Information Security FS2010 [1]

Stefan Heule

August 20, 2010

Contents

Part I
Ueli Maurer

1 Introduction and motivation

1.1 Basic definitions

1.1.1 Terminology

- **Entity.** Generally refers to a person or organization that acts, usually with its own will and directive, in the considered context. Sometimes also used for technical systems such as computer systems or processes.
- **User.** An entity that is a person
- **System.** Component of an overall system, which is treated
- **Channel.** A system that transports messages between two designated endpoints. A channel might also be accessible by other entities, e.g. an eavesdropper.
- **Attacker** or **adversary.** An entity acting maliciously, trying to obtain an illegitimate advantage or do some type of harm to other entities.
- **Threat.** Any type of general risk of unwanted outside influence on a system. A first course distinction divides threats into two types:
  - **Intentional threats.** Threats origination from an intelligent adversary, often targeted at the weakest point of a system.
  - **Accidental threats.** Threats due to unforeseen events, system errors, or unexpected (but non-malicious) user behavior. Example are power supply outages, fires, or (unintentional) software bugs.
- **Attack.** A malicious act by an attacker. A concrete implementation of an intentional threat. Examples include:
  - Hacking into user accounts, or more generally, obtaining unauthorized access to a system.
  - Unauthorized access to, modification, or deletion of data.
  - Wire-tapping a phone line or LAN cable.
  - Denial (or repudiation) of a performed transaction.
  - Denial-of-service (DoS) attack.
- **Vulnerability** or **weakness.** Unintentional system features, due to design, specification, or implementation errors. Vulnerabilities cause threats, namely that they are exploited by an attacker.

1.1.2 Basic security objectives

Often, the following three basic security goals (known as "CIA") are identified:

- **Confidentiality** or **secrecy.** Information should be accessible only to those authorized to see it. For example, the goal of a network sniffing attack is to violate confidentiality.
- **Integrity.** Information should not be modified without appropriate authorisation. For example, the state of a database or an on-line banking system should be changed only by legitimate users.
- **Availability.** Information and systems should be available when needed. For example, the goal of a denial-of-service attack is to violate availability.

While useful as a guideline, this classification is too simplistic, as it does not cover a number of security goals discussed below. Also, in the above classification, the term integrity is understood in a very genenral sense so as to include:

- **Authenticity.** Typically used in the sense that a message or other piece of information originates from the claimed entity.

Some more advanced and more complex security objectives are:

- **Non-repudiation.** Impossibility to inappropriately deny a transaction or having sent a message.
- **Auditability.** Ability to reconstruct (certain aspects of) earlier states of a system.
- **Accountability.** Ability to hold an entity accountable for its action. This is related to both non-repudiation and auditability.
- **Privacy.** There is no clear definition for the term privacy, but it basically refers to the security of *personal* information. Privacy means that a person has appropriate control over which information on him or her is generated, stored, processed, and deleted, and by whom.
- **Anonymity.** The identity of an entity is hidden. Anonymity is an aspect of privacy.

1.1.3 Classification of security measures

The following types of security measures and techniques can be distinguished:

- **Technical.**
  - Cryptography
  - Physical at macro-level: access to buildings, secured areas, shielding against electromagnetic radiation, etc.
  - Physical at micro-level: tamper-resistant devices, smart-cards, etc.
  - Biometric technology
  - Processor technology
  - Operating system security
- **Organizational.** Security policies, classification of information, defining responsabilties, etc.
- **People-related.** Selection, motivation, education, etc.
- **Legal.** Liability regulations, insurances, etc.

1.2 Digital objects

Definition 1.1: A *digital object* (DO) is a sequence of bits (also called a *bitstring*) which is independent of a physical representation. A digital object can equivalently be interpreted as an integer number.

Examples include 10001010110010110, an e-mail, a program, a virus, or a digital signature. In contrast, a CD, a memory chip or paper document are not digital objects.

Digital object differ radically from phyiscal objects, for instance:

- DO's can be reproduced (copied) without cost.
- DO's can be transported at the speed of light.
- DO's can be destroyed without leaving traces (if only stored locally)
- DO's cannot be revoked (once released)
- DO's are stable and can be stored essentially for eternity.
- DO's have well-defined properties.

1.2.1 The effect of digital objects in the real world

Digital objects are of interest only because of their potential *effect in the real, physical world.* In order to have an effect in the real world, digital objects must be converted by an interface between the digital and the physical world. Some examples of such interfaces are:

- A digital object can be represented as human-readable text, on paper or on a screen. This causes an effect in the real world if the person reading the text makes his or her behavior depend on it.
- Execution of a DO as software on a computer, which interacts with the real world.
- A digital signature verified by a judge who decides that a person is liable (in the real world).

Statement 1.1 (Functionality/security tradeoff dilemma)**:** The more functionality (and power) an interface from digital objects to the physical world provides, the more devastating are potential consequences when malicious digital objects are converted by the interface.

1.2.2 Distinguishing good and bad digital objects

Lemma 1.1: Deciding whether a program meets a given specification is in general undecidable.

Proof. The undecidable halting problem is a special case of deciding whether a program meets a given specification (namely, to halt). □

1.2.3 Some basic problems of the information economy

- **Archiving digital objects.** Archiving does not simply mean to store the digital objects and to provide a well-defined set of access and search operations, but it actually means to archive the *effect* of the DO on the real world. Consequently, this means that one also needs to archive the interface. For as simple as en example as a Word document, this means that one needs to archive the Word program, the operation system, and ultimately, the hardware on which the software can be installed and executed.
- **Selling digital objects.** How can one sell the *effect* of a digital object without showing it? This is a core problem of the content business (music, video, etc.) and in preventing software piracy.
- **Controlling digital objects.** How can one control what others can do with a DO? For instance, how can one restrict to which entities a digital object can be made available, say to protect privacy?

1.3 Three dilemmas

An obvious, but sometimes forgotten fact: Security can be defined only relative to a well-defined specification of the required behavior of a system.

Dilemma 1.1 (Specification complexity dilemma)**:** Typical security specifications are generally not precise, and precise specifications generally do not capture the actual requirements.

Dilemma 1.2 (Functionality/security tradeoff dilemma)**:** In some contexts, functionality and security requirements are mutually exclusive. In other words, the set of *admissible* system behaviors is empty.

Dilemma 1.3 (Implementation dilemma)**:** For some security specifications, a secure system is not implementable.

This last dilemma is a more general form of lemma 1.1 (deciding whether a digital object is good or bad might be undecidable).

1.4 Defining security

1.4.1 Two approaches to defining security

The following two approaches to making precise security definitions can be distinguished, depending on whether one focuses on what the adversary *can not* achieve, or on what the honest entities *can* achieve.

- **Attack-based security.** Security is defined as the absence of (successful) attacks. In this approach one must consider all conceivable attacks that an adversary could launch, and show that none of these attacks is successful.
- **Constructive approach.** This approach focuses on honest entities. Security means that the honest parties are guaranteed to have a specified system/functionality available (e.g. a secure channel).

Security definitions can also be classified according to whether or not they involve statements about probabilities. Attack-based definitions are often non-probabilistic, while constructive definitions involve probabilities.

1.4.2 Modularity and composability

A central paradigm in any constructive discipline is the decomposition of a complex system into simpler modules. This paradigm is only useful if the composition of modules is well-defined and preserves the relevant properties. In information security and cryptography, the modules are cryptographic schemes or protocols, and the composition must preserver the security of the modules.

Surprisingly, for attack-based security definition, this composition property is often unclear, or does not hold, or at best is highly non-trivial.

1.4.3 The constructive approach

Statement 1.2 (Resource)**:** A *resource* is a system, with an interface to every involved entity, that behaves in a specified manner. Resources model/specify both what the entities have available and what they want to achieve.

A resource can for example be thought of as modeling a trusted party that interacts with the entities in the specified manner.

Statement 1.3 (Constructive approach to security)**:** The goal of a security mechanism or protocol π is to *construct* (or *securely realize*) a resource S (the desired secure system) from a given resource R. A *protocol* π specifies for each (honest) entity what it is supposed to do. This can be written as

$$R \stackrel{\pi}{\Longmapsto} S$$

Statement 1.4 ([Composability of constructive security)**:** The constructive approach to security is composable, i.e. if system T is securely realized from system S by protocol π and system U is securely realized from system T by protocol π', then U is securely realized from S by the composition of protocols π and π'.

$$S \stackrel{\pi}{\Longmapsto} T \quad \wedge \quad T \stackrel{\pi'}{\Longmapsto} U \qquad \Longrightarrow \qquad S \stackrel{\pi \circ \pi'}{\Longmapsto} U$$

1.5 High-level classification of security problems

We can distinguish three cases in term of which entities in the system S are assumed to be (potentially) dishonest.

1. **Unilateral security.** S is a system with a *single* interface (to the environment). One typically wants to protect the system against a possibly hostile environment with which it is interacting.

   Example: A network must be protected against the hostile Internet.

2. **Fixed-adversary security.** S is a system with several interfaces. The adversary controls one (or more) fixed interface. The other entities are known or assumed to be honest (and execute a prescribed protocol).

   This setting can be reduced to the problem of **secure communication** between any two honest entities because pairwise communication between the honest entities allows them to implement any kind of system behavior. What remains is to realize secure communication channels.

   Example: SSL protocol.

3. **Multilateral security.** S is a system with several interfaces, where all entities are potentially dishonest. This assumption is justified when the entities have conflicting interests. Nevertheless they want to realize a system S, no matter who is cheating. The special case of only two entities can be called **bilateral security**.

   Examples: Certified e-mail, or voting.

1.5.1 Unilateral security

In order to define unilateral security, one must define two things:

1. **System specification.** How is the system supposed to behave?
2. **Adversary specification.** What types of adversaries must the system be protected against, i.e., what level of maliciousness must be tolerated.

In comparison, to define correctness of a system we need to define two things:

1. **System specification.** How is the system supposed to behave?
2. **Environment specification.** In which types of environments must the system meet the specification, i.e. what kinds of environments must be tolerated?

Therefore, it naturally follows:

Statement 1.5: System correctness and unilateral security are conceptually equivalent.

1.5.2 Multilateral security

Statement 1.6: Multilateral security can be specified by a resource (or trusted party) S. Achieving multilateral security means securely emulating S.

Examples of multilateral security include the following:

- Millionaires' problem. Two millionaires want to find out who is richer without having to tell each other how rich they are.
- Software piracy problem.
- Database security. How can a user protect itself against a malicious database, without loosing functionality.
- On-line actions.
- E-voting.

2 Cryptography - basic concepts

2.1 Defining security of cryptographic systems and functions

In order to define security, we need to specify at least the following parameters for the adversary:

- **Computing power.**
  - **Infinite computing power.** A system secure under this assumption is called *information-theoretically secure*.
  - **Bounded computing power.** One specifies an upper bound on the computing power, in some model of computation and considers the time required to break the system, using the fastest algorithm. A system secure under a reasonable assumption on the adversary's computing power is called *computationally secure*.

  Obviously, information-theoretic security is more desirable, but it is usually more difficulty to achieve and the practicality of such systems is usually quite limited.
- **Memory capacity.** Usually assumed to be unbounded.
- **Side information.** An adversary may be able to perform certain attacks, for instance choose a plaintext and obtain the corresponding ciphertext (for the active secret key).
- **Corruption capability.** In a multilateral security context one can specify which entities the adversary can corrupt, i.e., can make misbehave in a manner controlled by the adversary.

Regarding side information, there is the following important principle:

Statement 2.1 (Kerckhoffs' principle)**:** A cryptographic system should be designed so as to be secure when the adversary cryptanalyst knows all details of the system, except for the values explicitly declared to be secret (e.g. secret keys).

2.2 Keyless cryptographic functions

2.2.1 One-way functions

Definition 2.1 (One-way functions)**:** A *one-way function* is an efficiently computable function f from a domain A to a co-domain B, $f : A \to B$, such that for every efficient (possibly probabilistic) algorithm $\mathcal{G}$ taking an input from B and producing an output in A, and for $x \in A$ selected uniformly at random

$$P(f(\mathcal{G}(f(x))) = f(x))$$

is negligible.

Remark 2.1: If there are many preimages of $f(x)$, the algorithm $\mathcal{G}$ is only required to find one of them. This is captured in definition 2.1 by the extra $f(\cdot)$ on both sides of the equality sign.

Remark 2.2: Typical one-way functions are not bijective.

2.2.2 Collision-resistant hash functions

Definition 2.2 (Hash function)**:** A *hash function* is an efficiently computable function $h : D \to R$ where $|D| \gg |R|$, typically $D = \{0,1\}^*$ and $R = \{0,1\}^k$ for some suitable k. A hash function can have a parameter c from some set $\mathcal{C}$, selecting a function h_c from a class $\{h_c \mid c \in \mathcal{C}\}$ of functions.

Definition 2.3 (Collision resistance)**:** A hash function class $\{h_c \mid c \in \mathcal{C}\}$ with domain D is *collision-resistant* if for every efficient algorithm $\mathcal{G}$ taking an input $c \in \mathcal{C}$ and producing a pair (x, x') of values in D,

$$P(h_c(x) = h_c(x'))$$

is negligible for $c \in \mathcal{C}$ selected uniformly at random.

2.3 Randomness and pseudo-randomness

2.3.1 Distinguishers

The distinguisher concept is generally defined for systems, but here we only consider the special case of the distinction of two *random variables* S_0 and S_1 (from some set $\mathcal{S}$) by a distinguisher D that is given access to one of them and outputs a (decision) bit. Let $D(S_i)$ be the random variable corresponding to the decision bit when D get S_i.

Definition 2.4: The *advantage* of a distinguisher D in distinguishing S_0 and S_1, denoted $\Delta^D(S_0, S_1)$, is defined as

$$\Delta^D(S_0, S_1) := \Big|P(D(S_0) = 1) - P(D(S_1) = 1)\Big|$$

The *advantage of a class* $\mathcal{D}$ of distinguishers in distinguishing S_0 and S_1, denoted $\Delta^{\mathcal{D}}(S_0, S_1)$, is

$$\Delta^{\mathcal{D}}(S_0, S_1) := \max_{D \in \mathcal{D}} \Delta^D(S_0, S_1)$$

Lemma 2.1: If B is an unbiased random bit, then

$$\Delta^D(S_0, S_1) = 2 \cdot \Big|P(D(S_B) = B) - \frac{1}{2}\Big|$$

Lemma 2.2 (Triangle equality for the distinguishing advantage)**:** For three random variables R, S and T, and any distinguisher D,

$$\Delta^D(R, T) \leq \Delta^D(R, S) + \Delta^D(S, T)$$

2.3.2 Pseudo-randomness

For some reasonable notion of efficiency, let $\mathcal{E}$ denote the class of efficient distinguishers. Let U_k denote a uniform random bitstring of length k.

Definition 2.5 (Pseudo-randomness)**:** An m-bit random variable X is called *pseudo-random*, if it is *computationally indistinguishable* from a uniform random m-bit string, i.e. if $\Delta^{\mathcal{E}}(U_m, X)$ is negligible.

Definition 2.6 (Pseudo-random generator)**:** A function $g : \{0,1\}^k \to \{0,1\}^m$ for $m > k$ is called a *pseudo-random generator* (PRG) if $\Delta^{\mathcal{E}}(U_M, g(U_k))$ is negligible.

Remark 2.3: Pseudo-randomness appears to be a much stronger requirement than one-wayness. A function is one-way, if the entire input cannot efficiently computed, but it is acceptable that one can compute a large part of the input of a one-way function. In contrast, the definition of a PRG requires that one cannot see the slightest difference between two settings. One can easily see, that a PRG is actually a one-way function. The converse is trivially false.

Definition 2.7 (Pseudo-random bit generator)**:** A *pseudo-random bit generator* (PRBG) is an efficient algorithm which takes as input a seed (or key) value of fixed length, and output an (a priori) unbounded sequence of bits, such that the output is computationally indistinguishable from a truly random bit sequence.

2.4 Secrecy, authenticity, and their abstraction

2.4.1 Describing channels

We can describe a communication channel from A to B (with certain security properties) in two different, but equivalent ways.

- By properties such as secrecy, authenticity, availability, single-use vs. multi-use, etc.
- As a system with three interfaces for entities A, B, and E, where E is the adversary. This description is the basis for a constructive approach to security. In such a description, properties like secrecy, authenticity and availability are implicit.

2.4.2 Secrecy and authenticity

A *communication channel* is a system which transports a message (from a certain message domain) from a sender A to a receiver B. The adversary E is involved as a third party and, depending on the type of channel, can interact with it in certain ways.

Because a channel has two endpoints, an input and an output, there are two fundamental security properties a channel can provide:

- **Authenticity.** The *input* of the channel is accessible exclusively to the sender A. No other entity can send a message on the channel. Authenticity is a property considered from the receiver's viewpoint.
- **Secrecy** or **confidentiality.** The *output* of a channel is accessible exclusively to the receiver B. No other entity can read a message sent over the channel. Secrecy is a property considered from the sender's viewpoint.

2.4.3 Availability

In the model of an insecure communication channel, the adversary can always block the communication. The availability of the channel is not (and cannot be) guaranteed. This availability aspect is orthogonal to the secrecy and authenticity aspects, and we therefore ignore availability as a security requirement.

2.4.4 A symbolism for channels and keys

We introduce the following symbols for insecure channels

$A \longrightarrow B$ insecure channel from A to B
$A \longleftarrow B$ insecure channel from B to A

Security properties are modeled by the symbol $\bullet$ which stands for *exclusiveness* of a channel input or output. Therefore, three other types of channels from A to B exists:

$A \longrightarrow\!\bullet\ B$ channel with secrecy from A to B
$A \bullet\!\longrightarrow B$ channel with authenticity from A to B
$A \bullet\!\longrightarrow\!\bullet\ B$ secure channel from A to B (secrecy and authenticity)

For shared keys, we introduce the following notion:

$A \bullet\!=\!=\!\bullet\ B$ Secret key shared by A and B. More precisely, both A and B know that no other entity knows the secret key
$A =\!=\!\bullet\ B$ One-sided key: A knows that at most B knows the key, but B does not know who else holds the key.

The symbol $A \bullet\!=\!=$ means that B knows or believes that the key is secret for A and B, but from A's viewpoint it is possible that other entities know the key or that B does not even know the key.

2.4.5 Timing aspects

A parameter above a channel symbol indicates the time when this channel is available. For instance, $\xrightarrow{t}$ refers to a channel that is available at time t. We can also model channels with delay where the message is sent at t_1, but only received at t_2 (with $t_2 > t_1$). Such a channel is denoted as follows: $\xrightarrow{t_1\ t_2}$.

Similarly, we can add timing information to keys, e.g. $=\!=\!=^{t}$. Unlike a communication channel available at some time t, a shared secret key is persistent in time, i.e., $A \bullet\!=\!=^{t}\!\bullet\ B$ implies $A \bullet\!=\!=^{t'}\!\bullet\ B$ for any $t' > t$ (unless the key is leaked of course).

2.4.6 Trivial security transformation

The following transformation require no cryptography, only common sense. However, they illustrate how this abstract formalism can be used to transform channels.

Security properties obviously always can be dropped:

$$A \bullet\!\xrightarrow{t}\!\bullet\ B \quad \Longmapsto \quad A \bullet\!\xrightarrow{t}\ B$$

Messages can be forwarded:

$$\left.\begin{array}{l} A \xrightarrow{t_1\ t_2} B \\ B \xrightarrow{t_3\ t_4} C \\ \quad t_3 \geq t_2 \end{array}\right\} \quad \Longmapsto \quad A \xrightarrow{t_1\ t_4} C$$

2.4.7 Key-transport security transformations

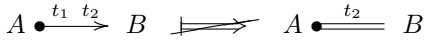
Using a secure channel, one can transport a secret key:

$$A \bullet\!\xrightarrow{t_1\ t_2}\!\bullet\ B \quad \Longmapsto \quad A \bullet\!=\!=^{t_2}\!\bullet\ B$$

This transformation can also be achieved when the channel does not provide authenticity:

$$A \xrightarrow{t_1\ t_2}\!\bullet\ B \quad \Longmapsto \quad A =\!=^{t_2}\!\bullet\ B$$

However, the same transformation without *secrecy*, i.e. only authenticity, does *not* work:

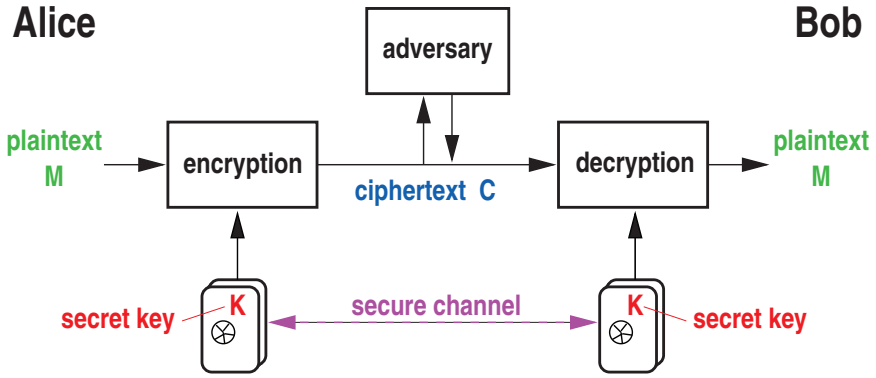


2.5 Symmetric cryptosystems

2.5.1 Concept and definition

The model of a (deterministic) *symmetric cryptosystem*, also called a *cipher*, is shown in figure 1.

Figure 1: Symmetric cryptosystem (cipher)



The message M, often called *plaintext*, is encrypted by a sender Alice, using a *secret key* K, resulting in the *ciphertext* C. The encryption transformation can be probabilistic, i.e., involve fresh random bits for each message to be encrypted. More formally:

Definition 2.8 (Cipher, symmetric cryptosystem)**:** A *cipher* for message space $\mathcal{M}$, ciphertext space $\mathcal{C}$ and key space $\mathcal{K}$ is a pair of functions; an encryption function $E : \mathcal{M} \times \mathcal{S} \to \mathcal{C}$ and a decryption function $D : \mathcal{C} \times \mathcal{S} \to \mathcal{M}$ such that $D(E(m,k),k) = m$ for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$. A *symmetric cryptosystem* is a possibly probabilistic cipher with randomness space $\mathcal{R}$, i.e., a pair of functions $E : \mathcal{M} \times \mathcal{S} \times \mathcal{R} \to \mathcal{C}$ and $D : \mathcal{C} \times \mathcal{S} \to \mathcal{M}$ such that $D(E(m,k,r),k) = m$ for all $k \in \mathcal{K}$, $m \in \mathcal{M}$ and $r \in \mathcal{R}$.

2.5.2 Types of symmetric cryptosystems

Without loss of generality, every practical cryptosystem can be modeled as a finite automaton that processes the message in units of a certain size and generates, per unit of plaintext, one unit of ciphertext. The ith ciphertext unit is a function of the state at time $i-1$ and the ith message unit. Three special instantiations of this general principle are used in general:

- **Block cipher.** A block cipher is a stateless encryption function where plaintext and ciphertext are n-bit strings. When used in the natural mode (called electronic codebook mode), a message is encrypted by cutting it into n-bit blocks and encrypting each block separately.
- **Additive stream cipher.** An additive stream cipher is obtained from the one-time pad by replacing the random key by the output of a pseudo-random generator, which is added modulo 2 (XOR) to the plaintext. In contrast to a block cipher, a stream cipher is stateful (the state of the PRG). But the state evolves autonomously, independent of the plaintext.
- **Self-synchronizing stream cipher (SSSC).** Encryption is bit-by-bit. The state consists of the m most recent ciphertext bits. The ith bit is encrypted by adding modulo 2 a bit computed by some function from the state and the secret key. Decryption is self-synchronizing, i.e., one can start decryption at any intermediate point of a ciphertext, without need for context information. After m ciphertext bits have been processed, decryption works properly. The so-called *cipher feedback mode* is a possible implementation of an SSSC.

2.5.3 Does encryption provide authenticity?

Naïvely viewed, encryption provides both secrecy and authenticity. The latter is provided because an attacker cannot insert or replace messages. This seems true because only the sender and receiver know the secret key and can hence compute a

valid ciphertext.

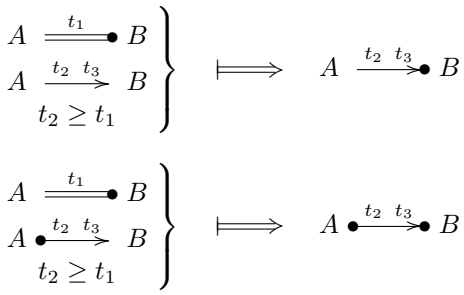
However, this reasoning is false in general for two reasons:

- If the cipher is non-expanding and the plaintext contains no redundancy, then every ciphertext is valid and an adversary could insert any ciphertext. He might not know what the message is, but he can fool the receiver into believing that the message came from the legitimate sender.
- Even a very secure cipher might not prevent modifications of the ciphertext, resulting in a certain predictable modification of the plaintext. An example is the one-time pad, where flipping the ith bit in the ciphertext results in a flipped bit at position i in the plaintext.

In typical applications one does not assume that encryption provides authenticity and instead uses an explicit mechanism for authentication, called a message authentication code, MAC for short.

2.5.4 Interpretation as security transformations

The basic security transformation provided by symmetric ciphers is the following



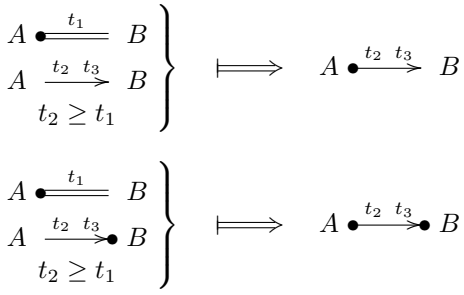
2.6 Message authentication codes (MAC)

2.6.1 Concept and definition

Definition 2.9 (Message authentication codes)**:** A *message authentication code (MAC)* for message space $\mathcal{M}$, key space $\mathcal{K}$, and tag space $\mathcal{T}$ is a function $f : \mathcal{M} \times \mathcal{K} \to \mathcal{T}$ such that the following security condition holds:

MAC security definition: Let k be chosen uniformly from $\mathcal{K}$ and consider an oracle $\mathcal{M} \to \mathcal{T}$ computing the tag $f(m, k)$ for an input message m. There exists no efficient alogrithm with access to the oracle, which outputs with non-negligible probability a message m' different from all messages asked to the oracle, as well as the corresponding tag $t = f(m', k)$.

As a security transformation, a MAC scheme achieves the following two transformations:

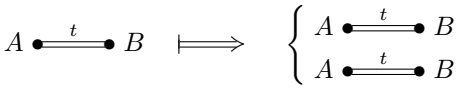


2.6.2 Limitations of MACs

In many applications a stronger form of authentication, namely the ability to *prove* to a another party (e.g. a judge) that a certain entity sent a message. This is called *non-repudiation*, and is not achieved by MACs. This is obvious, as not only the sender, but also the receiver (who knows the secret key) could have generated the message tag.

2.7 Combining encryption and message authentication

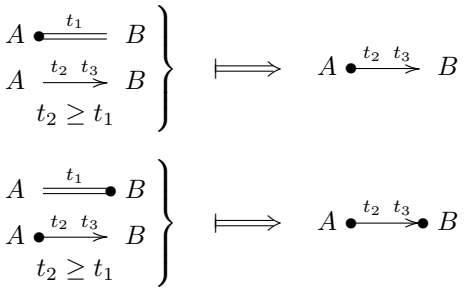
Both the symmetric cryptosystem and the MAC scheme require a secret key, but the key on the left side of a transformation is "consumed" when the transformation is applied. However, a pseudo-random generator (PRG) can duplicate the shared secret key by expanding the given key into a key of sufficient length:



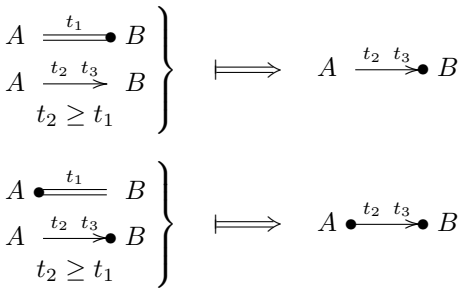
This step can be repeated as often as needed.

Using only a two-sided shared secret key and insecure channels, there are two possibilities to achieve a secure channel:

- **Encrypt-then-MAC.** Encrypt the message and then apply a MAC to the ciphertext.



- **MAC-then-encrypt.** MAC the message and then encrypt the message-MAC pair.



Note that the order in which the security transformations are applied is reversed compared to the order in which the operations are performed.

2.8 Public-key cryptography

2.8.1 Concept and definition

The concept of public-key cryptosystems is shown in figure 2. A user generates a key pair consisting of a secret key s_b and the corresponding public key p_b, which he makes public. There is an encryption operations, making use of the public key by which any entity can encrypt a message. However, only the owner of the private key can decrypt the ciphertext.

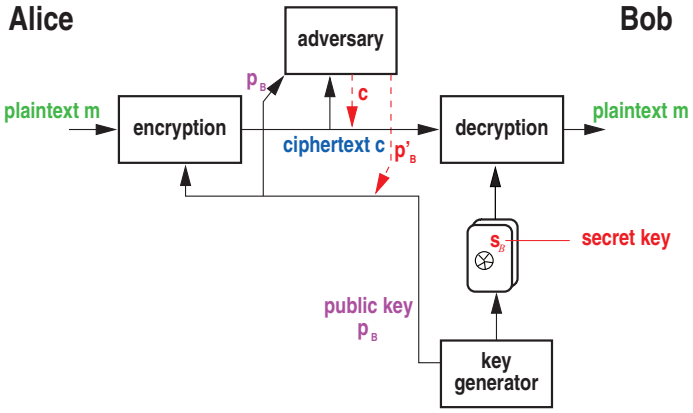
Definition 2.10 (Public-key cryptosystem)**:** A *public-key cryptosystem (PKC)* consists of three efficient algorithms:

- The *key generator* is a probabilistic algorithm which generates a key pair (consisting of an encryption key and a decryption key) according to a certain probability distribution.
- The *encryption algorithm* takes as inputs an encryption key an d a plaintext and computes the ciphertext. Encryption can be probabilistic.
- The *decryption algorithm* takes as inputs a decryption key and a ciphertext and computes the plaintext.

The following conditions must be satisfied:

- *Decipherability.* For every encryption/decryption key pair the decryption transformation must be the inverse of the encryption transformation.
- *Security.* It must be computationally infeasible to compute "useful" information about the plaintext for a given ciphertext without knowledge of the secret key.

Figure 2: Public-key cryptosystem



2.8.2 Interpretation as a security transformation

The basic security transformation achieved by a PKC is the following:

$$\left.\begin{array}{l} A \overset{t_1\ t_2}{\bullet\!\!\longrightarrow} B \\ A \overset{t_3\ t_4}{\longleftarrow\!\!\bullet} B \\ t_3 > t_2 \end{array}\right\} \Longmapsto \quad A \overset{t_4\ t_3}{\bullet\!\!\longleftarrow} B$$

The public key is sent over the authenticated channel and the message is encrypted by B (using the public key) and sent over an insecure channel.

If the channel from B to A also provides authenticity, then the resulting channel is secure:

$$\left.\begin{array}{l} A \overset{t_1\ t_2}{\bullet\!\!\longrightarrow} B \\ A \overset{t_3\ t_4}{\longleftarrow\!\!\bullet} B \\ t_3 > t_2 \end{array}\right\} \Longmapsto \quad A \overset{t_4\ t_3}{\bullet\!\!\longleftarrow\!\!\bullet} B$$

2.8.3 Secret-key establishment using public-key encryption

The typical use of a PKC is not for encrypting actual application messages, but for key management where the message is a session key. The corresponding transformations are the following:

$$\left.\begin{array}{l} A \overset{t_1\ t_2}{\bullet\!\!\longrightarrow} B \\ A \overset{t_3\ t_4}{\longleftarrow\!\!\bullet} B \\ t_3 > t_2 \end{array}\right\} \Longmapsto \quad A \overset{t_4}{\bullet\!\!=\!\!=} B$$

$$\left.\begin{array}{l} A \overset{t_1\ t_2}{\bullet\!\!\longrightarrow} B \\ A \overset{t_3\ t_4}{\longleftarrow\!\!\bullet} B \\ t_3 > t_2 \end{array}\right\} \Longmapsto \quad A \overset{t_4}{\bullet\!\!=\!\!=\!\!\bullet} B$$

2.8.4 The Diffie-Hellman key-agreement protocol

The Diffie-Hellman key-agreement (or key-distribution) protocol is based on the *discrete logarithm problem*. This problem is conjectured to be generally computationally infeasible, i.e. exponentiation modulo a large prime is a (conjectured) one-way function. The protocol uses only an authenticated channel to generate a shared, secret key.

This works as follows: A large prime p and a basis g are chosen as public parameters. Then, both Alice and Bob choose a number at random from $0, \ldots, p-2$, say x_A and x_B respectively. They exponentiate the base by there secret parameter modulo p and send the result, y_A and y_B, to the other person. Now, both parties exponentiate what they got by their secret parameter (again, modulo p), and thereby get the secret key:

$$k_{AB} \equiv_p y_B^{x_A} \equiv_p (g^{x_B})^{x_A} \equiv g^{x_A x_B} \equiv k_{BA}$$

2.8.5 The RSA public/key cryptosystem

Theorem 2.1: Let G be some finite group (with neutral element 1), and let $e \in \mathbb{Z}$ be a given exponent relatively prime to $|G|$ (i.e. $\gcd(e, |G|) = 1$). The (unique) e-th root of $y \in G$, namely $x \in G$ satisfying $x^e = y$, can be computed according to

$$x = y^d$$

where d is the multiplicative inverse of e modulo $|G|$, i.e.

$$d \equiv_{|G|} e^{-1}$$

When $|G|$ is known, then d can be computed efficiently using the extended Euclidean algorithm. No general method is known for computing the e-th root in a group of unknown order. This fact can be exploited to define a public-key cryptosystem.
The protocol works as follows:

Alice	**insecure channel**	**Bob**
Generate primes p and q		
$n = p \cdot q$		
$\varphi(n) = (p-1)(q-1)$		
select e		
$d := e^{-1} (\mathrm{mod}\, \varphi(n))$	$\xrightarrow{n,e}$	plaintext
		$m \in \{1, \ldots, n-1\}$
		ciphertext
$m := c^d (\mathrm{mod}\, n)$	$\xleftarrow{c}$	$c := m^e (\mathrm{mod}\, n)$

2.9 Digital signatures

2.9.1 Concept and definition

Definition 2.11 (Digital signature scheme)**:** A *digital signature scheme (DSS)* consists of three efficient algorithms:

- The *key generator* is a probabilistic algorithm which generates a key pair, consisting of a *signing key* (or secret key or private key) and a signature *verification key* *or public key), according to a certain probability distribution.
- The *signing algorithm* takes as input a signing key and a message and computes the signature for the message. Signing can be probabilistic.
- The *signature verification algorithm* takes as inputs a signature verification key, a message, and a signature, and outputs a bit (which can be interpreted as "accept" or "reject").

The following conditions must be satisfied:

- *Correctness.* For every signing/verification key pair the verification algorithm outputs "accept" for a signature computed by the signing algorithm.
- *Security.* Without access to the signing key it must be computationally infeasible to compute a signature for a message.

2.9.2 Interpretation as security transformation

A DSS can be used to preserve authenticity in time in the same way as a symmetric encryption scheme can preserve secrecy in time.
As before, there are two variants of the actual transformation:

$$\left.\begin{array}{l} A \bullet\!\xrightarrow{t_1\ t_2} B \\ A \xrightarrow{t_3\ t_4} B \\ \quad t_4 \geq t_2 \end{array}\right\} \quad \Longmapsto \quad A \bullet\!\xrightarrow{t_3\ t_4} B$$

$$\left.\begin{array}{l} A \bullet\!\xrightarrow{t_1\ t_2} B \\ A \xrightarrow{t_3\ t_4}\!\bullet\ B \\ \quad t_4 \geq t_2 \end{array}\right\} \quad \Longmapsto \quad A \bullet\!\xrightarrow{t_3\ t_4}\!\bullet\ B$$

3 Security in distributed systems and the role of trust

3.1 The security bootstrapping problem

3.1.1 Security bootstrapping

Security symbols (•) cannot be created by cryptography, they must be generated by physical means. One can distinguish two phases (which can overlap) of security bootstrapping:

- **Set-up phase.** The channels or keys with security symbols are generated by physical means.
- **Communication phase.** Only insecure channels such as the Internet are available. They are transformed into channels with the desired security properties.

3.1.2 Creating •'s

There are several possibilities to generate •'s by physical means:

- People can meat physically and exchange security parameters.
- A person can recognize another person's voice in a telephone conversation and hence authenticate a communication channel.
- A trusted courier can transport a secret key.

3.1.3 Naming entities, pseudonyms and entity invariance

A security symbol always belongs to an entity as specified (from another entities viewpoint) by some parameter. For instance, an entity could be a person. However, an entity can also be specified by other parameters like a pseudonym, which does not uniquely identify the person in a physical sense, but which for the application at hand is nevertheless sufficient. The use of "•" does not imply that the person is physically identified, only that in the given context it is the "right" person.

Taking this one step further, the entity of interest could also be a computer, known only by some parameter, but for which it is not even known where on the Internet it is located. The parameter could for instance be the session key agreed upon when setting up the session. In other words, it makes perfect sense for an entity B to assign a security symbol to a channel from C to B, even if B knows nothing about C other than its existence.

In such a context the concept of authenticity is sometimes also called *entity invariance* or *sender invariance*.

3.2 Limitations of cryptographic security transformations

3.2.1 Trust-free security transformations

Definition 3.1 (Cryptographic and trust-free security transformations)**:** *Cryptographic security transformations* are those realize by cryptographic systems. We use the term *trust-free security transformations* to refer to the trivial transformations, the key-transport transformations, and the cryptographic security transformations.

It is easy to see that cryptographic security transformations can move security symbols "•" from one channel to another, but they remain attached to the corresponding entities. This can be summarized in the following observation:

Observation 3.1: There exists no trust-free security transformation which can either

1. generate a "•",
2. move a "•" from one entity to another, or
3. move a "•" from one channel to another channel for which not both end-points are the same

even if insecure channels are freely available.

3.2.2 Security bootstrapping without trust assumptions

Motivated by the availability of the Internet as an insecure communication medium, we make the following simplifying assumption:

Assumption 3.1 (Assumption of permanent connections)**:** Insecure channels $\longrightarrow$ are available at any time between any two entities.

Proposition 3.1: Under the assumption of permanent connections, a secure channel between A and B after time t_0, $A \bullet\!\!\xleftarrow{t}\!\!\bullet\, B$ for $t \geq t_0$, can be obtained under the following two conditions:

1. There exists, at some time $t_1 < t_0$, a channel or key with a $\bullet$attached to A: $A \bullet\!\!\xleftarrow{t_1}$ B, $A \bullet\!\!\xrightarrow{t_1}$ B, or $A \bullet\!\!\xlongequal{t_1}$ B.
2. There exists, at some time $t_2 < t_0$, a channel or key with a $\bullet$attached to B: $A \xleftarrow{t_1}\!\!\bullet\, B$, $A \xrightarrow{t_1}\!\!\bullet\, B$, or $A \xlongequal{t_1}\!\!\bullet\, B$.

Under the additional assumption that only trust-free security transformations are available, these two conditions are also necessary.

3.2.3 The quadratic blow-up problem

In a large distributed system like the Internet it is typically required that any arbitrary two entities A and B can communicate securely, $A \bullet\!\!\longleftrightarrow\!\!\bullet\, B$, at any time.

Statement 3.1 (The quadratic blow-up problem)**:** If only trust-free security transformations are available, then, to be prepared for secure communication between any two entities, the number of security symbols "$\bullet$" that must be established in the set-up phase is *quadratic in the number of entities*. In particular, the number of "$\bullet$" to be created by each user is linear in the number of entities.

Obviously, this problem must be solved for security on the Internet to scale. This is the main purpose of using trust, i.e., entities that are trusted by some other entities to perform certain tasks correctly.

3.3 Trust-based security transformations

3.3.1 Roles of trusted entities

Trust requirements should be minimized. Two important issues to be to be addressed in the design of protocols are:

1. **Who needs to trust which entity?** For instance, is an entity trusted only by some entities or by all entities participating in a system?
2. **Involvment of the trusted entity:** One can distinguish three types:
   - **In-line:** The communication between A and B is relayed through T.
   - **On-line:** The session between A and B is set up by T, but the actual communication is directly between A and B.
   - **Off-line:** T is not involved in the setting up a session between A and B. In fact, T need not know that A and B may intend to establish a session.

3.3.2 Connection channels

A trusted entity can simply forward messages, which results in the following security transformation:

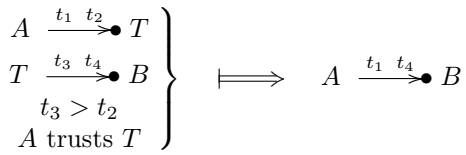
$$\left.\begin{array}{c} A \bullet\!\!\xrightarrow{t_1\ t_2} T \\ T \bullet\!\!\xrightarrow{t_3\ t_4} B \\ t_3 > t_2 \\ B \text{ trusts } T \end{array}\right\} \quad \Longmapsto \quad A \bullet\!\!\xrightarrow{t_1\ t_4} B$$

A few details about this transformation:

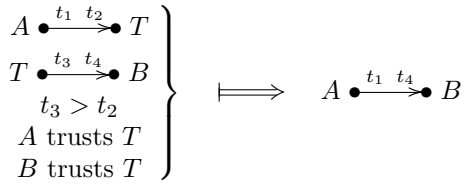
- Obviously, the time t_3 when the second channel is invoked must be after the message is received on the first channel. This limitation will be solved by public-key certification.
- The message on the second channel is related to the message on the first channel (same content). That means, T needs to be involved in-line. This also is solved by public-key certification.
- B must trust T to make correct statements about which entity the message originates from. This includes that T must properly authenticate the source. B relies on T's judgement.

- Note that A need not to trust T. Actually, if B's trust in T were not justified, A could not prevent B from accepting a fasle message from T.

Similarly, the following transformation, that does not appear to be of practical interest, is possible:

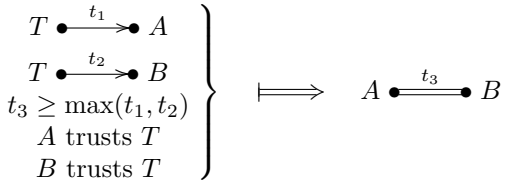


The above two transformations can be combined, where now both A and B must trust T:



3.3.3 Key distribution

We consider one more trust-based transformation by which A and B can delegate the generation of a secret key for them to T, provided there exists secure channels from T to A and from T to B.



Note that both A and B must trust T to generate a fresh random key K_{AB}, and not to leak this key to any other entity.

3.4 Necessary and sufficient conditions for establishing secure channels or secret keys

In the sequel, let the symbol •—— denote any of the three primitives •⟶ , •⟵ , •═══ , and let the symbol •——○ denote any of the six primitives •⟶ , •⟵ , •═══ , •⟶• , •⟵• , •═══• . In other words, "○" can either be replaced by a •or dropped.

Observation 3.2: There exists no trust-based security transformation which can either

1. generate a •,
2. move a •from one entity to another, or
3. move a •from a primitive A •——○ T to a primitive A ——○ B unless two conditions are satisfied: (1) there exists a channel T •——○ B and (2) B trusts T.

Proposition 3.2: Under the assumption of permanent connections, any primitive A •—^{t}— B for $t \geq t_0$ can be derived from a given scenario if there exists a path of primitives •——○ from A to B such that:

1. Every primitive is available at some time before t_0 and has a security symbol "•" on the endpoint on A's side.
2. B trusts *every* entity on the path.

Under the additional assumption that only trust-free and trust-based security transformation are available, these two conditions are also necessary for deriving any of the primitives A •—^{t}— B.

The following proposition is a direct consequence of proposition 3.2:

Proposition 3.3: Under the assumption of permanent connections, a secure channel between A and B after time t_0 (A •⟷^{t}• B for $t \geq t_0$) or a shared secret key A •═══^{t_0}• can be obtained only if and only if there exist two paths according to proposition 3.2, one from A to be, and, symmetrically, one from B to A.

It is important to note that the two paths can be disjoint. In other words, it is not necessary that there exists an entity trusted by *both* A and B.

4 Key management, certificates, and PKIs

4.1 Key management

Cryptographic keys are a way to preserve security properties "•" over time and can be used to achieve many other security goals. Establishing and managing cryptographic keys is called key management.

For the two types of cryptographic keys - (symmetric) secret keys and (asymmetric) public keys, there are two basic key management problems:

- **(Symmetric) key agreement.** Two (or more) entities A and B wish to establish a common *shared secret key*, which is typically used for encrypting and authenticating the communication within a session.
- **Public-key authentication.** An entity B wants to obtain the *authentic public key* of another entity A. There are two types of public keys, for public-key encryption and for digital signature verification.

The key agreement problem is often solved by reducing it to the public-key authentication problem. The public-key authentication problem is therefore of more fundamental interest than direct solutions to the key agreement problem.

4.2 Protocols based on shared secret keys with trusted authority

4.2.1 Star-shaped security set-up

We consider a central authority T trusted by all entities. This makes sense in a centralized organisation such as a company, but not necessarily in a heterogeneous environment. We assume that in the set-up phase, every entity, say A, establishes a secure channel $T \overset{t}{\longrightarrow} A$ with T (for some t) and exchanges a shared secret key K_{AT} over this channel, resulting in

$$A \overset{t}{\Longrightarrow} T$$

for some t. This can for instance be achieved by a physical visit. No security symbols "•" between entities must be established, solving the quadratic blow-up problem.

4.2.2 Trusted party relays messages

In a star-shaped topology of shared secret keys, the perhaps most natural way to achieve a secure channel between any two entities, say A and B, is for T to relay messages between A and B.

The main disadvantage of such a simple relay protocol is that T is *in-line* and therefore constitutes a communication bottleneck.

4.2.3 Trusted party distributes session keys

Instead of acting in-line, T can be used *on-line* to establish a shared secret key between A and B. There are several different variants for such a session key establishment.

First of all, we can use the protocol described above in section 4.2.2. A could generate a secret key K_{AB}, then sending it with the protocol described to B. From then on, the two entities can communicate without the help of T.

However, most protocols that make use of a trusted entity T to set up a session key actually let T (rather than A) generate a session key. There are two natural variants for this. In both variants, A initiates the session by informing T that it wants to establish a session with B, and T then generates a session key K_{AB} and encrypts it with both keys K_{AT} and K_{BT}. In one variant, T sends both ciphertext to A, whereas in the other case T sends the ciphertext for B directly to B. Both variants are shown in figure 3

4.3 Public key certificates

The key management protocols of section 4.2 all suffer from the following three basic drawbacks:

- The trusted entity T must be *on-line* when A and B establish a session. In particular, T must be aware of the fact that A contacts B.
- Both entities A and B must trust T, i.e., there is a need for a *commonly trusted* entity.
- T could read the messages exchanged between A and B, without possibility for A and B to notice this.

Public-key certificates can solve all these problems.

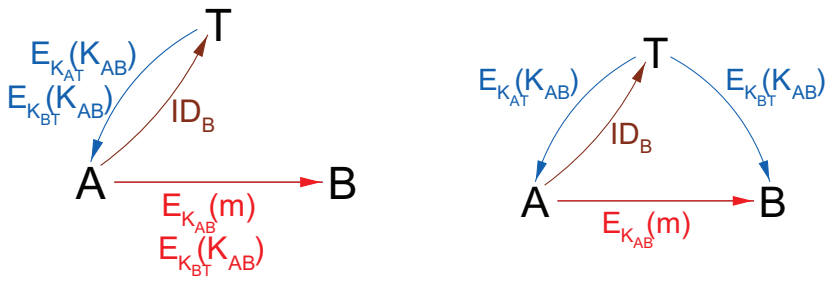


Figure 3: Two variants of a protocol for session key distribution by a trusted party T.

4.3.1 Certificates, the concept

Definition 4.1 (Certificate)**:** Informally, a *certificate* is a digitally signed statement by an entity C, often called a *certification authority*, stating that a certain public key P_A "belongs to" an entity A (with name ID_A). Such a certificate will be denoted by $\mathrm{Cert}_{C,A}$.

More concretely, a certificate consists of two parts: a data part, and the signature on the data part. The data part consists at least of a public key, the name of the entity for which the certificate is issued, the name of the entity who issued the certificate and possibly further parameters like an expiration date, the description of a context in which the certificate can be applied, the policy applied by the issuing entity, etc.

A certificate $\mathrm{Cert}_{C,A}$ issued by C is useful for an entity, say B, if the following two conditions are satisfied:

1. B holds an *authenticated* copy of C's public key, which is needed to verify the signature of the certificate.
2. B *trusts* C to properly authenticate entities for which it issues certificates and to make correct statements (at least in the context of certification).

At first, certificates may appear to be a useless mechanism since one reduces the problem of authenticating a public key (A's) to the *same* problem for C's public key, with an additional trust requirement. However, in practice it may be easier to authenticate the public key of C.

4.3.2 The semantics of certificates

The purpose of a certificate is to "bind" a public key to a particular entity. Possible meanings of a certificate $\mathrm{Cert}_{C,A}$ include the following:

- A claimed (towards T) that P_A is her public key.
- Like the above, but T also confirms it verified that A knows the corresponding secret key.
- Non-repudiation: A commits herself to be liable for statements (e.g. a contract) of certain types signed relative to her signature public key.

4.3.3 Interpretation of certificates in the •-calculus

We recall the following trust-based security transformation from section 3.3.2:

$$\left.\begin{array}{l} A \xrightarrow{t_1\ t_2}\bullet\ T \\ T \xrightarrow{t_3\ t_4}\bullet\ B \\ t_3 > t_2 \\ A \text{ trusts } T \end{array}\right\} \quad \Longmapsto \quad A \xrightarrow{t_1\ t_4}\bullet\ B$$

which allows a trusted entity T to transfer an authenticated message from a sender A to a recipient B who do not share any security symbols initially. However, there are two major drawbacks:

- The time condition $t_3 > t_2$ implies that the order in which the authenticated channels are established during the set-up phase matters.
- The message on the two channels $A \xrightarrow{t_1\ t_2}\bullet\ T$ and $T \xrightarrow{t_3\ t_4}\bullet\ B$ are related, which implies that T must be on-line.

Both problems are solved by public-key certification. The goal is to obtain an authenticated channel from A to B, i.e. $A \overset{t_1\ t_4}{\bullet\!\!\longrightarrow} B$.

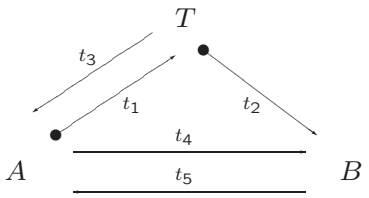


Figure 4: Formal interpretation of public key certification: the involved channels.

Figure 4 shows the involved channels. First, two insecure channels can be connected:

$$\left.\begin{array}{l} T \overset{t_3}{\longrightarrow} A \\ A \overset{t_4}{\longrightarrow} B \end{array}\right\} \Longmapsto \quad T \overset{t_3\ t_4}{\longrightarrow} B$$

The digital signature transformation allows to transfer the authenticity of the channel $T \overset{t_2}{\bullet\!\!\longrightarrow} B$ to the channel $T \overset{t_3\ t_4}{\longrightarrow} B$ available at a later time.

$$\left.\begin{array}{l} T \overset{t_2}{\bullet\!\!\longrightarrow} B \\ T \overset{t_3\ t_4}{\longrightarrow} B \\ t_4 > t_2 \end{array}\right\} \Longmapsto \quad T \overset{t_3\ t_4}{\bullet\!\!\longrightarrow} B$$

The message sent over the (virtual) channel $T \overset{t_2}{\bullet\!\!\longrightarrow} B$ is T's signature public key, and the security symbol "$\bullet$" on the channel $T \overset{t_3\ t_4}{\longrightarrow} B$ is obtained by signing the transmitted message. Here, the message sent over the channel $T \overset{t_3\ t_4}{\bullet\!\!\longrightarrow} B$ is A's public key, or more precisely, the data part of the certificate.

If B trusts T and $t_3 > t_1$, we can now apply the following transformation:

$$\left.\begin{array}{l} A \overset{t_1}{\bullet\!\!\longrightarrow} T \\ T \overset{t_3\ t_4}{\bullet\!\!\longrightarrow} B \\ t_3 > t_1 \\ B \text{ trusts } T \end{array}\right\} \Longmapsto \quad A \overset{t_1\ t_4}{\bullet\!\!\longrightarrow} B$$

Note that in this case there is no dependence of the messages sent over the two channels $A \overset{t_1}{\bullet\!\!\longrightarrow} T$ and $T \overset{t_2}{\bullet\!\!\longrightarrow} B$. This is a major advantage of public-key certification.

4.3.4 Using certificates

Since there are two types of public keys, there are (at least) two types of certificates. For a signature certificate one can apply the transformation

$$\left.\begin{array}{l} A \overset{t_1\ t_4}{\bullet\!\!\longrightarrow} B \\ A \overset{t_4}{\longrightarrow} B \end{array}\right\} \Longmapsto \quad A \overset{t_4}{\bullet\!\!\longrightarrow} B$$

to send an authenticated message (at the actual connection time t_4). Note that the second channel is used to transmit two messages, the certificate and the signed message for B.

For an encryption certificate, one can apply the transformation

$$\left.\begin{array}{l} A \overset{t_1\ t_4}{\bullet\!\!\longrightarrow} B \\ A \overset{t_4}{\longleftarrow} B \end{array}\right\} \Longmapsto \quad A \overset{t_4}{\bullet\!\!\longleftarrow} B$$

to send a confidential message from B to A.

4.3.5 Certificate chains

When given a certificate $\text{Cert}_{C,A}$ for entity A issued by entity C, we need to authenticate C's public key. Of course, we can do so with yet another certificate $\text{Cert}_{C,D}$, and so on. This leads to certificate chains.

Definition 4.2 (certificate chain)**:** A *certificate chain* of length k from an entity E to an enitty A is a list of certificates

$$\left[\text{Cert}_{E,D_1}, \text{Cert}_{D_1,D_2}, \ldots, \text{Cert}_{D_{k-2},D_{k-1}}, \text{Cert}_{D_{k-1},A}\right]$$

for some intermediate entities $D_1, \ldots, D_{k-1}$, where all but possibly the last certificates are signature certificates.

For suc ha certificate chain to be useful for an entity B to authenticate the public key of an entity A, two conditions must be satisfied:

- B must have an authenticated copy of the first entity E's public key.
- B must trust all entities on the chain, i. $E, D_1, \ldots, D_{k-1}$, but not A.

Note that in the set-up phase, one needs an authenticated channel $\bullet\!\!\longrightarrow$ from any entity in the chain to the previous entity.

4.4 A logical calculus for certificates and trust

4.4.1 Introduction and notation

We develop a calculus that can be used by A to reason about the authenticity of public keys and about trust. A's view consists of the following types of evidence:

- $\text{Aut}_{A,X}$: A believes to hold an authenticated copy of X's public key. This believe can either be first-hand knowledge, or it can be derived within the calculus, based on other evidence.
- $\text{Cert}_{X,Y}$: A certificate for Y carrying the issuer identity X.
- $\text{Trust}_{A,X,i}$: A trusts X of level i. This can either be based on first-hand knowledge or it can be derived within the calculus.
- $\text{Rec}_{X,Y,i}$: A recommendation for Y of level i, carrying the issuer identity X.

Trust of level 1 is the usual type of trust. higher levels are requred for issuing recommendations. There is also a graphical notation, where $\text{Aut}_{A,B}$ and $\text{Cert}_{A,B}$ are represented by a solid arrow from A to B, and $\text{Trust}_{A,B,i}$ and $\text{Rec}_{A,B,i}$ are drawn using a dashed arrow, labeled with i.

4.4.2 Inferring the authenticity of a public key

Definition 4.3: The general *inference rule for deriving the authenticity* of a public key is

$$\forall X, Y, i \geq 1: \quad \text{Aut}_{A,X}, \text{Trust}_{A,X,i}, \text{Cert}_{X,Y} \vdash \text{Aut}_{A,Y}$$

4.4.3 Recommendations

A certificate generally makes a statement about the authenticity of a public key, not about the trustworthiness of the certified entity. It is important to note, that:

Statement 4.1: Authenticity and trust are completely orthogonal.

However, one can introduce a special type of "certificate", which we call a recommendation.

Definition 4.4 (recommendation)**:** A *recommendation* $\text{Rec}_{X,Y,i}$ is an explicit statement, signed by an entity X, that entity Y is trustworthy (of level i).

The basic idea is that if one trusts X, then one can accept the recommendation and consider Y as trustworthy. But it is important to note that a higher level of trust is required for accepting a recommendation. Therefore, there are in principle an unbounded number of trust levels.

- Level 1: Trust X that a certificate issued by X is correct.
- Level 2: Trust X that an entity recommended by X can be considered trustworthy to issue certificates (i.e. level 1).
- Level i: Trust X that an entity recommended by X can be considered trustworthy of level $i-1$.

4.4.4 Inferring trust

Definition 4.5: The general inference rule for inferring trust in an entity is

$$\forall X, Y, i, j \geq 1: \quad \mathrm{Aut}_{A,X}, \mathrm{Trust}_{A,X,i+1}, \mathrm{Rec}_{X,Y,j} \vdash \mathrm{Trust}_{A,Y,\min(i,j)}$$

4.5 Certification structures

4.5.1 The certification graph

Public-key certification scenarios can be characterized by the *certification graph*, defining which entities certify which other entities. Two conflicting goals are:

- Entities must be able to efficiently identify suitable certificate chains in the graph such that they trust all entities on the chain.
- It must be feasible to issue the certificates, i.e., it must be feasible to establish a (physically) authenticated channel.

4.5.2 Hierarchical certification

It is natural to apply a hierarchically organized structure, i.e., to use a tree as the certification graph. The top certification authority is called the *root-CA*. Figure 5 gives an example of such an approach.

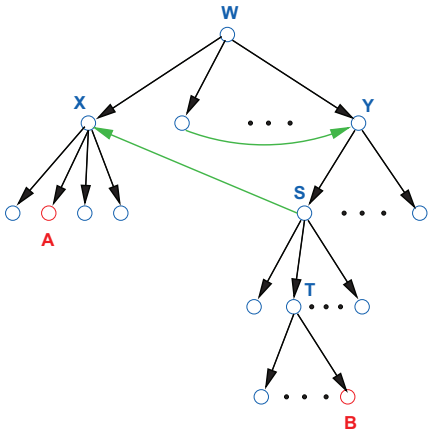


Figure 5: Hierarchical certification with root-CA W and two cross-certificates.

A tree is convenient because it is trivial to find a certificate chain from the root-CA to every entity in the tree. If A wants to authenticate the public key of B, A has to trust *all* entities on the path to B. However, A typically has no relationship to the entities on the path to B and hence may not be able to trust them. For instance, the lowest level entity (T in figure 5) who issued the certificate for B could be an administrator in an organization one does not know or even distrust to some extent.

To trust all entities in the hierarchy because one trusts the root-CA is a very weak argument, except for tightly managed hierarchies.

4.5.3 Cross-certification

A tree is a rigid and inflexible graph in which there is only one fixed path to a given entity. It may make sense to certify entities not only downwards, but also between different branches. This is called *cross-certification*. For instance, two companies who are entering a business relationship could mutually cross-certify their public keys to avoid the need for higher-level certificates.

4.5.4 Unstructured certification

While a hierarchical certification makes sense for establishing a PKI within a closed hierarchical organization, it is not suited for non-hierarchical settings like the user population on the Internet.

As an alternative, one can consider a setting where every entity can in principle serve as a certification authority. This leads to a potentially fine-meshed graph in which there can exist many different certificate chains for a given user. Hence there is

much more flexibility in choosing a chain and therefore choosing whom to trust. A disadvantage of this approach is the lack of central control.

4.5.5 Retrieving certificates

The problem of retrieving the certificates is in general independent of the topology of the certification graph. For example, certificates could be stored in a publicly accessible directory service, or they could be provided by the target entities.

4.6 Public-key infrastructures (PKI)

4.6.1 Concept

Definition 4.6 (public-key infrastructure (PKI))**:** The term *public-key infrastructure (PKI)* refers to the collection of systems and procedures for the

- generation of secret-key/public-key pairs,
- authentication of entities applying for certificates,
- generation of certificates,
- distribution of certificates,
- revocation of certificates, and
- verification of certificates.

4.6.2 Problems

Despite the usefulness, many have failed in implementing a PKI. In fact, there is still not a widely usable national or international PKI in place. Perhaps the most important reasons for this are:

- There is a chicken-egg-problem: In order for a PKI to be useful, there need to be applications making use of them. On the other hand, such applications can only emerge and be used broadly if certificates are available.
- Standardization. One needs clear, enforceable standards, both for applications and for the PKI. However, not all players in the market are interested in having strong standards in place, and therefore a clear standard is still missing. The de-facto standard X.509 is not sufficiently precise and flexible.
- There is still a lack of understanding what a PKI can achieve, and unrealistic hopes lead to failure.
- Legal problems. A PKI should be compatible with the diverse national legal systems. However, legal systems are slow in adapting to technical developments.

4.6.3 The naming problem

A *name* is a digital object (i.e. a bitstring) used to refer to an entity. Names should be unambiguous and unique, at least in the given context. A name can consists of several parameters, e.g. name, employer, one or several email addresses, or a photograph.

There is a trade-off regarding the amount of information contained in an entity's name. One one hand, there must be sufficient information for uniqueness of names, but on the other hand, the parameters of the name should be stable over time. For instance, an e-mail address often changes when a person changes jobs. There is also a *privacy* issue: certain information (e.g. a photograph) should not leak outside a defined environment.

Obviously a person can play the role of many different entities, e.g. as private person, as an employee or as a forum user (using a pseudonym).

4.6.4 Expiration and revocation

A certificate usually carries an expiration date, after which it is considered invalid. the concept of expiration is non-trivial. For instance, how does one prove that a digital signature was issued before or after the certificate has expired?

A certificate can also become invalid before it expires, for a number of possible reasons, including:

- Changes of certain parameters of the name (e.g. address)
- The expiration of a certain entity (e.g. a person leaving a company)
- Loss of control of the secret key corresponding to the certified public key
- Compromise of the secret key

Therefore there must be a mechanism for declaring a public key invalid. This process is called *revocation*. Note that one should speak about the revocation of a *public key*, rather then the revocation of a certificate. For a single public key, there might be several certificates.

Two approaches to the revocation of certificates are usually considered. In both approaches, to revoke a certificate, one must contact the CA who issued it.

- **Certificate revocation list (CRL).** The CA publishes regularly (say every week) a signed list of revoked certificates. Before using a certificate one must check the corresponding CRL.
- **Revalidation certificates.** The second approach is to ask the CA to revalidate a certificate each time one uses it. If a certificate (or a public key) has been revoked, the CA will not issue such a revalidation certificate.

Actually, one can argue that both approaches are essentially equivalent, differing only in the granularity of updates.

Revalidation certificates appear to be somewhat strange: Why is the certificate needed in the first place, if, whenever the certificate is used, a revalidation certificate has to be issued anyway. It seems that one could eliminate the certificate.

4.6.5 PKI key recovery

Key backup supports recovery of lost keys, e.g. due to memory lapse, dish crash, broken smart card, etc. This serves an important business need, e.g. for keys belonging to roles, dismissed/ill/forgetful employees, etc.

Key escrow concerns storing keys so that legal/government agencies may potentially use them to decrypt communications. However, this is generally unpopular.

The mechanisms used for this is to store the key with some authority (e.g. CA). One can reduce the trust in the authority by splitting the key into n shares stored by different parties where all (or m) shares must be combined to recover the key.

Regarding PKI key recovery, encryption keys can be distinguished from signing keys:

- Loss of a decryption key is problematic and should be stored somewhere.
- Loss of a signing key on the other hand is unproblematic, one can just get a new one. The old signatures can still be checked. A back-up inside a CA is not desirable (increases chance of theft).

4.6.6 PKI trust models

1. **Single CA**
   - A single CA for the entire world, where all systems are configured with the CA's public key.
   - All certificates are obtained from the directory of the organization running the CA.
   - Advantages
     - Certificate chains all of length 1
     - No need to trust recommendations
   - Disadvantages
     - No organization trusted by all countries, companies, etc.
     - Inconvenient, insecure and expensive to obtain certificates from a distant organization.
     - Periodically changing keys is good practice, but changing the CA's key means reconfiguring the world.
     - CA has a monopoly and can charge excessive fees.
2. **Single CA+RAs**
   - A single CA with multiple registration authorities (RAs) trusted by the CA to verify name/key binding and send request to the CA.
   - To authenticate requests, the CA has the key of all RAs

- Advantages
  - User interact with local RA.
  - Still need only one public key of a single CA.
  - Revocation of compromised RA key is easy (notify the CA).
- Disadvantages
  - Drawbacks of the single CA model remain.
  - Must trust all RAs to properly authenticate users.

3. **Oligarchy of CAs**
   - Like the single CA model, but with multiple CAs
   - Advantages
     - More convenient as CAs can be local.
     - Competition between CAs.
   - Disadvantages
     - Less secure than single CA model, as compromising one CA key suffices to generate bogus certificates.
     - Requires principals to have a "certificate store", which could be manipulated by adding keys.
4. **Variant 1 on oligarchy of CAs**
   - Rather than distributing all CA certificates to principals, CAs instead have cross certificates in their certificate repositories.
   - Advantages
     - Useful with multiple "domain-specific" CAs, which mutually certify each other keys.
5. **Variant 2 on oligarchy of CAs: name-spaces**
   - A *name-space* is a set of names for which a CA is trusted to certify name-key mappings.
   - Example: the ETH CA can certify `president@ethz.ch`, but not `president@white-house.gov`.
   - Can add cross-certificates.
   - Can add hierarchical name-spaces, each such name-space associated with a hierarchy of CAs.
6. **Configured and delegated CAs**
   - CAs whose keys are configured in a workstation can authorize other CAs to act as delegated CAs.
   - Both kinds of CAs completely trusted, both to issue certificate and to recommend delegates as trustworthy CAs. Principals thereby trust all certificates in the chain.
   - Advantages
     - Easy for users to obtain certificates from a local CA.
     - Principals must not be configured with all certificates.
   - Disadvantages
     - As before, compromise of any CA results in a complete compromise.
     - Strong assumption of transitive trust.

Part II
David Basin

5 Introduction

5.1 What is information security?

5.1.1 General definitions

- **Computer security** concerns the prevention and detection of improper actions by users of a computer system. The definition of *(im)proper* is central.
- **Information security** is even more general. It deals with proper use of *information*, independent of computer systems.

5.1.2 Security as policy compliance

For any system we have a *specification* that states what it is supposed to do, an *implementation* that shows how it does it, and one can consider *correctness*, i.e. does it really work.

In security, the *specification* is given by a policy, the *implementation* is a mechanism, and we speak of compliance for *correctness*.

Security policies are used to state what system behavior is, and is not, allowed, and security mechanisms are used to enforce these policies. More formally, a specification can be seen as a predicate Φ (i.e. a security property). A system P employing mechanisms is secure, if $P \vDash \Phi$.

5.1.2.1 Examples of security properties

The most traditional security properties are (CIA):

- **Confidentiality (Secrecy)**: No improper disclosure of information
- **Integrity**: No improper modification of information
- **Availability**: No improper impairment of functionality/service

Note that (im)proper must be specified individually for each system. CIA emphasis originated in military context with centralized systems.

There are several other relevant security properties, including:

- **Authenticity** (e.g. "message originated from Alice") is closely related to integrity, with a difference in emphasis.
- **Non-repudiation** (also called **accountability**) is where one can establish responsibility for actions.
- **Plausible deniability** is contrary to non-repudiation and can be seen as a weak form of secrecy.
- **Privacy**
  - **Anonymity**: Secrecy of principal identities or communication relationships
  - **Data protection**: personal data is only used in certain ways

5.1.2.2 Properties, policies and mechanisms

The distinction generally is rather fuzzy, but the following could be a definition:

- Properties are generally high-level, describing (un)acceptable system behavior
- Policies may just be the conjunction of different properties.
- Alternatively, policies may be more low-level and operational, e.g. rules like "when choosing passwords, strong passwords must be used".
- Mechanisms are concrete, e.g. implementation components.

The boundary to other non-security properties is also fuzzy. There is a large overlap between building secure systems and building correct systems.

5.1.2.3 Secure coprocessor as an example

A secure cryptographic coprocessor is a device with secure memory with crypto support protected against physical attacks. The IBM 4758 secure coprocessor supports layered applications, where each layer has an owner. Only IBM can alter the hardware or the boot system, i.e. IBM acts as a root of trust.

- Layers and owners each have a keypair.
- The private key of layers is stored in persistent storage and protected from higher layers.
- All layers are physically secured, e.g. parts of memory are read only, tamper-proof packaging, etc.
- Layer i uses private key for
  - outbound authentication,
  - certifying integrity of $i+1$st layer, and
  - assigning ownership of $i+1$st layer by certifying the owner's public key.
- The owner uses his private key to authenticate commands, e.g. load code at that layer.

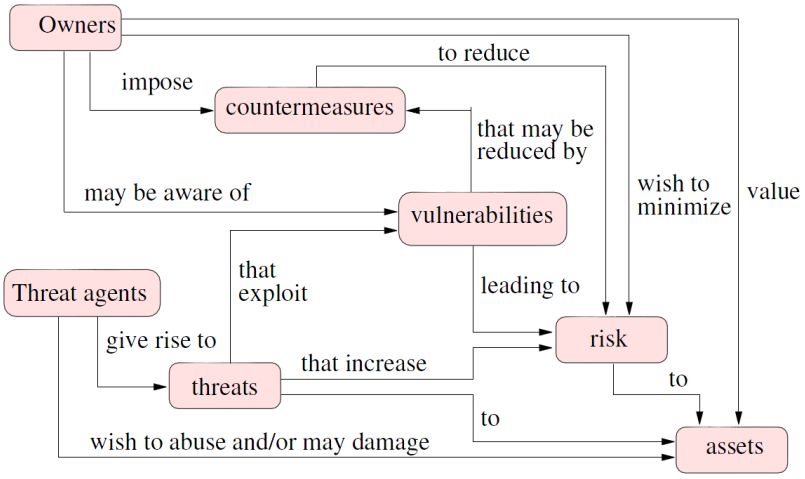
Examples of properties of a coprocessor include:

- **Outbound authentication**: It must be possible to distinguish between a message from a layer N with a particular software environment on a particular untampered device from a message from a clever adversary.
- **Inbound authentication**: Suppose A has ownership of a software layer in an untampered device, then only A or a designated representative can load code into that layer in that device.
- **Access to secrets**: Secrets belonging to layer N are accessible only by code that authority N trusts executing on an untampered device in an appropriate execution environment.

5.1.3 Security as risk minimization

Focus on *risks* from *vulnerabilities* and their *exploitation*.

Most practical approaches to information security (e.g. security audits) are oriented around such a classification. Especially for system *administration* rather than *construction*.



The risk analysis and reduction steps can be divided into two parts:

1. Analysis of existing risks
   i) Identify assets you with to protect. What are the (information) assets and their functionality?
   ii) Identify risks to these assets (the hard part). Requires understanding threat agents and their threats as well as vulnerabilities.
2. Analysis of proposed security solution.
   i) How well do proposed countermeasures reduce risk?
   ii) What other risks and tradeoffs do measures themselves bring?

6 Networks

6.1 Introduction

The internet is a confederation of networks using TCP/IP, with no global domain of trust. Different subnetworks may or may not be trustworthy, and 15+ hops for a packet to reach is destination is common. Furthermore, TCP/IP do not provide security (authentication or confidentiality). Addresses can be faked and the payload can be both read and modified. In such a setting, various questions arise, such as:

- How do we secure (encrypt and authenticate) communication?
- End-to-end? Hop-by-hop? Gateway-to-gateway?
- At which layer: link, internet, transport or application layer?

6.2 Application-managed security

Each application is responsible for setting up secure communication and enforcing least privilege (access control). The rational follows the *end-to-end principle*.

6.2.1 Advantages

- No need to trust intermediate nodes/layers/protocols.
- No need to change/configure them either, or manage their keys.
- Security decisions can be based on user ID, user data, etc.
- Examples include mail encryption and authentication (e.g. PGP or S/MIME) or SSL/TLS (endpoints are browsers and servers).

6.2.2 Problems

- Assumes mechanisms are prefect
  - Design: right use of crypto, no attacks from below, etc.
  - Implementation: no buffer overflows, injection attacks, etc.
- Assumes users are prefect
  - They understand the enterprise security policy.
  - They respect the policy, e.g. no insider fraud.
  - They properly configure the mechanisms, manage credentials, etc.
  - They are immune to social engineering and the like.
- Assumptions are clearly naive.

Further problems include the following:

- Lack of support for legacy or 3rd party applications
- Network (midbox) mechanisms are still needed, e.g. for incident response, it is simplest to restrict access at choke-points (firewall).
- Interference with midbox mechanisms. Example: mail virus scanning and archiving is best implemented as a central service, but this is difficult if mail is encrypted end-to-end.

In general application-managed security is a poor fit with the current trend to centrally managed and enforced security.

Finally, what are the actual endpoints? Could be the ends of the application, the relevant servers or even the users themselves. The answer depends on what you trust, i.e. end-to-end is more like trust-to-trust. Principle: place functionality where you can trust it.

6.3 Network-managed security

In general there are options of *where* to secure communication (end-points, etc.) and at what *level* in the network stack.

6.3.1 Implementing lower-layer/midbox security

An example is SSL (or TLS/SSH) where the OS does not change, but the applications do. The SSL API is an extension of sockets API to TCP. Another example is IPSec: The OS changes, but the applications and the TCP API remain unchanged.

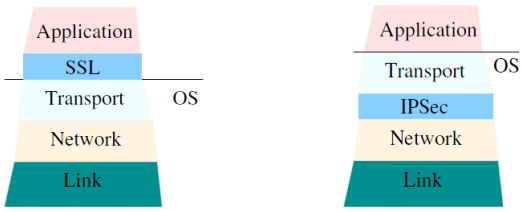


Figure 6: SSL and IPSec in the network stack.

Advantages of lower-level/midbox security are the following:

- Implement security solutions once, rather than at each application.
- Users are not bothered with security, most do not understand it either.
- Some commercial "off-the-shelf" systems do not provide security, therefore it is needed at a lower level (or some kind of wrapper).
- Some solutions are better operated/maintained at midboxes (e.g. intrusion detection systems or spam filtering which benefit from a global view of all subnet traffic).

Some examples and tradeoffs that are involved:

- SSL/TLS
  - \+ No modification to the OS, minimal changes to applications.
  - \+ Can work with user certificate (usually not in practice).
  - \- Lack of client-side authentication leads to various problems in practice (MITM attacks).
- IPSec
  - \+ Transport layer security without modifying applications.
  - \- Only authenticates IP addresses, no user authentication.
  - +/- More is possible, but requires changing API and applications.
- WPA(2)
  - +/- Secures just one link, but typically the most vulnerable one.

6.4 Protecting the network

Securing systems in just one way is not enough; protection of individual elements is usually to coarse. The principle of *defense in depth* argues for layered measures, such that if one mechanism fails, there are other fall backs. For instance, access control is implemented in firewalls, applications and the database back-ends.

This redundancy is usually beneficial, although adds complexity and cost, and contradicts the "*keep it simple*" principle. This principle says that all systems have bugs, and therefore the every executable program exposes you to attacks. Their execution should be disallowed whenever possible.

A related principle is *system hardening*, where all unneeded services are shut down, and only those required to get the job done are kept. Also, vulnerabilities should be minimized by good engineering/maintenance (i.e. code reviews, testing, patch maintenance).

6.4.1 Firewalls

Firewalls assume network adversaries as a threat model, and try do not allow them onto your machine. This establishes a trust perimeter, where the firewall acts as a boundary between untrusted and (internal) trusted machines. It is also possible to have multiple layers of trust with corresponding boundaries.

The firewall is a protection mechanism at the cross-over point which controls everything that crosses the perimeter.

There exist two kinds of firewalls:

- Packet filters
  - Essentially a router that performs access control.
  - Example: Disallow all inbound FTP connections, or only allow incoming connections to port 22 (SSH).
  - Simple to set up, although access control is coarse grained.
- Application-layer proxies
  - Control input/output/access to an application or service.
  - Can inspect content of traffic, filtering undesirable data. For instance input filtering against injection attacks, virus scanning, etc.
  - Can require additional authentication.

In general, the arguments for firewalls outweigh arguments against them, i.e. firewalls are rightfully popular.

- Pros
  - Scales better than host security.
  - One place (or more) to set up a secure policy.
  - Compatible with (insecure) legacy systems.
- Cons (mainly for packet filters)
  - Does not handle surfing/email very well.
  - Does not help when application traffic is tunneled over http, which is very common.
  - Problems with complex trust perimeters (e.g. joint ventures)

6.4.2 Example: securing a web server

A web server is a prime target for attacks, and our internal machines (e.g. database back-ends) should be protected from it. The server is better put in the DMZ, outside the main firewall, as shown in figure 7. This puts on a "sacrificial" machine, which can be restored from backups if hacked.

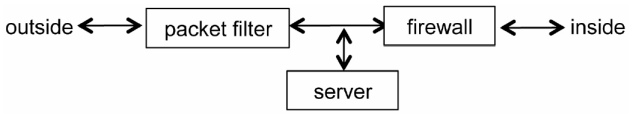


Figure 7: Where to put a web server.

6.4.3 Insider attacks

The trust perimeter is often incorrect as machines and users inside the perimeter are not always trustworthy. For instance a user adding a wireless access point to his machine will also change the trust perimeter and subvert the firewall.

Insider attacks (both unintentional and malicious) are more common than suspected. They are difficult to solve technologically while maintaining usability.

6.4.4 Intrusion detection systems

Intrusion detection is the process of monitoring and analysing system or network events for signs of possible incidents, which represent (imminent) violations of computer security policies. Possible forms:

- Host-based: monitor events of individual hosts, e.g. to identify a break-in.
- Network-based: examine network-wide traffic flow, e.g. to identify a distributed denial-of-service attack.

Intrusion detection is an example of defense in depth, and one can even combine multiple intrusion detection systems. They are also useful to spot insider attacks.
There exist various forms of intrusion detection systems, which can be grouped as follows:

- Signature based
  - Recognize patterns corresponding to known threats.
  - Example: repeated attempts to log in as root.
- Anomaly based
  - Uses machine-learning techniques to classify "normal" behavior, characterizing users, hosts, network connections or applications.
  - Compare observed events to identify significant deviations.
  - Example: number of emails sent by users, number of failed login attempts for a host, process usage, etc.
  - Can detect previously unknown threats, e.g. a break-in that initiates many network connections.

The drawbacks of intrusion detection systems include the following:

- Limited effectiveness
  - How well do rules/profiles capture future events?
  - It is not always clear what's inside the box.
- Too many false alarms
  - Handling these is costly, they are eventually ignored.
- High life-cycle costs, e.g. training and maintenance.
- Systems can be subverted
  - Arms race, especially for signature-based systems.

7 Security protocols

7.1 Basic notations

7.1.1 Security protocols

A *protocol* consists of a set of rules (conventions) that determine the exchange of messages between two or more principals. In short, it is a *distributed algorithm* with emphasis on communication. For instance:

1. $A \to B : \{A, N_A\}_{K_B}$
2. $B \to A : \{N_A, N_B\}_{K_A}$
3. $A \to B : \{N_B\}_{K_B}$

Security (or *cryptographic*) protocols use cryptographic mechanisms to achieve security objectives. For the rest of this document, the following notation is used:

- Names: A, B or *Alice*, *Bob*, ...
- Asymmetric keys: A's public key is K_A and the corresponding private key K_A^{-1}.
- Symmetric keys are denoted by K_{AB} (shared key for A and B).
- Encryption: asymmetric $\{M\}_{K_A}$ and symmetric $\{M\}_{K_{AB}}$.
- Signing: $\{M\}_{K_A^{-1}}$.
- Nonces: N_A. Nonces ("numbers used once") are fresh data items used for challenge/response.
- Timestamps: T. Denote time, e.g. for key expiration.
- Message concatenation: M_1, M_2 or $M_1||M_2$.

The fundamental event is communication between principals, e.g.

$$A \to B : \{A, T_A, K_{AB}\}_{K_B}$$

where A and B are *roles*. These roles can be instantiated by any principal playing the role. Communication is generally asynchronous (depending on the semantic model).

7.1.2 The attacker

There are several possibilities to model the attacker:

- He knows the protocol but cannot break crypto. Attacks on crypto and attacks on the communication are separate concerns.
- He is passive but overhears all communications.
- He is active and can intercept and generate messages.
- He might even be one of the principals running the protocol.

The standard symbolic attacker model known as *Dolev-Yao threat model* is mostly used in practise. In this model, the attacker is active and completely controls the network:

- He can intercept and read all messages.
- He can decompose messages into their parts, but cryptography is perfect. Decryption requires the inverse keys.
- He can construct and send new messages, any time.
- He can even compromise some agents learning their keys.

In this context, a protocol should ensure that communication between non-compromised agents achieves objectives.

7.1.3 Protocol objectives

The terminology is not completely standard, but the following are typical:

- **Entity authentication**: One party verifies the identity of a second party and that this party has recently, actively participated in the protocol ("I am here now").
  - *Agreement* is a stronger notation of entity authentication. A protocol guarantees that an initiator A has non-injective agreement with a responder B on a set of data items ds if, whenever A (acting as initiator) completes a run of the protocol, apparently with responder B, then B has previously been running the protocol, apparently with A, and B was acting as responder in his run, and the two agents agreed on the data values corresponding to all variables in ds.
  - *Injective agreement* when additionally each run of A corresponds to a *unique* run of B. The analogous notion of matching histories sometimes used.
- **Secrecy (confidentiality)**: Data available to those authorized to obtain it. For keys, this is sometimes called *key authentication*.
- **Freshness**: Data is new, i.e., not replayed from an older session.
- **Key confirmation**: One party is assured that a second party actually possesses a given key.

7.2 Problems and principles

7.2.1 Examples of kinds of attack

- **Man-in-the-middle** (or **parallel sessions**) **attack**: pass messages through to another session $A \leftrightarrow M \leftrightarrow B$.
- **Replay** (or **freshness**) **attack**: record and later re-introduce a message or part of a message.
- **Reflection attack**: send transmitted information back to the originator.
- **Oracle attack**: take advantage of normal protocol responses as encryption and decryption "services".
- **Type flaw (confusion) attack**: substitute a different type of message field.
- **Guessing attack**: protocol uses passwords and provides a means to verify a guessed password, e.g. $\{\text{known\_string}\}_K$.

7.2.2 Principles for designing cryptographic protocols

Martín Abadi and Roger Needham describe in their paper *Prudent Engineering Practice of Cryptographic Protocols* several principles:

1. Every message should say what it means. Its interpretation should depend only on its content.
2. The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not.
3. If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.
4. Be clear about why encryption is being done. Encryption is not wholly cheap, and not asking precisely why it is being done can lead to redundancy. Encryption is not synonymous with security and its improper use can lead to errors.

   If the cryptography is asymmetric, it may be obvious what is intended, with symmetric cryptography, it is generally not. Encryption is used for various purposes:

   - Preservation of confidentiality. In this case, it is assumed that only the intended recipients know the key needed to recover a message.
   - Guarantee of authenticity. Here it is assumed that only the proper sender knows the key used to encrypt a message.
   - Binding of parameters: receiving $\{X, Y\}_K$ is not always the same as receiving $\{X\}_K$ and $\{Y\}_K$. When encryption is used only to bind parts of a message, signature is enough.
   - As a source for random numbers.

5. When a principle signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of this message.

6. Be clear what properties you are assuming about nonces. What may do for ensuring temporal succession may not do for ensuring association. And perhaps association is best established by other means.

7. The use of a predictable quantity (such as the value of a counter) can serve in guaranteeing newness, through a challenge-response exchange. But if a predictable quantity is to be effective, it should be protected so that an intruder cannot simulate a challenge and later replay a response.

8. If timestamps are used as freshness guarantees by reference to absolute time, then the difference between local clocks at various machines must be much less than the allowable age of a message deemed to be valid. Furthermore, the time maintenance mechanisms everywhere becomes part of the trusted computing base.

9. A key may have been used recently, for example to encrypt a nonce, yet be quite old, and possibly compromised. Recent use does not make the key look any better than it would otherwise.

10. If an encoding is used to present the meaning of a message, then it should be possible to tell which encoding is being used. In the common case where the encoding is protocol dependent, it should be possible to deduce that the message belongs to this protocol, and in fact to a particular run of the protocol, and to know its number in the protocol.

11. The protocol designer should know which trust relations his protocol depends on, and why the dependence is necessary. The reason for particular trust relations being acceptable should be explicit though they will be founded on judgement and policy rather than on logic.

7.3 Formal methods

Formal analysis of protocols approaches *protocol correctness* as *system correctness*. A *formal symbolic model* M is build for the protocol:

- Formal: well-defined mathematical semantics.
- Symbolic: abstract away bitstrings to (algebraic) terms.
- Model: a transition system describing all actions of principals and the attacker.

Furthermore, a property Φ is specified. This property is typically a safety property, for instance that secrecy is an invariant. Correctness then is $M \vDash \Phi$. The main techniques used are theorem proving and model checking.

7.3.1 Modeling protocols as a trace set

A protocol can be formalized as an inductively defined trace set. As an example, we formalize the Needham-Schroeder protocol with a set P:

1. $\langle\rangle \in P$
2. $t, A \to B : \{A, N_A\}_{K_B} \in P$ if $t \in P$ and $\text{fresh}_t(N_A)$
3. $t, B \to A : \{N_A, N_B\}_{K_A} \in P$ if $t \in P, \text{fresh}_t(N_B)$ and $A' \to B : \{A, N_A\}_{K_B} \in t$
4. $t, A \to B : \{N_B\}_{K_B} \in P$ if $t \in P, A \to B : \{A, N_A\}_{K_B} \in t$ and $B' \to A : \{N_A, N_B\}_{K_A} \in t$
5. $t, Spy \to B : X \in P$ if $t \in P$ and $X \in \text{has}(\text{sees}(t))$

Rules 1-4 formalize the protocol steps and rule 5 the attacker model. $\text{sees}(t)$ is the set of messages in the trace t and $\text{has}(T)$ denotes the smallest set of messages inferable from the set T:

- $t \in T \implies t \in \text{has}(T)$
- $t_1 \in \text{has}(T) \wedge t_2 \in \text{has}(T) \implies (t_1, t_2) \in \text{has}(T)$
- $(t_1, t_2) \in \text{has}(T) \implies t_i \in \text{has}(T)$
- $t_1 \in \text{has}(T) \wedge t_2 \in \text{has}(T) \implies \{t_1\}_{t_2} \in \text{has}(T)$

- $t_1 \in \text{has}(T) \implies \text{hash}(t_1) \in \text{has}(T)$
- $\{t_1\}_{t_2} \in \text{has}(T) \wedge t_2^{-1} \in \text{has}(T) \implies t_1 \in \text{has}(T)$

These rules reflect the perfect cryptography assumption: Decryption requires the right key, hashing is collision free and the attacker cannot compute preimages of hash functions.

A *property* now also corresponds to a set of traces. For instance, authentication in the Needham-Schroeder protocol could be expressed as follows:

- (A authenticates B)$(t) \equiv$ **if** $A \to B : \{A, N_A\}_{K_B} \in t$ **and** $B' \to A : \{N_A, N_B\}_{K_A} \in t$ **then** $B \to A : \{N_A, N_B\}_{K_A} \in t$
- (Spy attacks A)$(t) \equiv \neg$(A authenticates B)(t)

Verification of protocols can occur in various ways. For instance it is possible to interactively verify protocols using an inductive approach. Proof scripts are constructed by hand (with machine support), and then verified by machines. Flaws come out in terms of unprovable goals.

It is also possible to use model-checking, as the inductive definition gives rise to an infinite tree, where each path from the root to a node corresponds to a trace. State enumeration can be used to find attacks, but pure state enumeration is hopelessly inefficient. However, effective model-checking tools exist and are based on automatic, algorithmic methods rather than interactively constructed proofs in a formal logic.

7.4 Protocols example

7.4.1 Kerberos

Kerberos is a mechanism for authentication in distributed systems, and one of the first successful *single sing-on* protocols. One password per session is used, and any subsequent authentication happens behind-the-scenes. The requirements when designing Kerberos at MIT where the following:

- **Secure**: Only authenticated users have access to authorized resources.
- **Single sign-on**: Each user enters a single password to obtain network services and should be unaware of the underlying protocols.
- **Scalable**: The system should scale to support large numbers of users and servers.
- **Availability**: As many services depend on Kerberos for access control, it must be reliable. Bottlenecks should be avoided.

The protocol supports the first three requirements, while the last one is achieved by replicating the Kerberos server.

7.4.1.1 Protocol overview

Kerberos makes use of the standard patter for session-key distribution using a trusted T by creating a secure channel between A and B if they each share a key with T. An overview is shown in figure 8.

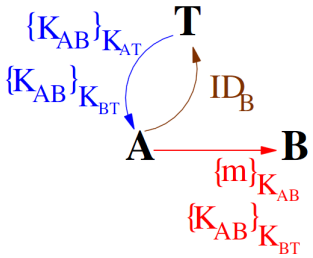


Figure 8: Overview of the Kerberos protocol.

The exact messages include nonces (or rather timestamps) to ensure recentness. The double encryption in the second message is actually not needed.

1. $A \to T : A, B, N_1$
2. $T \to A : \{N_1, B, K_{AB}, \{K_{AB}, A\}_{K_{BT}}\}_{K_{AT}}$

3. $A \rightarrow B : \{K_{AB}, A\}_{K_{BT}}$
4. $B \rightarrow A : \{N_2\}_{K_{AB}}$
5. $A \rightarrow B : \{N_2 - 1\}_{K_{AB}}$

7.4.1.2 Kerberos architecture

The architecture of Kerberos is shown in figure 9. Authentication happens using the Kerberos Authentication Server (KAS), authorization with the Ticket Granting Server (TGS) and access control takes place when the server (who provides a service) checks the TGS ticket.

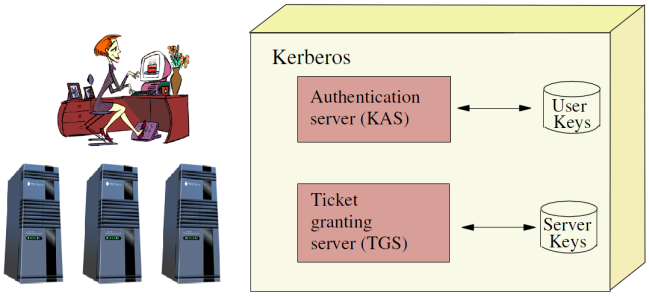


Figure 9: Kerberos architecture.

7.4.1.3 Detailed view how the protocol operates

How Kerberos operates in practise is shown in figure 10.

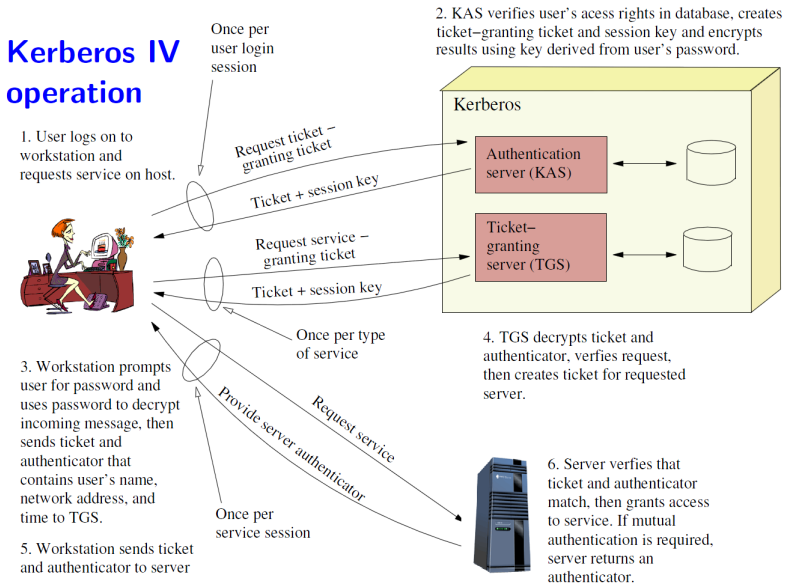


Figure 10: Kerberos operation.

Messages 1 and 2 are used for authentication and are only done once per user login session. Messages 3 and 4 achieve authorization and take place once for every type of service. Finally, messages 5 and 6 request the service, and take place once per service session.

The exact messages (slightly simplified) are as follows:

1. $A \rightarrow KAS : A, TGS$
2. $KAS \rightarrow A : \{K_{A,TGS}, TGS, T_1, \underbrace{\{A, TGS, K_{A,TGS}, T_1\}_{K_{KAS,TGS}}}_{\text{AuthTicket}}\}_{K_{A,KAS}}$

A logs onto its workstation and requests a network resource. The KAS accesses the database and sends A a session key $K_{A,TGS}$ and the encrypted ticket AuthTicket. The session key $K_{A,TGS}$ has a lifetime of several hours and $K_{A,KAS}$ is derived from the user's password. Both the user password and $K_{KAS,TGS}$ are saved in the database.

3. $A \to TGS : \underbrace{\{A, TGS, K_{A,TGS}, T_1\}_{K_{KAS,TGS}}}_{\text{AuthTicket}}, \underbrace{\{A, T_2\}_{K_{A,TGS}}}_{\text{authenticator}}, B$
4. $TGS \to A : \{K_{A,B}, B, T_3, \underbrace{\{A, B, K_{A,B}, T_3\}_{K_{B,TGS}}}_{\text{ServTicket}}\}_{K_{A,TGS}}$

Before A's first access to the network resource B, A presents AuthTicket from message 2 to the TGS together with a new authenticator, with a short lifetime (seconds). Tickets can be used multiple times, authenticators only once. This short validity of the authenticator is used to prevent replay attacks, and for immediate replays, the server stores recent authenticators.
The TGS then issues A a new session key $K_{A,B}$ (with a lifetime of a few minutes) and a new ticket ServTicket.

5. $A \to B : \underbrace{\{A, B, K_{A,B}, T_3\}_{K_{B,TGS}}}_{\text{ServTicket}}, \underbrace{\{A, T_4\}_{K_{A,B}}}_{\text{authenticator}}$
6. $B \to A : \{T_4 + 1\}_{K_{A,B}}$

Then, in the last phase, A can access the network resource B by presenting the ticket from message 4 with the new authenticator. In practise, other information for the server might be sent too. B can optionally reply to authenticate the service.

7.4.1.4 Scalability of Kerberos

A *realm* is defined by a Kerberos server (KAS+TGS). A large network may be divided into administrative realms, since Kerberos supports inter-realm protocols:

- Different servers register with each other.
- For A to access B in another realm, the TGS in A's realm receives request and grants a ticket to access the TGS in B's realm.

Protocol extension is simple, only two additional steps are required. However, for n realms, a $\mathcal{O}(n^2)$ key distribution problem arises.

7.4.1.5 Limitations of Kerberos IV

- Message 1 does not need encryption, but an attacker can flood the KAS.
- The double encryption in message 2 is redundant and eliminated in Kerberos V.
- Relies on (roughly) synchronised and uncompromised clocks.

Nevertheless, Kerberos is one of the most widely adopted authentication solutions.

7.4.2 SSL/TLS

The goals of SSL are secrecy, integrity and (optionally mutual) authentication, and it was originally developed to allow credit card payments over the internet.
The protocol consists of various subprotocols, which run on top of TCP:

- **SSL handshake**: initiates (or reinitiates existing) connection. Establishes channel with desired properties.
- **SSL record**: protocol for sending application data. Describes the way in which application data is compressed, authenticated with a MAC and how the resulting payload is encrypted.
- **Other protocols** for renegotiation ciphers and error recovery.

Although the protocol is complex, the underlying concepts are fairly simple. An *SSL session* is an association between the client and the server, with an associated state that specifies encryption method, client and server MAC secrets, encryption keys, etc. An *SSL connection* is basically a secure stream within a session.

7.4.2.1 SSL handshake

The SSL handshake works in four steps:

1. Establish security capabilities, negotiate cipher.
2. Exchange server certificate.
3. client key exchange and optional authentication of the client using a client certificate.
4. Finish establishing session.

In the hello phase, the client identifies itself (in practice, the IP address from TCP is used; the actual standard doesn't include this), and algorithm preferences P_A and P_B (chosen from P_A) are exchanged. S_{ID} is the session identifier.

1. $A \to B : A, N_A, S_{ID}, P_A$
2. $B \to A : N_B, S_{ID}, P_B$

Next, the server certificate is exchanged:

3. $B \to A : \text{certificate}(B, K_B)$

Then, the client exchange follows, where the client authenticates itself (optional), and the pre-master secret (PMS) is sent. The PMS is a pseudo-random string used to compute master secret M, in turn used to generate various keys. For the client authentication, A sends his certificate and a hash of all previous messages.

4. $A \to B : \text{certificate}(A, K_A)$ (optional)
5. $A \to B : \{\text{PMS}\}_{K_B}$
6. $A \to B : \{\text{hash}(...)\}_{K_A^{-1}}$ (optional)

In the last step, a hash of all previous messages is sent, encrypted by the symmetric keys for client and server encryption/decryption. These keys are generated from N_A, N_B, and M. Also, 4 other keys are generated, used for MACs and as initialization vectors for the stream cipher.

7. $A \to B : \{\text{Finished}_A\}_{K_{\text{client}}}$
8. $B \to A : \{\text{Finished}_B\}_{K_{\text{server}}}$

When no client certificate is used, this results in a secure channel with a pseudonym. Hence, SSL is often followed by additional client authentication *over* the secure channel.

7.4.2.2 Solutions to phishing

Phishing is a server-side authentication problem, for which various solutions are proposed:

1. Multiple communication channels
   - Combine SSL with GSM (cellular telephones).
   - Server sends user a TAN by SMS.
   - User sends TAN back over SSL connection.

   This method fails as a user authentication mechanism. A MITM attack is still possible, the attacker need not to eavesdrop on the SMS.

   However, as an additional transaction key, this method works. The bank could send "transfer $50 to UBS account 62357: confirm with JX56R5" via SMS to the user.
2. Session-aware user authentication
   - In the last round, hashes of previous messages are sent. This can be used to make the user authentication depend on both the user's credentials and on state information related to the session where the credentials are sent.

- The server can then check if the session in which it receives the credentials is the same used by the user when sending the credentials.

3. User client certificates
   - Fully prevents MITM attack as the attacker cannot sign a hash of all previous messages (including client and server certificate).
   - Unfortunately client-side certificates are rarely used.
   - Technological solutions exist, but the usability is the problem.

7.4.3 IPSec

IP security achieves end-to-end security where the endpoints are clients/servers or security gateways. It offers message authentication and encryption, protects packet headers (address and ports) and allows traffic filtering based on a policy database.

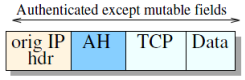
Prior to the actual communication, two IPSec peers must first determine the security services (encryption, integrity protection, authenticity), the algorithms (DES, 3DES, AES, MD5, SHA-1, ...) and session keys. This information is stored in a *security association* and can either be set up by hand, or negotiated using the *Internet Key Exchange Protocol (IKE)*. The IKE protocol is an application layer protocol, and at its heart is Diffie-Hellman with some extensions.

7.4.3.1 Protocol modes

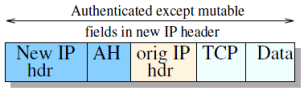
- **Transport mode**
  - Used between end-stations or between an end-station and a gateway.
  - Standard protocol encapsulation, except that the payload is encrypted/authenticated.
- **Tunnel mode**
  - Used between gateways, or from an end-station to a gateway action as a proxy for the host behind it.
  - Tunneling, where a payload protocol is encapsulated in a second delivery protocol.
  - E.g. setting up a VPN.

7.4.3.2 Headers

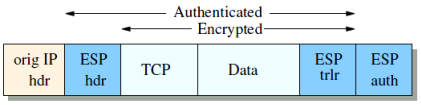
- **Authentication header (AH)**: protects the integrity and authenticity of IP datagrams, but not their confidentiality.
  - In transport mode: provides end-to-end protection between IPSec-enabled systems.



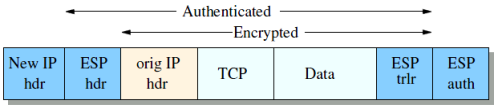
  - In tunnel mode: new outer headers also protected and may be different.



- **Encapsulating security payload (ESP)**: protects confidentiality and optionally also integrity.
  - In transport mode: encrypts the payload, but leaves the header untouched.



  - In tunnel mode: entire IP datagram encapsulated within ESP.



7.4.3.3 Security policy database

IPSec uses a security policy database to decide what to do with each packet, e.g. do we allow these peers to communicate, ESP or AH, etc. The SPD is configured by an administrator using rules of the form ⟨Condition, Action⟩, where the condition gives an IP address range, ports, protocols, flags etc. and the action is for instance discard, bypass IPSec, apply IPSec with given security association, run IKE to set up a new security association, etc.

7.4.3.4 Aside: Perfect forward secrecy

Perfect forward secrecy is the property in which an attacker who records an encrypted conversation cannot later decrypt it, even if he has since compromised the long-term cryptographic secret of each side.

7.4.4 Conclusions

"Real word" security protocols are complex.

- Complexity is in part due to standardization process.
- The problems themselves are complex in their full generality.
- But complexity is security's worst enemy.

8 Access control

8.1 Basic concepts

8.1.1 AAA

- **Identification**: The process of associating an identity with a subject.
- **Authentication**: The process of verifying the validity of something claimed by a system entity (typically their identity).
- **Authorization**: An authorization is a right or permission that is granted to a system entity to access a system resource.
- **Access control**: Protection of a system resource against unauthorized access. Namely, the process and controls for regulating the use of the system resource by a security policy whereby access is restricted to those entities authorized by the policy.

These concepts are independent.

8.1.2 Policies and models

A *security policy* defines what is allowed; that is, which system executions are (in)acceptable. A *security model* provides a formal representation of a class of systems and their behavior, highlighting their security features at some chosen level of abstraction.

Policies and models are often formalized in terms of *system states*, i.e. the collection of current values of all memory locations, storage, registers and other components. The part addressing protection is called the *protection state*.

Let P be the set of protection states and $Q \subseteq P$ be the protection states in which the system is authorized to reside in.

- When the system is in state $s \in Q$, the system is *secure*.
- For $s \in P \setminus Q$, the system is *insecure*.

In this model, a security policy characterizes Q. Hence, a security policy partitions the states of the system into authorized or secure states, and unauthorized or insecure states. Now, a *security mechanism* prevents a system from entering $P \setminus Q$, and a system is secure, if it starts in an authorized state and cannot enter an unauthorized state.

8.2 Access control matrix model

The *access control matrix model* is a simple framework for describing a protection system by describing the *privileges* of *subjects* on *objects*. This constitutes as a model, and, when implemented, as a mechanism.

The protection state (relative to a set of privileges P) is a triple (S, O, M):

- A set of current subjects S.
- A set of current objects O.
- A matrix M defining the privileges for each $(s, o) \in S \times O$, i.e. a relation $S \times O \times P$, or equivalently a function $S \times O \to \mathcal{P}(P)$.

The state transitions are modeled by a set of command, where each command is expressed in terms of six primitive operations:

1. **enter** p **into** $M(s, o)$ (for $p \in P$)
2. **delete** p **from** $M(s, o)$ (for $p \in P$)
3. **create** subject s
4. **destroy** subject s
5. **create** object o
6. **destroy** object o

For instance, the following command would be possible:

command *CreateFile*(s, f)
  create object f
  enter Own **into** $M(s, f)$
  enter R **into** $M(s, f)$
  enter W **into** $M(s, f)$
end

We now have a transition system with the following semantics:

- Write $(S, O, M) \vdash_c (S', O', M')$ to denote a transition associated with the command c.
- A starting state $st_0 = (S_0, O_0, M_0)$ and the set of commands C determine a state-transition system.
- So a model describes a set of system traces, i.e.

$$st_0, st_1, st_2, st_3, \ldots$$

  where $st_i \vdash_{c_i} st_{i+1}$ for $c_i \in C$.

The matrix model defines a family of systems, and this model is parametrized by the set of privileges, the set of commands, the initial state and a universe of subjects and objects. This (or any other) security model can now be used for several things:

- Specify concrete systems as instances.
- Verify that instances are secure with respect to a given property.
- Reason about general properties of all systems in the class.
- Use it as a bases for mechanism design.

Authorization can be defined as a set of authorized states. However, authorization may depend on the past, in ways not reflected in the current protection state. In this case, a more general definition is required whereby a policy defines a *set of authorized traces*. A system is then secure, when every possible trace is authorized. A simple example of a property that requires this trace-based definition: "A user can watch a video at most 4 times".

Such history-based policies are not well suited for the matrix model, as they require relevant history to be represented as matrix entries (e.g. how often a video has been played).

8.2.1 Access matrix data structures

There exists various ways to represent the access matrix, either using a two-dimensional object, or a set of one-dimensional objects. Figure 11 shows an overview of these possibilities.

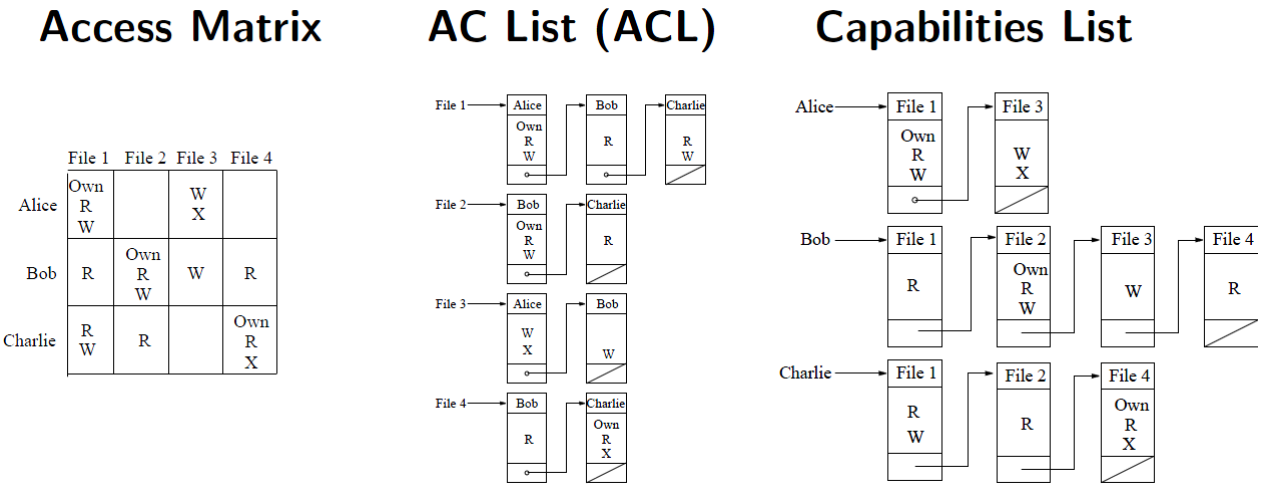


Figure 11: Access matrix data structures

8.2.1.1 Access-control lists (ACL)

Lists are used to express the view of each object o: The ith entry in the list gives the name of a subject s_i and the rights r_i in $M(s_i, o)$ of the access matrix. A standard example for AC lists are files.

The implementation of ACL is straight forward: The ACL is associated with each object, and typically maintained by the OS, middleware, server, etc. Then, the user can just be checked against the list. However, this relies on authentication (need to know the user).

ACLs are usually used for *discretionary access control*, where owners have the (usually sole) authority to grant or revoke rights to the objects they own.

8.2.1.2 Capability lists

This corresponds to the subject view of the access matrix, and a *capability* is essentially a pair of an object and an operation.

Obviously, users should not be able to forge capabilities:

- Centralized system
  - OS manages capabilities in a protected address space.
- Distributed system
  - Pair protected using cryptography, e.g. signatures.
  - Reference monitor checks ticket.
  - Need not know identity of user or process, at least if transitive delegation is allowed.

Capabilities are used less frequently, and typically in a distributed setting (e.g. Kerberos ticket and authenticator).

Compared with ACLs, the situation looks as follows:

- **ACLs**
  - \+ ACLs are compact and easy to review.
  - \+ Deleting an object is simple.
  - \- Deleting a subject is more difficult.
  - \+ Delegation is possible in a discretionary access control setting: Owners have the (usually sole) authority to grant or revoke rights to the objects they own.
- **Capabilities** (in particular when distributed)
  - \+ Delegation is easy.
  - \- Revocation is difficult.
  - \- Not so compatible with object-oriented view of the world.
  - \- In general, difficult to know who has permissions on an object.

8.3 Role-based access control (RBAC)

Security policies should be highly scalable as a very large number of subjects and/or objects is not uncommon in practise. However, AC matrices (both ACLs or CLs) scale poorly and are difficult to maintain.

The key insight to solve this problem is that user typically have *roles* within an enterprise and these can be used for access control.

- Create role for job function in the enterprise.
- Associate each role with a set of permissions.
- Assign users to roles based on their function and associated responsibilities.

The AC-matrix before was defines as $M \subseteq S \times O \times P$ where P is a privilege like "read" or "write". We now recast this matrix as $AC \subseteq \text{Users} \times \text{Permissions}$. This gives us *declarative access control*, i.e. authorization is specified by a relations: A user is granted access if and only if he has the required permission.

$$u \in \text{Users has } p \in \text{Permissions} \iff (u, p) \in AC$$

Role-based access control now decouples users and permissions by introducing *roles*. This can be formalized by a set of roles, and the relations $UA \subseteq \text{Users} \times \text{Roles}$ and $PA \subseteq \text{Roles} \times \text{Permissions}$ where

$$AC = PA \circ UA$$

This notion can be extended to *role hierarchies* (i.e. a partial order $\geq$) on roles. Now we have

$$AC = PA \circ \geq \circ UA$$

8.3.1 Advantages and disadvantages

RBAC has many advantages, which make it enormously popular. For instance:

- Roles are stable semantic concepts within an enterprise. User come and go, but activities/functions change only slowly.
- Supports different well-known security principles:
  - **Least privilege**: achieved by assigning only those permissions required for the tasks to the role.
  - **Separation of duty**: achieved by ensuring mutually exclusive roles (in terms of users) must be invoked to complete a task.
- Assignments are understandable from the business perspective, which eases both administration and audit.

However, there are some limitations:

- Can not base access-control decisions on the system state.
- Can not specify *obligations* incurred by the granted access.
- No way to enforce *sequences* of events using RBAC.

To overcome this problems, RBAC is often combined with *programmatic access control*. For example: "A user in the role customer can withdraw money from an account (RBAC) when he is the owner and sufficient funds are available (programmatic access control).

8.4 Discretionary access control (DAC)

The principle behind DAC is that users own resources and control their access. In particular, this means

- Owners may change an object's permissions at his discretion.
- This allows the direct delegation of rights.
- Owners may even be able to transfer ownership to other users.

This is rather flexible, but also open to mistakes, negligence, or abuse.

- Requires all user to understand the mechanisms and to understand and respect the security policy.
- No control of information dissemination.

8.5 Mandatory access control

In MAC, access control decisions are formalized (and controlled) by comparing **security labels** indicating sensitivity/criticality of objects with **formal authorization**, i.e. security clearances, of subjects.

In this model, a system-wide access restriction to object is specified. It is *mandatory* because subjects may not transfer their rights. This shifts the power from the users to the system owner.

The model has a military background with **clearance levels** such as *top secret*, *secret*, *confidential*, and *unclassified*.

Besides clearance levels, there are also **compartments** (or **categories)**, which specify a domain for a need-to-know policy.

8.5.1 Lattice

A set S with a relation $\leq$, written $(S, \leq)$, is *partially ordered* if $\leq$ is

- Reflexive: for all $a \in S$, $a \leq a$.
- Transitive: for all $a, b, c \in S$, if $a \leq b$ and $b \leq c$, then $a \leq c$.
- Anti-symmetric: for all $a, b \in S$, if $a \leq b$ and $b \leq a$, then $a = b$.

A partially ordered set $(S, \leq)$ where for all $a, b \in S$ there exists a *least upper bound* $u \in S$ and a *greatest lower bound* $l \in S$, i.e.

- $a \leq u, b \leq u$ and $(a \leq v \wedge b \leq v) \implies (u \leq v)$ for all $v \in S$.
- $l \leq a, l \leq b$ and $(k \leq a \wedge k \leq b) \implies (k \leq l)$ for all $k \in S$.

Latices are well suited for access control, because questions like "Given two objects with different labels, what is the minimal label a subject requires to be allowed to read both objects" can uniquely be answered. This is useful for a need-to-know policy, where each subject is assigned a label reflecting *least privilege* required for this function. Classifications can be combined, e.g.

$$\text{Class} = \langle \text{level}, \text{compartment} \rangle$$

The dominance relation is defined componentwise (lattice product) on linearly-ordered levels and subset-ordered compartments:

$$(r_1, c_2) \leq (r_2, c_2) \iff r_1 \leq r_2 \wedge c_1 \subseteq c_2$$

8.5.2 Bella-LaPadula (BLP) model

The BLP models security policies for confidentiality, with authorization for subjects to *read* and also to *write* objects. The main challenge is preventing *information flow*.

Access decisions satisfy the following properties:

- **Read-down** (also called simple security property): A subject with label x_s can only read information from an object with label x_o if x_s dominates x_o.
- **Write-up** (also called the *-property): A subject with label x_s can only write information to an object with label x_o if x_o dominates x_s.

The main insight in this model is that *prohibiting write-down* is essential for confidentiality as otherwise information can effectively be reclassified.

The BLP model (if implemented securely), no information leakage is possible:

- No read-up and no write-down prevent subjects from simultaneously having read access to information at one level and write access to information at a lower level.

However, this also prevents legitimate communication from high-level subjects to lower-level ones. Possible solutions include

- Temporarily downgrade the subject's security level.
- Identify a set of trusted subjects that may violate the *-property.

8.5.3 Other models

8.5.3.1 Integrity models

Analogy with water, where water can have different qualities which is influenced by many factors. In the computing world, programs process data from multiple sources, and integrity model try to answer what the integrity of the result is.

The simplest such model is the *low watermark* model, where an objects integrity is the lowest integrity level of all objects it interacts with. However, this is too simplistic, as everything gets contaminated.

8.5.3.2 Biba integrity model

This model is dual to BLP in the sense that information may only flow from high-integrity regions to low-integrity regions.

- Write-down: The writer's label must dominate the object's.
- Read-up: The object's label must dominate the reader's.

This is reasonable, as a manager can overwrite subordinate's data, but untrusted software may not taint data or other programs. The integrity is preserved.
However, what if we want both confidentiality and integrity?

- Only read and write at the same classification.
- Use BLP for classifying some data, Biba for others.
- Allow read-down and combine Biba with other mechanisms.

8.5.3.3 Chinese wall model

The *Chinese wall model* is a confidentiality model to avoid conflict of interest situations. Each object is associated with a company and the set of companies is partitioned into conflict of interest classes (e.g. a class for banks, etc.).

A subject may access a company's objects if and only if he has not previously accessed data of a different company in the same conflict of interest class, i.e., he is on the right side of the wall. This implies that access rights may change with each access.

8.6 Limitations of access control models

While access models restrict operations like read and write, information may still be revealed in various other ways. Examples of such information leaks are

- Error messages to the user (e.g. "file not found")
- Timing behavior based on CPU usage (send a 0-bit with 100% of CPU, and a 1-bit via sleep)
- Locking and unlocking files
- Encode information in the invoice sent for services rendered

8.6.1 Interface models

An alternative to this are *interface models*, where not operations are restricted, but rather restrictions on the systems' input/output.

Interface models specify a system (security) model with an interface that provides functions from inputs to outputs. *Non-inference* specifies a security property: A group of subjects using a set of commands is *non-interfering* with another group of subjects if whatever the first group does has no effect on what the second group can observe.

The idea is that "low" output should not "depend" on "high" inputs. This can be formalized by associating security levels with subjects s and inputs to (deterministic) functions f_s. If s has level l, then the result of f_s may only depend on inputs dominated by l.

A possible formalization based on Goguen and Meseguer is the following. Let $\text{out}(u, I)$ be the system's (deterministic) output function, whose value is the output to user u generated by the input history $I = i_1.\cdots.i_n$. Let $\text{purge}(u, I)$ purge from I all inputs i_j where the clearance of the user who input i_j dominates u's clearance.

In this setting, a system is non-interfering if its output to low-level users is independent from input from high-level users. Namely, for all input histories I and users u:

$$\text{out}(u, I) = \text{out}(u, \text{purge}(u, I))$$

8.6.1.1 Issues with non-inference

- What are appropriate formalizations for realistic systems?
- Non-inference appears very strong. A secret password will influence the system output, e.g. the login succeeds or fails.

8.7 Monitor-based enforceability

Most standard enforcement mechanisms are based on *execution monitoring*, which is a very general idea and includes security kernels, reference monitors, firewalls and most OS and hardware-based enforcement mechanisms. A central question is what types of policies are actually enforceable this way?

Let Ψ denote the universe of all possible sequences, either finite or infinite. Executions are represented at some abstraction level, where sequences of actions, program states, state/action pairs, etc. are used. A security policy P is specified as a predicate on *sets of executions*, i.e. characterizes a subset of $\mathcal{P}(\Psi)$.

A system S defines a set $\Sigma_S \subseteq \Psi$ of actual executions and satisfies P if and only if $P(\Sigma_S)$ is true. This definition is again very general and includes e.g. non-inference.

Execution monitoring (EM) works by monitoring the target execution. So any enforceable policy P must be specified as

$$P(\Pi) \quad \Longleftrightarrow \quad \forall \sigma \in \Pi : \hat{P}(\sigma)$$

$\hat{P}$ formalizes the criterion used by the EM to decide whether a trace σ is acceptable, i.e. whether or not to abort.

For a policy P exists a EM enforcement mechanism, if P is a *safety property*. A policy P is such a safety property, if three requirements are fulfilled.

1. P must be a **property** formalizable in terms of a computable predicate $\hat{P}$ on executions. A set is a *property* if and only if set membership is determined by each element alone, and not by other members of the set.

   Note that not all security policies are properties, e.g. non-inference is not a property. This is because a system is non-inferring, if *for all input histories* I and user u, $\text{out}(u, I) = \text{out}(u, \text{purge}(u, I))$.

2. **Prefix closure**: If a trace is not in $\hat{P}$, then the same holds for all extensions. Or conversely, if a trace is in $\hat{P}$, so are all prefixes.

   This is necessary as mechanisms cannot decide based on possible future executions.

3. **Finite refutability**: If a trace is *not* in $\hat{P}$, we must detect this based on some finite prefix.

Safety properties are a class of temporal properties, essentially stating that noting bas ever happens. Invariants are an important special case of safety properties.

EM mechanisms can be implemented by security automata (another important security model).

9 Privacy

9.1 Definitions

- **Confidentiality**: no unauthorized access to information.
- **Anonymity**: identities of communicating agents are secret.
- **Confidentiality**: collected data only used for limited, pre-defined purposes.

Privacy includes both anonymity and data protection.

9.1.1 Anonymity

You are only anonymous within a group if your actions cannot be distinguished from the actions of others in the group. This group is called the *anonymity set*, and should be as large as possible: You cannot have anonymity by yourself.

Anonymity has several use cases:

- Socially sensitive communication, e.g. disease or crime victim chat rooms.
- Law enforcement: anonymous tips or crime reporting.
- Corporations: hiding collaborations with divisions or partners.
- Political dissidents: criticizing their governments.
- Governments: e.g. for politically sensitive negotiations, whistle-blowing, etc.
- Criminal activities.

Pseudonyms are a lightweight mechanism to achieve unobservability/anonymity, i.e. provide confidentiality for the principals' identities. In some cases, it is possible to link to the actual identity.

There are various possibilities to achieve anonymity:

- **Multicast**, radio transmission, ring network: The addressee must be identified by an attribute that is visible to him while invisible to others. For instance, the attribute could be encrypted with the public key of the addressee, and every recipient of the multicast decrypts all messages.

  Obvious drawbacks are scalability, denial of service, etc.

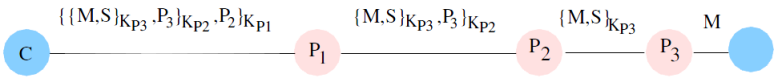
- **Proxies**: Instead of direct communication with a server, the user talks to a proxy which then talks to the server. This approach has two major weaknesses:

  – The proxy knows everything.

  – Traffic analysis is possible.

  A possible solution to this is a cascaded proxy chain.

- **Cascaded proxies with encryption**: The client encrypts its message and the destination address with the key of the proxy, which decrypts and makes the request. Again, this can be generalized to cascaded chains of proxies. In this case, each proxy only knows the previous and next hop, and all stages are encrypted.



  Now, one uncompromised proxy is enough to still ensure anonymity. However, traffic analysis is still possible. This can be solved by mix networks.

9.1.2 Mix networks

A mix network builds an *anonymous channel* and is designed to work in an environment with an active attacker who can:

- Learn the origin, destination(s), and representation of all messages in the communication system.
- Inject, remove or modify messages.
- However, it is not possible to determine anything about the correspondences between a set of encrypted items and the corresponding set of unencrypted items, or create forgeries.

To send a message M to an agent at address A, one sends

$$\{R_1, \{R_0, M\}, A\}_{K_1}$$

to the mix, where R_i is a random string and K_1 the mix's public key. The mix then forwards $\{R_0, M\}$ to A. So far, this is just a variant of a single proxy using randomized encryption. However, the mix also performs additional operations to foil traffic analysis.

- Agents/mixes work with uniformly sized items. Messages are split or padded into fixed-sized blocks.
- The order of arrival is hidden by outputting items in batches. Either a fixed ordering (e.g. lexicographic) or random ordering can be used.
- Repeated information must be blocked, i.e. mixes filter duplicates, cross-checking across batches.
- Sufficient traffic from a large anonymity set is required. Solution involves clients regularly sending/receiving dummy messages. This overhead is only sensible against a global observer who can measure traffic at both endpoints.

As a single proxy, a single mix has the weakness, that if compromised, the attacker knows everything. This can be lessened by forming mix networks, each mix in a different "administrative domain".

9.1.2.1 Receipts

If desired, a mix can return a receipt Y for each message received

$$Y = \{C, \{R_1, \{R_0, M\}_{K_A}, A\}_{K_1}\}_{K_1^{-1}}$$

for C a large, known constant. A participant can later prove that he sent a message by supplying

- $X = \{R_0, M\}_{K_A}, A$.
- The retained string R_1.

9.1.2.2 Untraceable return address

To reply to an anonymous sender x with return address M in the case of a single mix with key K_1:

- The sender includes the "return address": $\{R_1, A_x\}_{K_1}, K_x$
  - R_1 is a random string that can also be used as a shared key.
  - K_x is a fresh public key, created for this purpose..
  - A_x is x's actual address.
- Receiver sends a reply to the mix: $\{R_1, A_x\}_{K_1}, \{R_0, M\}_{K_x}$.
- The mix transforms this to $\{\{R_0, M\}_{K_x}\}_{R_1}$ and sends it to A_x.
  - The encryption with R_1 is used to mask the input/output correlation.

9.1.2.3 Summary of mix networks

- Very high degree of anonymity
  - No correlation between mix input and output.
  - Only some nonzero fraction of the mixes need to be honest.
  - With enough dummy traffic, the anonymity set is the entire network.
- Drawbacks
  - Network delays.
  - Multiple encryption.
  - Network overhead, though in networks with substantial traffic, dummy messages can be reduced or even eliminated.

9.1.3 Crowds

Crowds are developed to achieve anonymous surfing by randomization. No cryptography is used, which reduces the overhead and eliminates the key distribution problem. Crowds are peer-oriented, where each client (sender) also acts as server (receiver), and messages are sent on random paths, through other senders. This ensures anonymity of the sender and the communication relationship, however, there is no server anonymity, but this is often unimportant.

Each user has an associated anonymizing process, called a *jondo*, which acts as both a client for the user and as server for other jondos. The user can join or leave the crowds by registering with a server (called a *blender*). To send a (http) request (for a user or another jondo), a biased coin is flipped, and depending on the result, the request is either sent to the web-server, or to a randomly chosen jondo in the crowd. This process is repeated until the server is reached. Each jondo also records information needed to return replies.