the existence of the two hidden instructions and the existence of `Stolen-data` (and her privileges on it). Consider now the execution of the program by Alice. The electronic agenda process activated by Alice, acquires Alice's privileges. Consequently, the two hidden operations will be successfully granted (Alice has authorizations for them) and information will be copied from file `Patient` to file `Stolen-data`, as illustrated in Figure 5(b). Bob can now access his file and read the information he is not authorized to access directly. All this happens without Alice even knowing.

Improper information leakages such as the one discussed can be prevented by controlling, besides direct accesses, also information flow within the execution of processes, as done in mandatory access control policies discussed next.

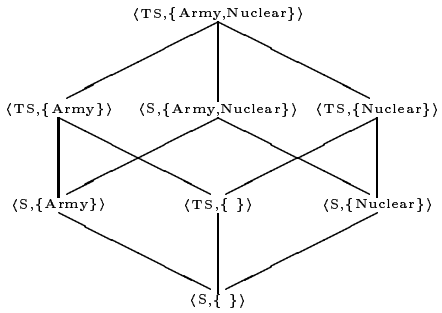

Figure 6: Example of security lattice

3.2 Mandatory Access Control

The Trusted Computer System Evaluation Criteria (TCSEC) defines mandatory access control as follows:

> "A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects access information of such sensitivity."

Mandatory access control policies distinguish between users and processes operating on their behalf (subjects) and are based on the classification of subjects and objects in the system. All users and resources are assigned security classifications. Processes (subjects) activated by a user take on the classification with which the user connected to the system. Classifications are elements of a partially ordered set and may reflect secrecy or integrity labels associated with subjects and/or objects. Usually, partially ordered access classes are modeled as pairs the form (l, c), where l is a security level of a hierarchical ordered set, and c is a set of non-hierarchical categories. Example of secrecy-based security levels are TopSecret (TS), Secret (S), Confidential (C), and Unclassified (U), where TS > S > C > U. Categories allow the definition of areas of competence, and enforce the need-to-know principle. Examples of categories can be: `Nato`, `Army`, and `Nuclear`. Partial order is defined on such pairs as a *dominance relation* $\succeq$ as follows. An access class (l, c) dominates another access class (l', c'), denoted $(l, c) \succeq (l', c')$, if and only if l is at least as high in the hierarchical ordered set as l' (i.e., $l \geq l'$) and $c' \subseteq c$. For instance, access class (S,{`Army`,`Nuclear`}) dominates access class (S,{`Army`}) because they have the same security level and {`Army`} is a subset of {`Army`,`Nuclear`}. The set of access classes with the corresponding dominance relation form a *security lattice*. Figure 6 illustrates an example of lattice for classes over two levels (TopSecret and Secret) and two categories (Army and Nuclear).

MAC policies can be divided in two classes: the most popular *secrecy-based*, controlling data confidentiality, and *integrity-based*, controlling data integrity.

Secrecy-based MAC policies In secrecy-based MAC policies each user and object in the system is assigned a classification. The security class assigned to a user (also called user's *clearance*) reflects the user's trustworthiness not to disclose sensitive information to individuals who do not hold appropriate clearance. The secrecy class assigned to an object reflects the sensitivity of information contained in the object and the potential damage that could result from its improper leakage. Users can log into the system at any security class dominated by their clearance. Processes activated by a user are subjects that take on the security class with which the user connected. Access is regulated by two basic principles, originally formulated by Bell and LaPadula:

- *No read-up* A subject can read only objects classified at the subject's access class or below.
- *No write-down* A subject can write only objects classified at the subject's access class or above.

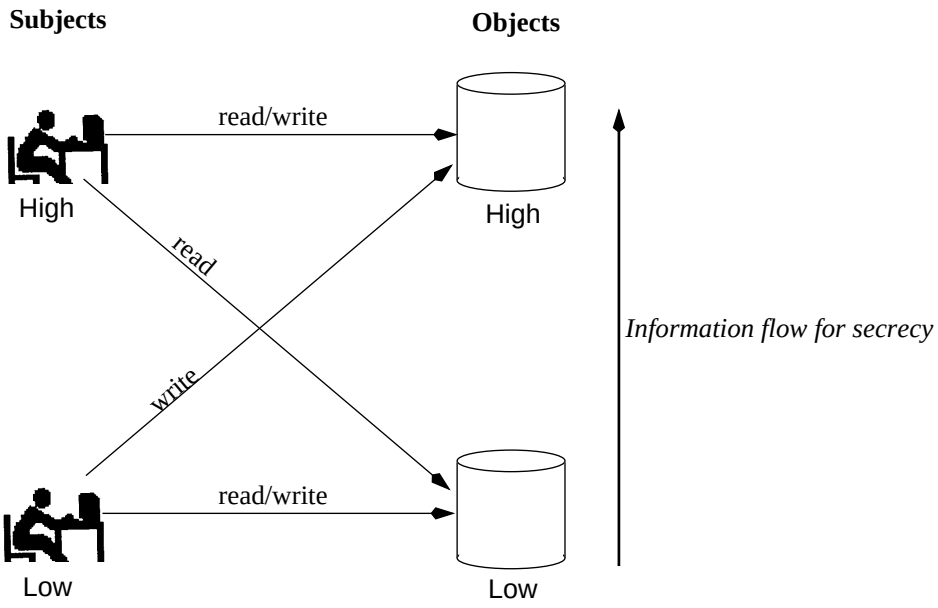

Figure 7: Information flow for secrecy

Satisfaction of these principles prevents information stored in high level objects to flow to objects at lower or incomparable levels (see Figure 7); thus blocking possible information flow making data accessible to subjects not allowed to access them directly. To illustrate, consider the Trojan horse example described in the previous section. Since Bob cannot access `Patient`, his security class (and that of his objects) will be lower or incomparable to that of Alice and file `Patient`. Note also that Bob does not have discretion of granting privileges to Alice. Let us assume that Alice and `Patient` are classified High while Bob and `Stolen-data` are classified Low. Consider again the execution of the Trojan horse application by Alice. Assume Alice is connected as a High subject. The write operation to `Stolen-data` requested by the application will be rejected since it does not satisfy the no-write-down principle. Assume instead Alice connects at level Low. In this case, the read operation on object `Patient`, which does not satisfy the no-read-up principle, will be rejected. In both cases the improper information flow is blocked.

Integrity-based policy Secrecy mandatory policies control only improper leakage of information; they do not safeguard integrity. Integrity can be controlled in a dual way. Again, security classification (integrity class) are assigned to subjects and objects. Examples of integrity levels are: Crucial (C), Important (I), and Unknown (U). Integrity classes assigned to users reflect user's trustworthiness not to improperly modify sensitive information; the integrity classes assigned to objects reflect the degree of trust in information contained in the objects and the potential damage that could result from its improper modification/deletion. Access control is then performed according to the following principles:

- *No read-down* A subject can read only objects classified at the subject's access class or above.
- *No write-up* A subject can write only objects at the subject's access class or below.

Satisfaction of these principles prevents information stored in low level objects to flow to high or incomparable objects (see Figure 8) and therefore to corrupt them.

Note that secrecy and integrity policies can coexist *but* two different classes (one for secrecy and one for integrity) must be maintained in this case. Mandatory policies enforce stricter control on the flow of information than discretionary policies. However, they can be applied only to environments where it is possible to classify information and the better security gained is to be preferred over the loss of flexibility (due to the nondiscretionality of the access regulations). Mandatory and discretionary policies can be applied jointly. In this scenario an access will be granted if both it satisfies the MAC restrictions and there is a

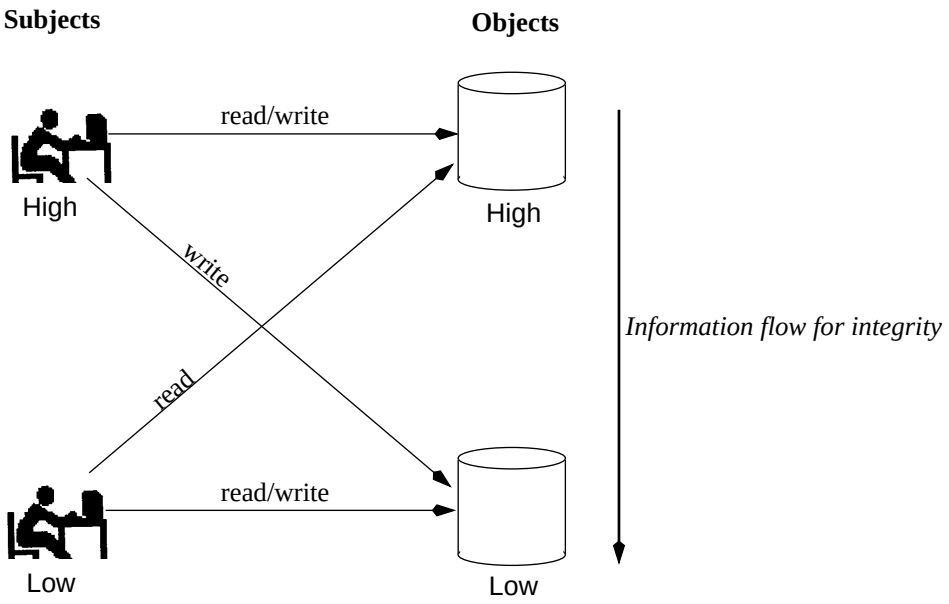

Figure 8: Information flow for integrity

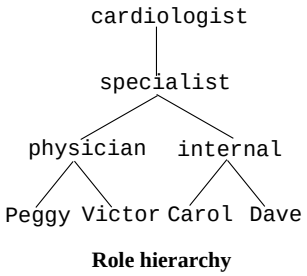

Figure 9: An example of role hierarchy

discretionary authorization for it. In other words, discretionary authorizations operate within the boundaries of the mandatory policy: they can only restrict the set of accesses that will be granted.

3.3 Role-based access control

A third class of policies considered in more recent access control models is represented by role-based policies. A role-based access control (RBAC) policy bases access control decisions on the *roles* a user takes on when executing activities in an organization. For instance, the roles a user associated with a hospital can assume may include `physician`, `nurse`, `researcher`, and `pharmacist`. Roles in a bank may include `teller`, `loan officer`, and `accountant`. Intuitively, access privileges by users to objects are mediated by roles. Authorizations to access objects are assigned to roles, not to users anymore. Users are given authorizations to activate roles. The operations that a user is permitted to perform are based on the user's currently active roles. For instance, within a hospital system the role of `physician` can be defined to which authorizations to perform diagnosis and prescribe drugs are granted. Users authorized to activate the role will, upon activation, be able to exercise such privileges. The fact that privileges associated with a role can be exercised only when the role is active provide enforcement of the least privilege principle, according to which a subject is authorized only for the privileges it needs to perform its job. Role-based access control have also advantages in terms of access rule management. For instance, when a user leaves the organization or is promoted it is sufficient to change his/her roles and reassigning them to his/her substitute (instead of changing all the

involved access authorizations). In many applications there is a natural hierarchy of roles, based on the familiar principles of generalization and specialization. For instance, a `specialist` role may be specialized into `physician` and `intern`. In turn, role `specialist` may be a specialization of a more general role like `cardiologist` (see Figure 9). The role hierarchy has implication on role activation and access privileges: a specialized role inherits the authorizations of its generalizations; also, users authorized to activate a role inherit the authorization to activate the generalizations of the role. For instance, with reference to Figure 9, role `intern` inherits all authorizations of role `specialist`. Also, users authorized to activate role `intern` will also be allowed to activate role `specialist`. Although groups (discussed in the previous section) and roles may seem similar, they capture two different concepts: groups define sets of users while roles define sets of privileges. Also, roles can be activated or deactivated by users depending on their needs while group membership always applies (a user cannot disable his/her membership in groups at his/her discretion).

The process of assigning roles to users can be enriched with the support of constraints. For instance, mutual exclusion constraints can restrict role assignment so that users will not be authorized for roles that are considered in conflict with one another (e.g., author and referee for a same paper) or that would grant the user too much privileges (e.g., accountant and supervisor). The set of privileges granted to roles and to users must obey the separation of duty principle, according to which no single user or role should be granted enough power to misuse the system. Separation of duty can be enforced statically or dynamically. In the static approach, separation of duty constraints are accounted for in the authorizations assignment (no user/role will be authorized for operations that are in a separation of duty constraint). In the dynamic separation of duty, constraints are enforced at run-time: users/roles can be authorized for any access but executing one operation will rule out their ability to execute any other operation which is in a separation of duty constraints with it.

3.4 Inference controls

Access control systems described in the previous section intercept every request submitted to the system and determine whether the request can or cannot be granted according to some specified access control rules. As already discussed, direct control of every single request has limitations, which are addressed in mandatory policies by restricting information flow. There is another threats to data confidentiality which the policies discussed above do not address, which is *inference*. Inference refers to the ability to withdraw information about some data by observing other data. The consideration of inference implies that security restrictions should take into account not only the data directly released, but also those data that can be withdrawn from those released. If we do not, users not authorized to access some data may be able to infer them from data that we release to them. For instance, in a health care environment specific prescriptions can be symptomatic of specific illnesses. By knowing the prescriptions of a patient, a user can then infer the illness from which the patient suffers. Consequently, if we do not want the user to know the illness, we should not release to that user prescription data. Inference usually exploits relationships between data. In addition, inference problems may be due the contributions of released data with to external knowledge that the user has available. Inference due to the external knowledge is clearly more difficult to control since it is not usually possible to know what other information users may know. Even inferences due to the relationships between data are far from being trivial to control. Inference strategies by which users can indirectly acquire information include:

- *Inference by deductive reasoning* New information can be inferred either through classical or non-classical logic deduction. Classical deductions are based on basic rules of logic (e.g., from assertions "A is true" and if "A then B" it is possible to deduce that "B is true"). Non-classical logic-based deduction includes, for example, probabilistic reasoning, fuzzy reasoning, non-monotonic reasoning, and modal logic-based reasoning.

- *Inference by analogical reasoning* Statements such as "X is like Y" can be exploited to infer properties of X given the properties of Y.

- *Inferred existence* A user can infer the existence of a data element from certain information (e.g., from the information, "Alice is a physician" one can infer that there is some entity called Alice).

- *Statistical inference* Information about a data element is inferred from various statistics compared on a set of data elements.

Inference of a set of data elements can be *exact* or *inexact*. It is exact when knowledge of the values in a set A of data elements allows one to derive the values of another set B of data elements. It is inexact when knowledge of the values in A allows one to reduce the values of B to a subset of possible values. For instance, there may exist exact inference from the pair (`rank`,`position`) to `salary`, meaning that knowing the value for `rank` and `position` allows users to infer the value of `salary`. The inference is inexact if knowing the values of `rank` and `position` allows the user to restrict the range of possible values for `salary`.

Inference controls can be best modeled within mandatory security policies, where the labels assigned to the data reflects their sensitivy. In this context, there is an inference problem if high level data can be inferred from low level data. The earliest formal characterization of the inference problem is that of Goguen and Meseguer. According to their definition, the inference problem in mandatory contexts can be stated as follows. Given two data elements A and B, there exists an *inference problem* if and only if from A it is possible to infer B and the classification of A is lower than the classification of B. For instance, suppose that the classification of `rank` and `position` is Low and the classification of `salary` is High. It is clear to see that, even in the respect of the no-read-up principles, users cleared Low, and therefore not allowed to see attribute `salary` can indeed know it by requesting the values of `rank` and `position`, which they are cleared to see. To block such improper release, the mandatory policy should be enriched to take inference channels into account. Inference related constraints can be taken into account either in the *database design* or at *query processing time*. Approaches to inference analysis during database design locate all inference channels through an analysis of the database schema and all the complex relationships between data elements of the application domain. Channels leaking information at higher or incomparable levels are blocked by upgrading selected schema components or redesigning the schema. With reference to the `salary` example, this solution may imply that either `rank` or `position` will be classified as High. In this way, subjects labeled Low will not be able to access *both* `position` and `rank` and infer the value of `salary`. Data upgrading to prevent inference channels requires a complete examination of all the possible inference channels and is far from being trivial. A further complication is due to the necessity of ensuring non overclassification of data. In other words, only upgrades strictly necessary to prevent improper channels should be enforced. Overclassifying data would unnecessarily restrict data visibility. For instance, with reference to our example, the inference channel could be blocked by upgrading both attribute `rank` and attribute `position` but this is not necessary as upgrading only one attribute gives the sufficient protection. Determining an optimal solution to an upgrading problem can be a NP-hard problem. Inference controls at query time leave data classification invaried. However, they keep track of the information released to subjects and block data release (even if mandatory axioms are satisfied) if a subject would acquire enough data to withdraw inference above its clearance level. Again, with reference to our example, this would mean that a High subject can be returned either `rank` or `position` but once it has been returned one it will not be granted access to the other. Run-time controls can be useful in cases where data upgrading is not possible. However, its enforcement has some complications. In particular, it requires keeping the history of requests and determine inferences that can be withdrawn from them, which is a complex and expensive process. Also, determining inference channels simply by looking at subjects (and therefore controlling information released at a given level) may be limiting since the fact that a user has acquired some data would rule out the possibility for other, possibly unrelated, users operating at the same level to acquire other data.

4 Audit

Authentication and access control do not guarantee complete security. Indeed, unauthorized or improper uses of the system can still occur. The reasons for this are various. First, security mechanisms, like any other software or hardware mechanism, can suffer from flaws that make them vulnerable to attacks. Second, security mechanisms have a cost, both monetary and in loss of system's performances. The more protection to reduce accidental and deliberate violations is implemented, the higher the cost of the system will be. For this reason, often organizations prefer to adopt less secure mechanisms, which have little impact on the system's performance, with respect to more reliable mechanisms, that would introduce overhead processing.

Third, authorized users may misuse their privileges. This last aspect is definitely not the least, as misuse of privileges by internal users is one of the major causes of security violations.

This scenario raises the need for *audit control*. Audit provides a post facto evaluation of the requests and the accesses occurred to determine whether violations have occurred or have been attempted. To detect possible violations, all the user requests and activities are registered in an *audit trail* (or *log*), for their later examination. An audit trail is a set of records of *computer events*, where a computer event is any action that happens on a computer system (e.g., logging into a system, executing a program, and opening a file). A computer system may have more than one audit trails, each devoted to a particular type of activity. The kind and format of data stored in an audit trail may vary from system to system, however, the information which should be recorded for each event include: the subject making the request, the object to which access is requested, the operation requested, the time and location at which the operation was requested, the response of the access control, and the amount of the resources used. An audit trail is generated by an *auditing system* that monitors system activities. Audit trails have many uses in the computer security :

- *Individual Accountability* An individual's actions are tracked in an audit trail making users personally accountable for their actions. Accountability may have a deterrent effect, as users are less likely to behave improperly if they know that their activities are being monitored.
- *Reconstructing Events* Audit trails can be used to reconstruct events after a problem has occurred. The amount of damage that occurred with an incident can be assessed by reviewing audit trails of system activity to pinpoint how, when, and why the incident occurred.
- *Monitoring* Audit trails may also be used as on-line tools to help monitoring problems as they occur. Such real time monitoring helps in detecting problems like disk failures, over utilization of system resources, or network outages.
- *Intrusion Detection* Audit trails can be used to identify attempts to penetrate a system and gain unauthorized access.

It is easy to see that auditing is not a simple task, also due to the huge amount of data to be examined and to the fact that it is not always clear how violations are reflected in the users' or system's behaviors. Recent research has focused on the development of automated tools to help audit controls. In particular, a class of automated tools is represented by the so called *intrusion detection systems*, whose purpose is to automate the data acquisition and their analysis. Among the issues to be addressed in data acquisition and analysis are:

- *Audit data retention* If the audit control is based on history information, then audit records already examined must be maintained. However, to avoid the "history log" to grow indefinitely, pruning operations should be executed removing records that do not need to be considered further.
- *Audit level* Different approaches can be taken with respect to the level of events to be recorded. For instance, events can be recorded at the command level, at the level of each system call, at the application level, and so on. Each approach has some advantages and disadvantages, represented by the violations that can be detected and by the complexity and volume of audit records that have to be stored, respectively.
- *Recording time* Different approaches can be taken with respect to the time at which the audit records are to be recorded. For instance, accesses can be recorded at the time they are requested or at the time they are completed. The first approach provides a quick response to possible violations, the second provides more complete information for analysis.
- *Events monitored* Audit analysis can be performed on any event or on specific events such as the events regarding a particular subject, object, operation, or occurring at a particular time or in a particular situation.
- *Audit control execution time* Different approaches can be taken with respect to the time at which the audit control should be executed.

- *Audit control mechanism location* The intrusion detection system and the monitored system may reside on the same machine or on different machines. Placing the audit control mechanism on a different machines has advantages both in terms of performances (audit control do not interfere with normal system operation) and security, as the audit mechanism will not be affected from violations to the system under control.

5 Cryptography

The word cryptography comes from the Greek words `kryptos`, which means "hidden", and `logos`, which means "word". Cryptography is essentially a technique for protecting information and is used in many aspects of computer security. For instance, cryptography is useful to store sensitive information (e.g., password) in a way that it is unintelligible to unauthorized users or to preserve the confidentiality and integrity of a communication in the presence of an adversary. Cryptography is traditionally associated with keeping data secret (confidentiality). As we will see, however, this is only one part of today's cryptography. One of the best ways to obtain data confidentiality is through the use of *encryption*. Encryption transforms data in user or machine readable form, called *plaintext*, to an unintelligible form, called *ciphertext*. The conversion of plaintext to ciphertext is controlled by an electronic key k. The key is simply a binary string which determines the effect of the encryption function. The reverse process of transforming the ciphertext back into the plaintext is called *decryption*, and is controlled by a related key k^{-1}. The transformation rule used to encrypt and decrypt messages is called *cryptographic algorithm* or *cipher*. The concept of securing messages through cryptography has a long history that can be divided into three main stages. In the first stage, cryptography was viewed as an *art*. Early cryptographic algorithms often took the form of *substitution ciphers*, where each character in the plaintext was substituted for another character in the ciphertext; or *transposition ciphers*, where a plaintext is partitioned into fixed-size blocks and the letters within each block are arranged according to some permutation. One of the most famous and simplest example of substitution cipher is the *Caesar cipher*, which is said to have been used by Julius Caesar to communicate with his army. The Caesar cipher transforms a message by replacing each letter with the letter appearing three positions after it in the alphabet, wrapping around at the end. For instance, using the Caesar cipher, the message "return to rome" would be encrypted as "uhwxua wr urph". A simple example of transposition cipher is the *rail fence* cipher. This cipher first writes the message in two different rows, alternating letters in the two rows. The ciphertext is obtained by reading the first row and then the second row. For instance, the message "return to rome" is first written as:

```
r   t   r   t   r   m
  e   u   n   o   o   e
```

The corresponding ciphertext is then "rtrtrm eunooe". The rail fence is a transposition cipher since the characters in the ciphertext are the ones appearing in the original text, but they appear in a different position. This transposition cipher can be generalized by writing a message in n different rows, alternating letters in these rows. The ciphertext is then obtained by reading each row, from top to the bottom. Substitution and transposition ciphers are simple but weak. Substitution ciphers can be easily broken by analyzing single-letter frequency distribution by mapping the letters in the ciphertext to letters in the corresponding position in the frequency distribution of the language (e.g., the letter appearing most frequently is translated to the letter with the highest frequency in the language). Assuming the language (and the letter frequency distribution) with which the original text was written is known, if the message is reasonably long, frequency distribution attacks are likely to succeed. Transposition systems are easy to identify. Their single-letter frequency distribution will necessarily look like the one for a plaintext since the same letters are still present. However, identifying which type of transposition is used is much more difficult at first, and it may be necessary to try different possibilities. However, when the type of transposition has been detected, simple attacks can be enforced to break the cipher. For instance, when simple columnar transposition ciphers are used (the plaintext is inserted into a matrix of a predetermined width and the ciphertext is obtained by extracting the letters in each column, from left to right), their security depends only on the matrix width. Thus, to solve a message enciphered by simple columnar transposition it is necessary to try different matrix widths until the right one is determined. The second stage of cryptography was that of *cryptographic engines*.

Characteristic	Symmetric-key	Asymmetric-key
number of keys	one key	pair of keys
types of keys	key is secret	one key is secret and one key is public
relative speeds	faster	slower

Table 1: Comparison between symmetric-key and asymmetric-key ciphers

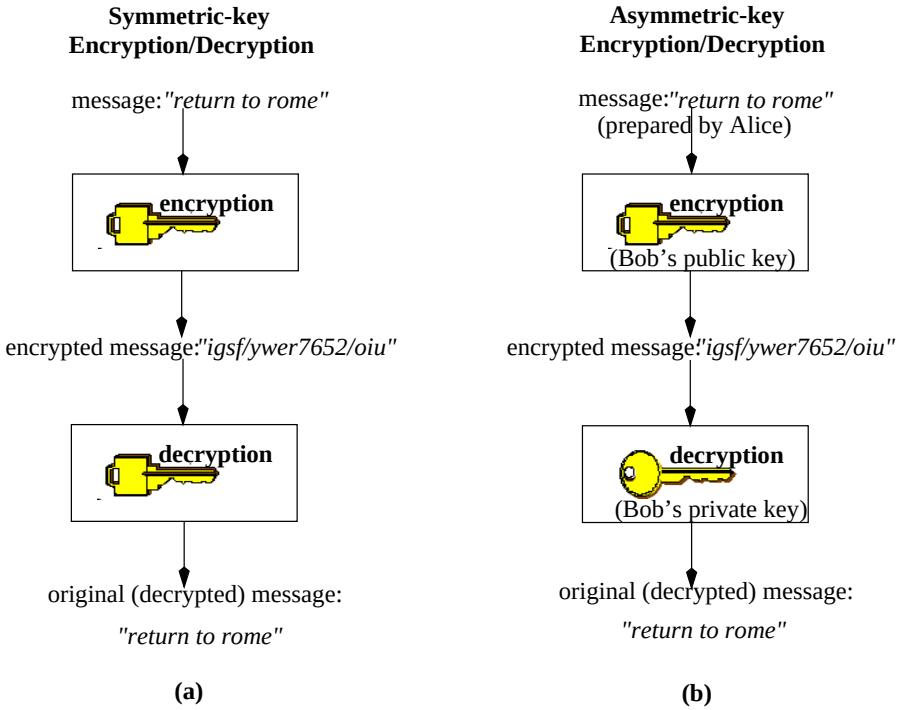

Figure 10: Symmetric (a) and asymmetric (b) encryption/decryption

This stage is associated with the period of the second world war, and the most famous cryptographic engine was the German *Enigma* machine. The basic Enigma was invented in 1918 by Arthur Scherbius in Berlin. The Enigma machine gives a mechanized way of performing one alphabetic substitution cipher after another. The last stage is *modern cryptography* that relies upon advanced mathematics and electronic computers. In modern cryptography, algorithms are complex mathematical formulae and keys are strings of bits; computers are necessary to implement the algorithms. It is this last stage that we will explore. In particular, we will describe fundamental aspects of the basic cryptographic technologies and some specific ways cryptography can be used to improve security.

5.1 Basic cryptographic technologies

Ciphers can be divided in two categories: *symmetric-key* ciphers and *asymmetric-key* ciphers. Table 1 compares some of the distinct characteristics of symmetric-key and asymmetric-key ciphers.

Symmetric-key cipher In a symmetric-key cipher (also called *secret-key cipher*), the same key is used for both encryption and decryption. As shown in Figure 10(a), the sender uses the key to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key to decrypt the message and recover the plaintext. Symmetric-key ciphers can be divided in two classes: *block ciphers* and *stream ciphers*. A block cipher encrypts blocks of data at a time; given a key, the same plaintext block will always be encrypted to the same ciphertext. A stream cipher operates on a single bit, byte, or word at a time,

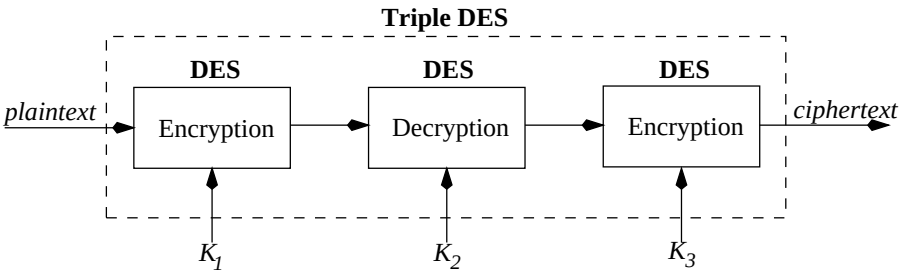

Figure 11: Triple DES

and applies a feedback mechanism so that the same plaintext will yield different ciphertexts every time it is encrypted.

The most famous symmetric cipher is the *Data Encryption Standard* (DES) which was adopted in 1977 by the American National Bureau of Standard (now National Institute of Standard and Technology - NIST) for commercial and unclassified government applications. It was developed by IBM with technical advice from the National Security Agency (NSA). DES is a block cipher using a 56-bit key that operates on 64-bit blocks. DES has a complex set of rules and transformations that were designed specifically to yield fast hardware implementations and slow software implementations. On July 17, 1998, the Electronic Frontier Foundation (EFF) announced the construction of a hardware device that could break DES in an average of 4.5 days. A variant of DES, called *Triple-DES* (3DES) was introduced in 1998 after DES was compromised. Triple-DES uses the Triple Data Encryption Algorithm (TDEA) which employs three secret-key cryptographic steps and one, two, or three keys, denoted $k1$, $k2$, and $k3$, respectively. Generation of the ciphertext c from a block of plaintext p is accomplished by: $c = E_{k3}(D_{k2}(E_{k1}(p)))$, where E_k and D_k denote the encryption and decryption processes, respectively, performed by using key k (see Figure 11).

There are a number of other symmetric-key cryptography algorithms that are also in use today: *International Data Encryption Algorithm* (IDEA), another DES-like 64-bit block cipher using 128-bit keys; *RC2* (Rivest Cipher 2), named from its inventor Ron Rivest, a 64-bit block cipher using keys of variable size; *RC4*, a stream cipher, using keys of variable size, widely used in commercial cryptography products; and *RC5*, a block-cipher supporting a variety of block sizes, key sizes, and number of encryption passes over the data.

In 1997, NIST initiated a process to develop a new secure cipher for U.S. government applications. The result, the *Advanced Encryption Standard* (AES), will be the "official" successor of DES. From the many algorithms submitted, at the end of 1999 a group of five candidate algorithms was selected: *MARS* (multiplication, addition, rotation and substitution) from IBM; Ronald Rivest's *RC6*; *Rijndael* from Belgian researchers Joan Daeman and Vincent Rijmen; *Serpent*, developed jointly by a team from England, Israel, and Norway; and *Twofish*, developed by Bruce Schneier. In October 2000, NIST selected Rijndael as the AES algorithm. Rijndael is a block cipher. The algorithm can use a variable block length and key length: the current specification describes keys with a length of 128, 192, or 256 bits to encrypt blocks of length 128, 192 or 256 bits (all nine combinations of key length and block length are possible). The final approval of Rijndael is expected in July 2001.

Asymmetric-key cipher Asymmetric-key ciphers (also called *public-key ciphers*) have been introduced in 1976 by two Stanford researchers, Whitfield Diffie and Martin Hellman. In an asymmetric-key cipher, the abilities to perform encryption and decryption are separated (see Figure 10(b)). The encryption rule employs a *public key* k, while the decryption rule requires a different (but mathematically related) *private key* k^{-1}. Therefore, in public-key systems, each individual must be associated with a pair keys: a private key known only to the user, and a public key, that can be publicized to others. Messages addressed to a user are encrypted by using the user's public key. The recipient user can decrypt them with the corresponding private key. Since the private key is kept secret, only the intended individual can decrypt the ciphertext. While the public and private keys must be related so that texts ciphered with one key can be deciphered with the other, it must be computationally infeasible to derive the private key from the knowledge of the

public key (given that this is made known). The establishment of the pair of keys exploits mathematical problems that are hard to solve, such as computing discrete logarithms, as in the proposals by Diffie and Hellman, (the proponents of public key cryptography) and by ElGamal, or factoring numbers, as in the RSA algorithm.

The *RSA* algorithm, named from the three MIT mathematicians who developed it (Ronald Rivest, Adi Shamir, and Leonard Adleman) is the most popular asymmetric-key cipher. It is currently used in many software products and can be used for key exchange or encryption of small blocks of data. The key-pair is derived from a very large number n obtained by the product of two prime numbers chosen according to special rules; these primes may be 100 or more digits in length each, yielding a number n with twice as many digits as the prime factors. The steps necessary to create an RSA public/private key pair are the following: (1) choose two prime numbers, p and q, and compute $n = pq$; (2) select a third number e that is relatively prime to $(p-1)(q-1)$; (3) determine a number d such that $ed \equiv 1 \mod (p-1)(q-1)$. The public key is the number pair (n, e), while the private key is the number pair (n, d). A message m is then encrypted by using the equation: $c = m^e \mod n$. The receiver then decrypts the ciphertext c with the private key using the equation: $m = c^d \mod n$. The security of this scheme is related to the mathematical problem of factorization: it is easy to generate two large primes and to multiply them, but given a large number that is the product of two primes, it requires a huge amount of computation to find the two prime factors. The ability for computers to factor large numbers, and therefore attack schemes such as RSA, is rapidly improving and systems today can find the prime factors of numbers with more than 140 digits. The presumed protection of RSA, however, is that users can easily increase the key size to always stay ahead of the computer processing curve.

The main drawbacks of symmetric-key ciphers is that a key must be established between each pair of entities (e.g., users) that wish to communicate in confidence. In applications where a limited number of users exist, symmetric-key cryptography is effective. However, in large networks with users distributed over a wide area, key distribution and maintenance becomes a problem. Each individual in a network should have a distinct key to communicate with each single individual in the system. Therefore, a large number of keys must be established and stored securely. Providing individual to individual secret communication in a system with n users would require $\frac{n(n-1)}{2}$ keys. In a system with 1000 users this would amount to almost half a million keys. Exchanging and managing such a large number of keys is at best a difficult task and at worst impossible.

Among other advantages, asymmetric-key ciphers do not suffer from the explosion of the number of keys to be maintained: only two keys need to be maintained for every individual. Providing secure individual to individual communication in a network with n individuals therefore requires only $2n$ keys. In a network with 1000 users, this would amount to total of 2000 keys, in contrast to the half a million keys required in the symmetric-key approach. In a Public Key Infrastructure, distribution of public keys can be enforced through external authorities, called Certification Authorities, which maintain a database of identities and public keys registered for them.

By comparing symmetric-key and asymmetric-key systems, it may appear that asymmetric-key systems are functionally superior to symmetric-key techniques and that there is little need to consider the latter. However, symmetric-key systems are still widely used because they are able to process data much faster than current asymmetric-key systems. For instance, RSA is about 100 times slower in software and 1.000 to 10.000 times slower in hardware than DES. A common approach is to combine the most attractive features of each system: an asymmetric-key scheme is used to exchange a common secret key, after which a symmetric-key scheme performs data encryption using the common key exchanged. Such a "hybrid" system offers the extra speed that a symmetric-key cipher affords, while employing an asymmetric-key cipher to avoid the key distribution problem.

5.2 Uses of cryptography

The main uses of modern cryptography include: data authentication, user authentication, non-repudiation of origin, and data confidentiality. We look at them in more details in the following. For simplicity, we refer all examples to a hypothetical communication between two users, Alice and Bob, in the presence of an adversary user named Eve.

Data confidentiality Data confidentiality, that is maintaining the secrecy of the message transmitted, can be achieved by encrypting data for their transmission. Encryption can employ either symmetric or asymmetric key systems. In the first case, to communicate a message to Bob, Alice encrypts the message with the secret key shared between them, Bob will decrypt it by using the same key. In the second case, Alice encrypts the message with the public key of Bob, and Bob decrypts it with his private key.

Data authentication Data authentication includes two different aspects: *data integrity* and *data origin authentication*. Authenticating data integrity means guaranteeing that the message has not been unaltered improperly. Authenticating data origin means guaranteeing that the message has really been sent by the claimed sender.

Let us first look at data integrity. Suppose that Alice sends a message to Bob and that the adversary Eve intercepts it. Without the support of data integrity, Eve can just change the message and then relay the corrupted message to Bob. Bob will not see that the message has been changed and will assume Alice states what is written in the message received. While not applicable for prevention, cryptography can effectively *detect* integrity compromises due to intentional or unintentional modifications to the original data. Both symmetric-key and asymmetric-key methods can be used to this purpose. When symmetric-key cryptography is used, a *message authentication code* (MAC) is calculated from the message and appended to it. The MAC is a function of the message and of the secret key shared between the sender and the recipient. That is, MAC=$f(k, m)$, where f is a public function, k is a secret key, and m is the message. The MAC, which should be a function of every bit of the message, works as an integrity checksum. By using the secret key k, the recipient can then verify the integrity of a message m' received simply by computing $f(k, m')$ and matching the result with the MAC appended to the message. If the two do not match, $m' \neq m$, which means that integrity has been compromised. Note that matching is only a necessary condition for integrity as two different messages m and m' could result in a same value of f . The condition can be made sufficient by defining function f so to avoid collisions (there is a collision if two different messages map to a same MAC). The longer the MAC, the less chance there is of a match for a fraudulent message.

Asymmetric key cryptography verifies integrity by using *digital signatures* and *secure hashes*. As the name suggests, digital signatures are the electronic equivalent of traditional handwritten signatures. Handwritten signatures provide security services because each individual has distinct handwriting, making their signature hard to forge. It is important to note that if digital signatures were formed in the same way as written signatures, that is, by simply appending a fixed string to each message that somebody wants to sign, then security would easily be compromised. For instance, to forge Bob's signature, the adversary Eve can duplicate a previous copy of Bob's signature and append it to any message she chooses, claiming that the message was been signed by Bob. This problem arises because, unlike an individual's handwriting, electronic information is easy to duplicate, and is therefore vulnerable from replay attacks (see Section 2). To avoid this, digital signatures are performed in a more complex manner using an asymmetric-key system. The essential difference between the use of an asymmetric-key system for signing and its use for encrypting is that the order in which the keys are used is reversed. To encrypt a message addressed to Bob, Alice uses Bob's public key. If Alice wants instead to sign a message, she uses her own private key. The recipient can verify the signature by using Alice's public key. More precisely, the signature process can be schematized as follows. Suppose that Alice wishes to sign a message m. Alice first transforms m using a *hash function* h. A hash function h is a transformation that takes an input m and returns a fixed-size string, which is called the hash value. The basic requirements for a cryptographic hash function are: (1) $h(x)$ must be relatively easy to compute for any given x; (2) $h(x)$ must be *one-way*, meaning that given a hash value y, it is computationally infeasible to find some input x such that $h(x) = y$; (3) $h(x)$ must be *collision-free*, meaning that it is computationally infeasible to find two messages x and y such that $h(x) = h(y)$ (strongly collision-free). The output of the hash function, denoted $h(m)$, is a value which is specific to the content of the message and is called *message digest*. Alice signs m by transforming $h(m)$ using her private key, to obtain $s = D_{k_{alice}^{-1}}(h(m))$, where k_{alice}^{-1} is Alice's private key. Alice now sends m and s to Bob as her signature on m. If Bob wants to verify Alice's signature on m, he first retrieves Alice's public key k_{alice}. Then, he recomputes the message digest $h(m)$ from m using the publicly available hash function. Finally, Bob transforms s using k_{alice} and compares the result with $h(m)$. If Bob finds that $E_{k_{alice}}(s) = h(m)$, then he accepts Alice's signature as valid. Otherwise, Bob concludes that s cannot be Alice's signature for message m, and that m has been improperly modified.

Data origin authentication guarantees that the person who is claiming to be the sender of a message

really is the one from whom the message originated. For instance, in absence of authentication of origin the adversary Eve could send a message to Bob claiming that the message is from Alice. Data origin authentication is naturally provided in symmetric-key approaches because of the different keys shared by each pair of users: to masquerade as Alice, Eve should know the secret key shared by Alice and Bob. In asymmetric-key approaches, data origin authentication can be provided through the signature process just discussed. Indeed, transformation from $h(m)$ to s (i.e., the signature) can be computed only by knowing the private key of the sender (Alice's private key k_{alice}^{-1} in the example above).

User Authentication When one party tries to communicate with another, it is necessary to verify that the parties involved in the communication are who they say they are. This process is called user authentication. Cryptography is the basis for several advanced authentication methods. As an example, consider the following scenario. Alice needs the user authentication service to be sure that she is involved in a real-time communication with Bob. Suppose Bob attempts to provide this assurance by simply signing the message "this is Bob". Alice is assured that the message originated from Bob at some stage, because, as seen in the previous discussion, digital signatures provide data origin authentication. The problem here is that once Bob has signed "this is Bob ", the adversary Eve can save this signature and use it to authenticate herself as Bob at any later time. Thus, digital signature alone does not provide Alice with user authentication. To obtain this service, the interaction between Alice and Bob can be modified as follows. When Alice initiates a real-time communication with Bob and wants user authentication, she first generates an unpredictable random number r and sends it to Bob; r is called a *challenge* in this context. Now, instead of signing just "this is Bob", Bob signs "this is Bob, r". Provided this signature is valid, and the signed value r is the same as the challenge Alice generated, then Alice is assured that she is communicating with Bob in "real-time". Thus, the combination of a digital signature together with the Alice's random challenge provides user authentication.

Non-repudiation of origin Non-repudiation protects against denial by one of the entities involved in a communication of having participated in all or part of the communication. Non-repudiation with proof of origin protects against any attempts by the sender to repudiate having sent a message, while non-repudiation with proof of delivery protects against any attempt by the recipient to deny having received a message. For instance, suppose that Bob is the owner of a company collecting orders electronically and Alice one of his customer who submitted an order. Non repudiation of origin allows Bob to demonstrate to a third party that the order he received was indeed from Alice. Non repudiation of delivery allows Alice to demonstrate to a third party that Bob indeed received her order. Both non repudiation of origin and non repudiation of delivery require the use of a public key system. Indeed, since in a secret key system Bob and Alice share the same key, any of them can write messages claiming they originated from the counterpart. Again, the signature process described above provides non-repudiation of origin. To provide non-repudiation of origin, Bob has to save Alice's signature s on purchase order po. The presence of the signature proves that the message could have been sent only by somebody knowing Alice's private key (who should be known only to Alice). Public key can also be used to provide non repudiation of delivery by requesting the recipient of a message to provide a receipt for the message signed with his/her own private key.

5.3 Applications of cryptography

In the previous section, we have seen how modern cryptography can be used to provide each of the major security services: confidentiality, user authentication, data origin authentication, data integrity, and non-repudiation. In this section, we mention some current applications of cryptography.

Multi party protocols In a multi party protocol, several parties want to coordinate their activities to achieve some goal. To ensure correctness even in the presence of corrupted parties, the protocol should guarantee that the non-corrupted parties are able to achieve the goal even though the corrupted parties send misleading information. A typical example of secure multi party computation is *electronic elections*. In the general instance of this problem there are m people with their private input x_i, and it is necessary to compute the result of a function f over such values, without revealing them. In the case of electronic elections, the involved parties are the voters, their input is a binary value (e.g., yes or no), the function to

be computed is a simple sum, and the result is the score. The properties that the electronic elections should satisfy are the following: (1) only authorized voters can vote, (2) nobody can vote more than one time, (3) secrecy of votes is preserved, (4) nobody can duplicate the vote of anyone else, (5) the score is computed correctly, (6) anybody should be able to check that the score is computed correctly, (7) the protocol should be fault-tolerant (i.e., it should work even in presence of corrupted parties), and (8) it should be impossible to coerce a voter. A single cryptographic protocol or mechanism alone is not enough to assure all these desired properties. In general, an electronic elections schema relies on different cryptographic techniques like *blind signatures*, *mix-nets*, and *vote-tags*. Blind signature mechanisms allow a party to get a message digitally signed by another party without revealing the signer information about the content of the message. A mix-net is constructed by cascading several *mixes*. The goal of a mix-net is to realize an anonymous channel. The input of a mix-net is a set of messages. The output of a mix-net consists of the same messages but in a distinct order, in such a way that the link between a message and the sender of the message is unknown. A vote-tag is a piece of data that is linked with the vote, but it does not reveal any information on it. There is no a formal definition of vote-tag, however it must satisfy two important properties: given a vote-tag, it has to be difficult, (even better, impossible) to discover the corresponding vote; and a vote has to be bounded to the corresponding vote-tag in such a way that it is not possible to create a different valid vote without modify the vote-tag.

Another example of multi party protocol is the problem of *sharing a secret*. The general model for secret sharing is called a *k-out-of-n* schema (or *(k,n)-threshold* schema), for integers such that $k \leq n$. In the schema, n people $p_1, \ldots, p_n$ share a secret s (e.g., a secret cryptographic key) by dividing it into pieces, called *shares* or *shadows*, in such a way that any subset of k people can recreate the secret from their pieces, but no subset of less than k people can recreate the secret. The first solution to this problem has been given by Shamir in 1979, and is based on polynomial interpolation. The secret is divided into pieces by:

- generating a random polynomial $p(x)$ with degree $k-1$ and constant term s;
- publicly choosing n random distinct evaluation points $x_j \neq 0$ and secretly distributing to each person p_j the share $share_j = p(x_j)$.

The secret s can then be reconstructed from any k shares by using Lagrange interpolation to find the coefficients of the polynomial $p(x)$ (including the secret s that is equal to $p(0)$). Shamir's solution, however, suffers of two problems. First, if one person is dishonest, he can give pieces that when put together do not univocally define a secret. Second, dishonest players, at the reconstruction stage, may provide different pieces than they received and therefore cause an incorrect secret to be recreated. To avoid these drawbacks, Chor, Goldwasser, Micali, and Awerbuchhave proposed a schema based on the intractability of factoring.

Anonymous transactions is another example of multi part protocol. The purpose is to protect individuals from tracing of their transactions. Using *digital pseudonyms*, like those proposed by Chaum, individuals can enter into electronic transactions with assurance that the transactions cannot be later traced to the individuals. However, since the individual is anonymous, the other party may wish to assurance that the individual is authorized to start the transaction, or is able to pay.

Digital cash The electronic payment scenario assumes three kinds of players: a customer, a merchant, and the bank. In this context, the most common type of payments used on-line are credit card payments. The main problems with the credit card payments are:

- *Security*, all user's private information is exposed to the merchants.
- *Anonymity*, all purchases are traceable since the identity of the customer is established every time he/she makes a purchase; in real-life situations, we can use cash whenever we want to buy something without establishing our identity.
- *Small payments*, small amount purchasing with credit card is difficult because of the high transaction cost of credit card.

There are many other different types of cryptographically payment systems (e.g., electronic check and digital cash); we briefly describe the *digital cash* schema. The term digital cash (or *electronic cash*) is applied to

any electronic payment schema that aims to recreate over the Internet the concept of cash-based shopping. Among the properties that these digital cash schemas should satisfy there are: (1) forgery should be hard, (2) duplication should be either prevented or detected, (3) customer's anonymity should be preserved, and (4) the on-line operations on large databases should be minimized. There are two different types of digital cash: *identified* digital cash and *anonymous* digital cash. Identified digital cash contains information revealing the identity of the person who originally withdraw the money from the bank. Anonymous digital cash works just like real paper cash. Once anonymous digital cash is withdrawn from an account, it can be spent or given away without leaving a transaction trail. In addition, digital cash systems can be *on-line* or *off-line*. In on-line systems a transaction between customer and merchant does not take place unless a third party server first verifies the customer's identity or the validity of the customer's digital cash, and authorizes the payment of the good to the merchant. By contrast, in off-line systems involve no third party in the payment from customer to merchant. Off-line anonymous digital cash is the most complex form of digital cash because of the *double-spending* problem. Double-spending refers to fraudulently spending the same money twice. Because digital cash is a set of bits, it can be easily copied. If digital cash can be copied and spent twice, then it can also be copied and spent n-times (multi-spending). To overcome this problem, on-line systems typically keep a record of digital coins that have been spent, and do not authorize transactions involving money that have been previously spent. Obviously, the problem here is that the list of digital coin spent grows over time, which creates issues of storage and access time. By contrast, off-line anonymous digital cash systems frequently rely on exposure as a preventive measure: the anonymous identity of the spender is publicly revealed by double-spending. One way of doing this is to structure the digital coins and cryptographic protocols to reveal the identity of the double spender at the time the digital coin makes it back to the bank. If customers of the off-line system know they will get caught, the incidence of double spending will be minimized (in theory). To make digital cash possible, asymmetric-key ciphers and digital signatures are used. The basic idea is that banks and customers would have public-key encryption keys. Banks and customers use their keys to encrypt (for security) and sign (for identification) blocks of digital data that represent money orders. A bank "signs" money orders using its private key and customers and merchants verify the signed money orders using the bank's widely published public key. Customers sign deposits and withdraws using their private key and the bank uses the customer's public key to verify the signed withdraws and deposits.

Acknowledgments

The work of Sabrina De Capitani di Vimercati and Pierangela Samarati was supported in part by the European Community within the FASTER Project in the Fifth (EC) Framework Programme under contract IST-1999-11791.

References

[1] J.P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, vol. 1 AD-758 206, ESD/AFSC Hanscom, AFB Bedford, Mass, 1972. [This is a historical report whose object is the development of a plan to provide secure multilevel computer systems for the Air Force.]

[2] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P Anderson Co., Fort Washington, PA, April 1980. [This is a historical report that presents a study the purpose of which was to improve the computer security auditing and surveillance capability of the systems.]

[3] S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison Wesley Publishing Company, 1995. [This book provides a comprehensive discussion about security issues in database systems and shows how systems can be designed to ensure both integrity and confidentiality.]

[4] D. Chaum. Blind Signatures for Untraceable Payments. In *CRYPTO 82*, pages 199–204, 1982. [This paper introduces the concept of blind signature. David Chaum is the father of blind signatures and in particular of anonymous digital cash.]

[5] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *26th IEEE Symposium on Foundations of Computer Science*, pages 383–395, Portland, 1985. [This paper presents a schema for the problem of sharing a secret which allows for secure distributed computations against Byzantine faults.]

[6] S. Dawson, S. De Capitani di Vimercati, P. Lincoln, and P. Samarati. Minimal Data Upgrading to Prevent Inference and Association Attacks. in *Proc. of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, Philadelphia, PA, May 31-June 3, 1999. [This paper presents an approach to compute minimal data upgrading to combat unlegitimate data leakages due to inference and association channels.]

[7] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transaction on Information Theory*, 22(6):644–654, November 1976. [This ground-breaking paper presents the Diffie-Hellman key agreement protocol. The protocol, based on the discrete logarithm problem, allows two users to exchange a secret key over an insecure network without any prior secrets.]

[8] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme based on Discrete Logarithms. *IEEE Transaction on Information Theory*, 31(4):469–472, 1985. [This paper presents a public-key encryption and signature schemes. The proposed schema depends on the discrete logarithm problem for its security.]

[9] J.A Goguen and J. Meseguer. Unwinding and Inference Control. In *Proc. of the 1984 Symposium on Security and Privacy*, pages 75–86, 1984. [This paper addresses the noninterference problem. The proposed security model has been the basis for later work elsewhere, including restrictiveness (nondeterministic noninterference).]

[10] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, rfc 2459, January 1999. http://www.ietf.org/rfc/rfc2459.txt. [This Internet draft is the product of the Public-Key Infrastructure (X.509) Working Group. It specifies an Internet profile for the use of X.509 certificates.]

[11] S. Jajodia and C. Meadows. Inference Problems in Multilevel Secure Database Management Systems. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security - An Integrated Collection of Essays*, pages 570–584. IEEE Computer Society Press, 1995. [This paper discusses inference and aggregation problems in multilevel databases. It presents some techniques that has been developed to solve some inference problems and tools developed for them.]

[12] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. A Unified Framework for Supporting Multiple Access Control Policies. *ACM Transactions on Database Systems.* To appear. [This paper presents a logic-based authorization model supporting multiple access control policies.]

[13] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World.* PTR Prentice Hall, 1995. [This book presents network security protocols and mechanisms. It explains the cryptographic algorithms on which most security systems depend, includes a description of the Kerberos authentication system used in UNIX networks, ans also describes secure electronic mail standards.]

[14] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, 1996. [This book presents practical aspects of conventional and public-key cryptography and offers information on the latest techniques and algorithms in the cryptographic field.]

[15] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria.* National Computer Security Center (NCSC), 1985. DOD 5200.28-STD. [This document defines the trusted computer system evaluation criteria that provide a basis for the evaluation of effectiveness of security controls built into automatic data processing system products.]

[16] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of ACM*, 21(2):120–126, 1978. [This is the paper where Rivest, Shamir, and Adleman described the famous RSA asymmetric-key cipher.]

[17] P. Samarati and S. Jajodia. Data Security. In J.G. Wenster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1999. [This paper presents basic aspects about data security. It focuses on the authentication, access control, auditing, and encryption security services.]

[18] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996. [This paper provides a concise discussion of various forms of Role-Based Access Control.]

[19] R.S. Sandhu and P. Samarati. Authentication, Access Control and Intrusion Detection. In A. Tucker, editor, *CRC Handbook of Computer Science and Engineering*, pages 1929–1948. CRC Press Inc., 1997. [This paper presents an exhaustive description of three basic security services: Authentication, Access Control, and Intrusion Detection.]

[20] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 1994. [This book is a good introduction to the cryptography field. It focuses on four main subjects: protocols, algorithms, source code (in C), and politics.]

[21] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979. [This paper presents the classic 1979 result of Shamir to solve the problem of sharing a secret.]

[22] M. Shand and J. Vuillemin. Fast Implementations of RSA Cryptography. In *Proc. of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Los Alamitos, CA, 1993. [This paper analyzes some techniques which may be combined in the design of fast hardware for RSA cryptography.]

[23] D.R. Stinson. *Cryptography: Theory and Practice*. CRC Press, Boca Raton, Florida, 1995. [This book covers all the major areas of cryptography. It also deals with some active areas of research and gives the reader an introduction and overview of the basic results in the area.]