The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Week 10 (13s1)

Ihor Kuz, Felix Rauch, Manuel M. T. Chakravarty & Gernot Heiser

Security in Distributed Systems

An important aspect of dependability in distributed systems relates to security. There are two security related aspects of dependability: *confidentiality* and *integrity*. In a system that offers confidentiality, information will only be disclosed to (and services will only be provided to) authorised parties. A system that provides integrity will ensure that alterations (to data or services) can only be made in an authorised way. Furthermore, improper alterations will be detectable and recoverable.

Confidentiality and integrity are generally provided through a combination of *secure communication* and *authorisation*. Secure communication is concerned with providing a secure communication channel between entities (such as users and processes) in a distributed system. A secure channel provides confidentiality in that data sent over such a channel will not be disclosed to unauthorised third parties. Likewise the integrity of data being sent over secure channels is protected because it cannot be tampered with by unauthorised third parties. Authorisation is concerned with allowing entities to only access those resources that they are entitled to access. This requires determining the identity of entities in the system and keeping track of which resources they are allowed to access, as well as monitoring which resources they attempt to access and preventing access to unauthorised resources. By definition authorisation is required for both confidentiality and integrity.

Threats

Confidentiality and integrity are not difficult to achieve in a system where no one attempts to perform unauthorised actions. Security becomes important when entities (inside or outside the system) do attempt to perform unauthorised actions (such as accessing or modifying data). The goal of security in a distributed system is to protect the services and data against security threats.

We distinguish between four types of security threats.

Interception: An unauthorised party attempts to gain access to a service or data. Examples of interception include eavesdropping, breaking into a system, and copying data.

Interruption: Services or data become unavailable, unusable, destroyed, etc. Examples of interruption threats include denial of service attacks, deletion of data, and corruption of data.

Modification: Unauthorised changing of data or tampering with a service so that the service no longer adheres to its specifications. Examples of this include intercepting and changing transmitted data (e.g., changing the parameters of an RPC call), and changing a program so that it performs differently than originally intended (e.g., logs user data).

Fabrication: Generation of additional data or activity that would normally not exist. Examples of fabrication include adding an entry to a password database and replaying previously sent messages.

The last three threats have in common that they all include some form of data falsification.

Attacks

We distinguish between two forms of attacks: *active attacks* and *passive attacks*. Passive attacks are mainly based on observation without altering data or compromising services, they represent

the interception and interruption forms of security threats. The simplest form of attack is *browsing,* which implies the non-destructive examination of all accessible data. This leads to the need for confidentiality and the *need-to-know principle.* Related is the *leaking* of information via authorised accomplices, which leads to the *confinement problem.* More indirect are attempts to infer information from traffic analysis, code breaking, and so on.

In contrast, active attacks alter or delete data and may cause service to be denied to authorised users. They represent the modification and fabrication forms of security threats. Typical active attacks attempt to modify or destroy files. Communication related active attacks attempt to modify the data sent over a communication channels.

Attacking the Communication Channel. Because of its networked nature, the communication channel presents a particularly important vulnerability in distributed systems. As such, many of the threats faced by distributed systems come in the form of attacks on their communication channels. We distinguish between five different types of attacks on communication channels.

Eavesdropping attacks involve obtaining copies of messages without authorisation. This could, for example, involve sniffing passwords being sent over the network.

Masquerading attacks involve sending or receiving messages using the identity of another principal without their authority. In a typical masquerading attack the attacker sends messages to the victim with the headers modified so that it looks like the messages are being sent by a trusted third party.

Message Tampering attacks involve the interception and modification of messages so that they have a different effect than what was originally intended. One form of the message tampering attack is called the *man-in-the-middle* attack. In this attack the attacker intercepts the first message in an exchange of keys and is able to establish a secure channel with both the original sender and intended receiver. By placing itself in the middle the attacker can view and modify all communication over that channel.

Replay attacks involve resending intercepted messages at a later time in order to cause an action to be repeated. This kind of attack can be effective even if communication is authenticated and encrypted.

Denial of Service attacks involve the flooding of a channel with messages so that access is denied to others.

Mechanisms

In order to deal with security threats a distributed system must have appropriate mechanisms in place. The four main mechanisms used to protect against security threats are:

Encryption (cryptography): transforms data into something an attacker cannot understand. Encryption prevents entities from being able to read or write data that they are not authorised to read or write. Note that they may still be able to access the data, but they cannot do anything with it. Encryption techniques also help to determine if data has been modified.

Authentication: verifies the claimed identity of an entity (a user, server, client, etc.). In order to determine if an entity is authorised to access particular data or services it is necessary to know who the entity is. Examples of authentication mechanisms include passwords, chip cards, and biometric identification.

Authorisation: determines what actions an authenticated entity is authorised to perform. Authorisation mechanisms are responsible for tracking who is authorised to access what, tracking who is attempting to access what, and preventing unauthorised access to data and services.

Auditing: traces which entities access what. Although auditing does not directly protect against security threats by preventing attacks, it is necessary in order to determine what went wrong when an attack could not be prevented.

Two lesser mechanisms can also help to improve security if they are used in combination with the above mechanisms, however, on their own they provide little benefit besides a false sense of security.

Obscurity: counting on system details being unknown. Obscuring the actual system implementation details makes finding weaknesses and attacking the system harder. Important to note here is that the underlying mechanisms must be secure for obscurity to work. Simply expecting a system to be secure because its inner workings are obscured provides very weak protection. As soon as details about the system become known, the security gained by obscurity becomes worthless.

Intimidation: counting on fear to keep a system safe. This approach often manifests itself as tough laws against 'cybercrime', and the threat of real world punishments. It is always a good idea to have a good security system in place, even if there is a high level of intimidation.

Policy

Implementing security is always a question of tradeoffs. In particular it is important to consider that the cost (in terms of money, effort, performance penalty, decreased scalability, etc.) of the security mechanisms deployed should not overshadow the value of the data and services protected. As such, it is generally not necessary for a system to be protected against all possible security threats. Instead it is necessary to determine a *security policy* and design the security of a system according to this policy. A security policy is a statement of security requirements. It describes the actions that the entities in a system are allowed to take and the actions that are prohibited. It also states which threats the system must be protected against.

Design Issues

There are a number of issues that must be taken into account when designing a secure distributed system. The first issue is the *focus of control*. This addresses the question of what the system protects against. In particular the system can protect against *invalid operations*, *unauthorised invocations*, and *unauthorised users*. In the first case the security mechanisms will focus on the protection of data in the system and ensure the integrity of the data (e.g., making sure that no operation leaves the data in an inconsistent state). In the second case the security will focus on specifying which operations can be performed and by whom. Depending on granularity the security could, for example, be at the method, interface, or object level. Finally the third approach focuses on the users of the systems and the roles they fulfill. Security is, therefore, based on the authorisation of users and their roles to access data and services.

The second design issue is the *layering of security mechanisms*. In a distributed system security mechanisms can be implemented at many different levels, including the application level, the middleware level, the network level, the OS level, and the hardware level. What level the mechanisms are implemented at depends on how much trust is placed in the software (or hardware) at that level (and the levels above it). For example, if the security mechanisms are placed in the OS, then the applications must trust the OS and the middleware with their data. On the other hand, if the security mechanisms are build into the application itself then the trust does not have to be placed in the middleware, OS, or network hardware.

Closely related to the previous issue, the issue of *distributing security mechanisms* deals with minimizing the *trusted computing base* (TCB). The TCB of a system consists of those parts of a system that must be trusted in order for the system to be secure. In other words, the TCB is that part of the system that is able to compromise security. Ideally the TCB should be as small as possible. Sometimes this may require having to implement key middleware or OS services yourself

to prevent the whole middleware or OS from becoming part of the TCB. It may also be necessary to physically separate security sensitive services from other parts of the system so that those other parts do not become part of the TCB.

Finally, an important issue (not just with regards to security) is *simplicity.* Simplicity of a system contributes to trust. The simpler a system is, the easier it is to understand, and the more trustworthy it becomes. Unfortunately it is generally difficult to build a simple and secure system, since security mechanisms usually add much complexity to systems.

Cryptography

The basic idea behind modern cryptographic techniques is the following: An *encryption* function maps a *cleartext* (or *plaintext*) T to a *ciphertext* (or *cryptogram*) C. The encryption function is *well-known,* but parameterised with a *key* K and it is infeasible to reconstruct the cleartext from the ciphertext without knowledge of the key. More formally, we have

$$E(K_E, T) = E_{K_E}(T) = \{T\}_{K_E}$$

for the encryption of T with a key K_E and

$$D(K_D, C) = D_{K_D}(C) = \{C\}_{K_D}$$

for the decryption of C with a key K_D. For a *cognate* (matching) key pair K_D, K_E, we have

$$D(K_D, E(K_E, T)) = T$$

Properties of encryption functions

The art of designing cryptographic algorithms lies in finding a function E that itself is easy to compute, but for which it is hard to compute T from $\{T\}_{K_E}$ without knowing a matching decryption key K_D for K_E. By "hard to compute" we mean that, with the computing resources of a possible attacker, it must take at least hundreds of years to reverse E without knowledge of K_D or, alternatively, to compute K_D. Such functions are known as *one-way functions.*

Modern cryptography usually assumes that the functions E and D are well known. Under these circumstances, the cipher must be resilient to the following forms of attacks:

- *Ciphertext only attacks,* where the attacker is only in possession of ciphertexts $\{T\}_{K_E}$
- *Known plaintext attacks,* where the attacker has access to matching cleartext-ciphertext pairs $(T, \{T\}_{K_E})$
- *Chosen plaintext attacks,* where the attacker can access ciphertexts $\{T\}_{K_E}$ for cleartexts T chosen by the attacker
- *Brute-force attacks,* where the attacker guesses keys by an exhaustive search through the key space

The latter is important as, in practice, keys are of finite length. The longer the keys, the more time is required to complete a brute-force attack—in fact, the time to guess a key is exponential in the number of bits comprising the key. However, long keys usually make E and D more expensive. In practice, a balance between the required security and the available computational resources has to be found. Moreover, any strong encryption function must be secure against any systematic attacks that are significantly faster than an exhaustive search of the key space.

The following is a list of properties that a good crypto system should possess:

- The information contained in a single input bit should be distributed over large number of output bits.

- Encryption and decryption functions should be fast to compute, and ideally, suited to implementation in hardware.
- They should be easy to use.
- They should not depend critically on users selecting "good" keys.
- They should have been heavily scrutinised by experts.
- They must be based on operations which are provably "hard" to invert.

Types of ciphers

Generally, cryptographic algorithms are categorised by two parameters. Firstly, we distinguish between *symmetric* and *asymmetric* ciphers. For symmetric ciphers, we have $K_E = K_D$, i.e., a single key is used for both encryption and decryption; whereas for asymmetric ciphers, the keys differ and, most importantly, computing one key from the other is computationally as infeasible as breaking the cipher in the first place. The latter property makes asymmetric ciphers a basis for so-called *public key* crypto systems, where the encryption key is not held secret but published.

Secondly, we distinguish between *block* ciphers and *stream* ciphers. The former encrypt fixed sized blocks of data, one at a time, whereas the latter encrypt a bit stream, bit by bit. Before moving on to some concrete ciphers, let us discuss these categories in some more detail.

Symmetric ciphers. Symmetric key crypto systems, which use the same key for encryption and decryption, are also called *secret key* systems since the key may only be known to the communicating parties. The main benefit of symmetric ciphers is that the algorithms are usually fast and can, therefore, be used to encrypt large amounts of data. Their disadvantage is that a secure channel is needed so that two communicating parties can establish a shared secret key. Furthermore, for any two communicating parties, a distinct secret key is required. Examples of symmetric ciphers are the Enigma machine, UNIX crypt, DES [Nat77] (data encryption standard), and IDEA.

Asymmetric ciphers. Asymmetric crypto systems are due to Diffie & Hellman [DH76]. Instead of one secret key for each pair of agents, one public/private key pair for each agent is sufficient. These key pairs are pairs of encryption and decryption keys, where $K_E \neq K_D$ and it is infeasible to compute K_D from K_E. We define the *public key* $K_E =: K_{\text{pub}}$ and *private key* $K_D =: K_{\text{pri}}$. Examples of such algorithms are RSA [RSA78] and variants of Diffie & Hellman's original algorithm, such as ElGamal [ElG85]. These algorithms are too slow to encrypt large volumes of data.

Public key cryptography depends on so-called *trap-door functions,* which can be considered to be one-way functions with a secret exit; or in other words, these functions are easy to compute in one direction, but infeasible to invert unless a secret (the secret key) is known. The key pair is usually derived from a common root, such as large prime numbers, in a way that makes it infeasible to reconstruct the root from the public key.

Block ciphers. Most algorithms encrypt fixed-size blocks of data (e.g., 64 bits), one at a time, which generally requires some padding in the last block. The latter may be considered a possible weakness for attacks, but since modern algorithms are required to resists known plaintext attacks, this is not really much of an issue. More important is that the blocks of ciphertext are independent, so that an attacker has the opportunity to spot repeating patterns and can try to infer their relationship to the plaintext. Moreover, a checksum or similar mechanism needs to be used to ensure the the integrity of a transmitted message.

These problems can be alleviated using a technique known as *cipher block chaining,* which is graphically illustrated in Figure 1. Before encryption, each block is xor-ed with the *encrypted* version of the preceding block. The same is done after decrypting a block to obtain the plaintext. As a result two occurrences of the same plaintext block in a stream of blocks will usually not lead to the same ciphertext. However, there is still a weakness at the start of each sequence of blocks.

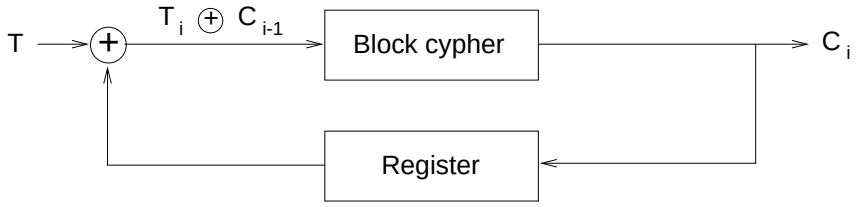

Figure 1: Cipher block chaining

It can be removed using varying *initialisation vectors* (such as a time stamp) at the start of a message.

Stream ciphers. In contrast to block ciphers, stream ciphers encode a given plaintext bit by bit, which is more appropriate for some applications (such as realtime voice transmission). These algorithms work by xor-ing a so-called *keystream,* which is a sequence of seemingly random bits, with the plaintext. The security of the resulting ciphertext is the same as that of the keystream. The keystream is produced from the output of a random number generator by encoding it with a block cipher algorithm. The receiver can reconstruct the plaintext by generating the same keystream and xor-ing it with the ciphertext. This implies the knowledge of the starting value of the random number generator and the key used to encrypt the key stream.

As long as the receiver knows how many bits have been transferred, even messages where part of the ciphertext has been lost can be decoded, as long as it is know how many bits have been lost. This is in contrast to block ciphers using cipher block chaining.

Design of cryptographic algorithms

The following two information preserving manipulations are used to conceal the content of ciphertext blocks [Sha49, Sch96]:

Confusion. The use of information preserving operations (e.g., xor and circular shifting) to combine blocks of plaintext with the encryption key. This obscures the relationship between corresponding positions in the plaintext and ciphertext.

Diffusion. Transposing portions of plaintext blocks to dissipate regular patterns arising from repetition and redundancy in the plaintext.

TEA

As a concrete example of the actual functioning of a symmetric encryption algorithm, consider the *Tiny Encryption Algorithm (TEA)* by Wheeler & Needham [WN94]. Its encryption function is the following:

```
void encrypt (unsigned long k[], unsigned long text[])
{
  unsigned long y = text[0], z = text[1];
  unsigned long delta = 0x9e3779b9, sum = 0; int n;
  for (n = 0; n < 32; n++) {
    sum += delta;
    y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);
    z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);
  }
  text[0] = y; text[1] = z;
}
```

The function encodes a single 64-bit block `text`, which consists of two 32-bit integers (here, `long` is assumed to be 32-bit), using a 128-bit key `k` represented by four 32-bit integers. Encryption is realised in 32 rounds, each of which shifts and combines the halves of `text` with the four parts of the key. Furthermore, a constant `delta` is used to obscure the key in portions of the plaintext that do not vary.

The xor operations and shifting of the text in the algorithm implement confusion. The shifting and swapping of the two halves of the text implement diffusion.

The corresponding decryption function is

```
void decrypt (unsigned long k[], unsigned long text[])
{
  unsigned long y = text[0], z = text[1];
  unsigned long delta = 0x9e3779b9, sum = delta << 5;  int n;
  for (n = 0; n < 32; n++) {
    z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
    y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
    sum -= delta;
  }
  text[0] = y; text[1] = z;
}
```

Despite its simplicity, TEA is a secure and reasonably fast encryption algorithm, which can easily be implemented in hardware, and is approximately three times as fast as DES.

DES

The Data Encryption Standard (DES) was developed by IBM for the US government in 1977. Its dataflow is illustrated in Figure 2. The algorithm encodes a 64-bit block M with a 56-bit key K in 16 rounds. In each round, the 56-key is rotated by 1–2 bits to produce the key for that round K_i. Furthermore, the encryption step function f transposes and partially duplicates 32 bits of R yielding 48 bits, xors it with the key of that round K_i and substitutes/reduces the number of bits to 32.

In hardware, DES has achieved a throughput of up to 1.2×10^9 bits/s [ET92]. This cipher is heavily used in practice and, in particular, also in banks and government institutions who keep very sensitive data. This is despite the recent success in breaking DES keys by brute-force attacks. There have been four challenges to break DES keys:

- 06/1997 (DES-I), Verser: 12 weeks
- 02/1998 (DES-II-1), `distributed.net`: 39 days
- 07/1998 (DES-II-2), Electronic Frontier Foundation (EFF): 3 days
- 1999 (DES-III), `distributed.net` together with EFF: *23h!*

As a response to the weakness of DES and because the algorithm does not allow a simple extension of the key length, a variant called *Triple DES* was introduced, which uses a $112 = 2 \times 56$ bit key to encrypt-decrypt-encrypt with standard DES. Unfortunately, this makes the already slow DES algorithm even slower and there is work showing that the effective key length of triple DES is actually shorter than 112 bits [MH81].

Other symmetric algorithms

IDEA. The International Data Encryption Algorithm uses a 128-bit key to encrypt 64-bit blocks (like TEA). It is approximately three times as fast as DES and, like DES, uses the same function for encryption and decryption.

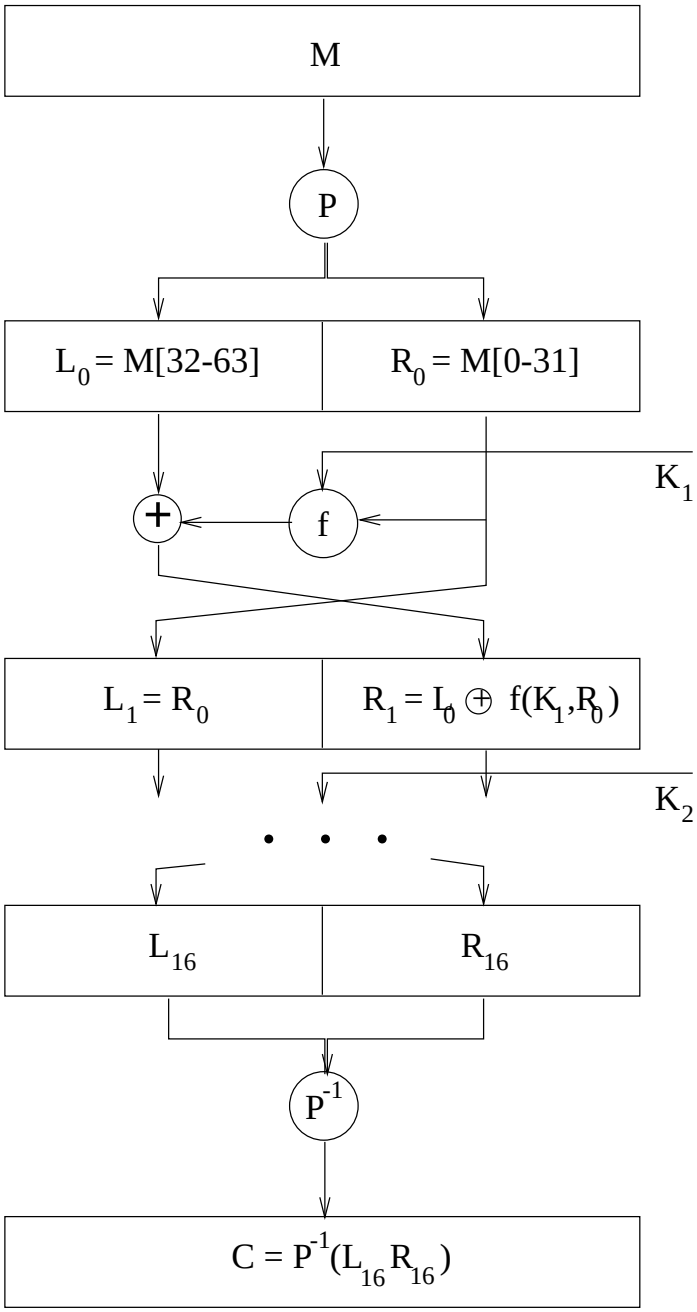

Figure 2: Schema of DES

AES. The Advanced Encryption Standard was defined in 2001, where the algorithm *Rijndael* (by Daemen & Rijmen) was selected from a set of submissions and publicly reviewed. AES was defined as a replacement to the ageing DES.

Rijndael has a variable block and key length with a specification for keys with a length of 128, 192, or 256 bits to encrypt blocks with a length of 128, 192 or 256 bits. The block and key length can be extended to multiples of 32 bit.

RSA

The RSA algorithm is an asymmetric (public key) cipher by Rivest, Shamir and Adelman [RSA78]. The key pair is determined as follows:

1. Choose two primes $P, Q > 10^{100}$
2. $N = PQ; Z = (P-1)(Q-1)$
3. Choose K_E relatively prime with Z
4. Determine K_D such that $K_E K_D = 1 \bmod Z$

Using these keys, k bits, with $2^k < N$, are encrypted using the function

$$\{T\}_{K_E} = T^{K_E} \bmod N$$

and ciphertext is decrypted using

$$\{C\}_{K_D} = C^{K_D} \bmod N$$

RSA is slow (kilobits/s even in hardware), but it is well suited to establish secure channels, which can be used to distribute the secret key needed for a faster symmetric algorithm. Moreover, as

$$\{\{T\}_{K_E}\}_{K_D} = \{\{T\}_{K_D}\}_{K_E} = T, \text{forall } T \text{ with} 0 \leq T \leq N$$

holds, RSA can be used for digital signatures.

The security of RSA depends on the computational difficulty of determining the prime factors of large numbers. However, there is no rigorous proof that excludes the existence of a cheap method for factoring. Furthermore a key of ≈ 500 bits was recently broken by brute force.

ElGamal

ElGamal is a derivative of Diffie-Hellman, which compares to RSA as follows [Sim99]:

- Message expansion: ciphertext is twice as big as plaintext in DH
- Computational intensity: DH requires more processing power
- Randomness: DH requires a good source of randomness; otherwise, the private key may be compromised
- Copyright & patents: RSA was covered by a patent until September 2000; there are copyright issues with the RSA implementations
- Key strength: RSA offers less "security-per-bit" of key material
- Theoretical basis: DH appears to be based upon more solid mathematical theory
- Reconstruction of session keys: DH sessions keys are evanescent, i.e., cannot be reconstructed even if private key of agents is obtained later

Digital signatures & digests

In addition, to hiding the contents of a message, cryptography can also be used to guarantee that a message has not been altered in transit and that it was authorised by a specified entity.

Secure digest. The simplest method to ensure that a message has not been altered is to compute a *digest* or *hash,* i.e., a fixed-length value that has been produced by condensing the information in the message. For this method to be secure, given a message M and its hash $H(M)$, it must be very hard to find a message M' that has the same digest; i.e., for which $H(M) = H(M')$.

This is particularly important since statistically it is much more likely to find a pair of two elements with a matching feature in a given set than finding a single element with a pre-specified feature. This is often illustrated using groups of people and finding two people with the same birthday compared to finding a person with a given birthday in the same group. Hence, attacks that try to exploit this statistical weakness are called *birthday attacks.*

Secure hash functions operate in a manner not unlike encryption functions. However, as they are clearly not information preserving, they can use a wider range of bit-level operations than encryption functions. The two most widely used algorithms are *MD5* and *SHA*. Ron Rivest's MD5 algorithm computes a 128-bit digest and is the more efficient of the two algorithms. SHA is a standardised algorithm that is more secure than MD5, as it produces a 160-bit digest. Using cipher block chaining, any symmetric encryption algorithm could be used as a hashing function, but this is generally less efficient than a dedicated hash function and requires the use of a key.

Digital signatures. A secure digest alone guarantees the integrity of a message if we trust into having obtained the correct digest. However, if the digest is transmitted alongside the message, anybody who modifies the message can as easily replace the digest by one matching the modified message. To prevent this and simultaneously assert the identity of the sender of the message, the digest may be encrypted by the sender. If the sender encrypts the digest by public key cryptography using a private key whose matching public key is available to the receiver, that encrypted digest serves as a digital signature.

More formally, the signed message consists of the pair $(M, \{H(M)\}_{K_{\text{pri}}})$. The recipient can use the matching public key K_{pub} to decrypt $\{H(M)\}_{K_{\text{pri}}}$ and compare the result to its own computation of $H(M)$. Given that it is infeasible to find a message M' that has the same digest and assuming that only the sender is in possession of K_{pri}, the receiver can be sure that M has indeed been signed by the correct sender and that the received message is the same as the one sent.

Secure and unforgeable data transmission. In order for digital communication to gain the same trust as is placed in a signed document that is sent in a sealed envelope, we need to combine digital signatures with encryption. In such a transaction, the communicating parties want to ensure the following:

Authentication: The receiver wants to be sure of the senders identity.

Confidentiality: The transmitted information is kept private between sender and receiver.

Integrity: The message could not have been altered in transit.

Non-repudiation: The sender cannot credibly deny having signed the message.

Let us assume the two communicating parties Alice and Bob own private and public keys $A_{\text{pri}}/A_{\text{pub}}$ and $B_{\text{pri}}/B_{\text{pub}}$, respectively. Alice signs and encrypts a message M as follows in three steps:

1. She computes a digital signature $H(M)$ using a secure hash (e.g., SHA) and encrypts it with her private key A_{pri}, yielding $\{H(M)\}_{A_{\text{pri}}}$.

2. Then, she generates a symmetric encryption key K_s, called a *session key*, which she uses to encrypt (e.g., using AES) both the message M and the digital signature, yielding $\{M\}_{K_s}$ and $\{\text{Alice}, \{H(M)\}_{A_{\text{pri}}}\}_{K_s}$.

3. Finally, she encrypts the session key with Bob's public key, yielding $\{K_s\}_{B_{\text{pub}}}$.

The resulting triple $(\{M\}_{K_s}, \{\text{Alice}, \{H(M)\}_{A_{\text{pri}}}\}_{K_s}, \{K_s\}_{B_{\text{pub}}})$ is useless to the casual observer, as all data is encrypted. Only Bob can decrypt the message and verify the identify of the sender. Bob does so as follows in four steps:

1. Using his private key B_{pri}, he recovers the session key K_s from $\{K_s\}_{B_{\text{pub}}}$.

2. The session key allows him to recover the original message from $\{M\}_{K_s}$ and Alice's identity and signature from $\{\text{Alice}, \{H(M)\}_{A_{\text{pri}}}\}_{K_s}$.

3. After looking up Alice's public key A_{pub}, Bob can decrypt Alice's signature $\{H(M)\}_{A_{\text{pri}}}$ to recover the hash value computed by Alice for M.

4. Finally, Bob computes $H(M)$ by himself and compares the result to the hash value determined in the previous step. If they are the same, Bob can trust that the message is unaltered and has indeed been signed by Alice.

Cryptographic protocols

In order to solve actual problems using the cryptographic techniques discussed in the previous parts, we develop cryptographic protocols. As with regular (e.g., communication) protocols, a cryptographic protocol defines a series of predetermined steps performed by several parties in order to achieve a particular goal. The main difference with cryptographic protocols is that they must be secure, that is, they must prevent any of the parties from 'cheating'. Such protocols make use of both symmetric and asymmetric ciphers, secure digests, signatures, and random number generators.

Several standard mechanisms used in these protocols include challenges and responses, the use of session keys, and the use of tickets. In challenge-response based protocols, one party sends a challenge that the receiving party can only respond to if it knows a shared secret. Often a *nonce* (such as a unique random number) is included in both the challenge and response to uniquely associate the messages to each other thereby preventing replay attacks. A session key is often used to encrypt communication messages. This is a short-lived shared key and avoids problems (such as susceptibility to replay attacks and weakening of the key) related to reusing the same key for all secure communication. Finally tickets consist of secured data that is received by one party and meant to be passed on to another party. Tickets typically contain data that is needed by the ultimate receiver to proceed with the protocol.

Besides these mechanisms there are also several principles that the design of cryptographic protocols should be based on.

- A protocol should avoid having two parties do things identically (for example generate the same kinds of challenges).

- A party should never give away valuable information (for example encrypt a challenge) if it does not know who the receiver is (i.e., it hasn't authenticated the receiver yet).

- A message should contain all relevant information (i.e., none of the information required for the interpretation of the message, for example the sender id, should be implied).

Further discussion of principles for cryptographic protocols can be found in [AN96].

Key Distribution

A set of keys provides a *secure channel* for communication. But how does the secure channel get established in the first place? We can either use a separate channel to establish keys or a use key distribution protocol. These protocols vary depending on whether symmetric or asymmetric encryption is used. Symmetric keys are often communicated over a channel which was established using an asymmetric cipher.

Symmetric keys (Needham-Schroeder)

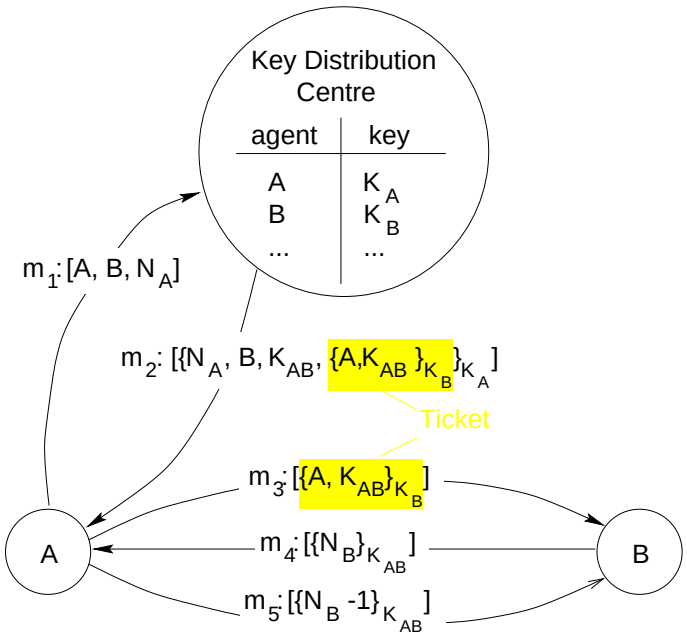

Figure 3: Needham-Schroeder protocol for symmetric ciphers

The Needham-Schroeder protocol [NS78] with secret keys is illustrated in Figure 3. It is based on a central *key distribution centre* D that shares a (symmetric) key K_A with each agent A. If agent A wants to communicate with B, it obtains a *session key* K_{AB} as follows:

1. A requests session key from D,
2. D replies with key K_{AB} and *ticket* $\{A, K_{AB}\}_{K_B}$,
3. A sends ticket to B,
4. B replies with *challenge (nonce)* $\{N_B\}_{K_{AB}}$,
5. A answers challenge

Now both A and B know that they share a key provided by D.

In the protocol, the ticket and challenge implicitly *authenticate* A and B. The nonce and challenge protect against replay attacks. A disadvantage is the use of a centralised resource D. It may be distributed using a hierarchical scheme, but every agent must trust D and as D maintains highly sensitive information (secret keys), if D is compromised all communication is compromised. Moreover, a large number of keys are required (one per pair of agents), which have to be manufactured by D on-the-fly. D must take care to make key sequences non-predictable.

Public keys

A major weakness of the Needham-Schroeder protocol and Kerberos is the existence of the trusted key distribution centre and, in particular, that it contains all secret keys of all parties. If the key distribution centre is compromised, not only future communication is threatened, but if recordings of past encrypted communication exists, it may be decrypted retroactively. This is one of the reasons that makes a *public key infrastructure (PKI)* more attractive. Public keys can be exposed without risk. The only value of a distribution centre of public keys is in guaranteeing that a particular public key belongs to a specific entity; in other words, the distribution centre serves to establish a link between identities and public keys.

Certificates and certification authorities. We generally link an identity to a public key by means of a *certificate.* Hence, distribution centres for public keys, which are also called *certificate servers* or *certificate directories*, simply distribute entire certificates. To avoid the need for secure channels between the certificates servers and their clients, digital signatures are used to assert the validity of certificates. The certificates are usually formatted according to the X509.1 standard format or the PGP format.

However, this raises the question of whose signature on a certificate the receiver of that certificate is prepared to trust. The classic answer is to establish *certification authorities,* which are trusted entities that sell certification as a service.

In any case, the entity that signed a certificate may not be immediately trusted. Instead, to establish trust in a certificate C_1 signed by Alice, Bob may need to obtain Alice's certificate C_2, which has been signed by an entity that has Bob's trust. Bob can then use the public key in C_2 to validate Alice's signature on C_1. Obviously, this *chain of certificates* can be even longer and establishing trust in a given certificate is generally a recursive process. It terminates once the self-signed certificate of a *root certification authority* has been reached.

Expired and revoked certificates. To minimise the risk of being compromised, certificates can be made valid for only a pre-determined time. After that, the certificate expires and a new certificate must be obtained. This requires getting a new signature that validates the certificate. Alternatively, certificates may be revoked if they should not be used anymore. However, in contrast to the expiry of a certificate, a client will know about a certificate being revoked only by regularly contacting the relevant certificate server.

Authentication

Authentication involves verifying the claimed identity of an entity (or *principal*). Authentication requires a representation of identity (i.e., some way to represent a principal's identity, such as, a user name, a bank account, etc.) and some way to verify that identity (e.g., a password, a passport, a PIN, etc.). Depending on the system's requirements different strengths of authentication may be required. For example, in some cases it is enough to simply present a user id, while in other cases a certificate signed by a trusted authority may be required to prove a principal's identity. A comprehensive logic of authentication has been developed by Lampson et al in [LABW92].

A verified identity is represented by a *credential*. A certificate signed by a trusted authority stating that the bearer of the certificate has been successfully authenticated is an example of a credential. A credential has the property that it *speaks for* a principal. In some case it is necessary for more than one principal to authorise an action. In that case multiple credentials could be combined to speak for those principals. A credential can also be made to represent a role (e.g., a system administrator) rather than an individual. Roles can be assigned to specific principals as needed.

A *delegation credential* allows a principal to perform an action with the authority of another principal. Delegation credentials are needed whenever a service needs to access a protected resource in order to complete an action on behalf of its client. When rights are delegated it is common to restrict them to a subset of the original rights so that receiver cannot abuse them.

Verifying a principal's identity in a distributed system requires sending some form of proof of identity over the network. This proof is usually in the form of a response to a challenge (e.g., a password, or a nonce encrypted with the principal's private key). The communication of this proof of identity should be protected against interception. For example, when logging in to a remote computer a password that is sent over the network as cleartext can easily be copied, giving a potential attacker access to the user's account. There are four approaches to secure authentication.

The first approach is the *shared secret key* approach. It requires that the principal and authenticator share a secret key with which the challenge and response are encrypted. The major drawback of this approach is that a different key must be stored for every pair of communicating entities. The second approach improves on this problem by storing all keys at a key distribution centre (KDC). The KDC stores a copy of each entity's secret key, and can, therefore, communicate securely with each entity. A drawback of the KDC approach is that it requires a centralised and trusted service (the KDC).

The third approach makes use of public keys to securely authenticate a principal. By sending a message encrypted with its private key a principal can prove its identity to the authenticator. A problem with this approach is that the authenticator must have the principal's public key (and trust that it does indeed belong to that principal). A different approach combines the public key and shared secret key approach. In this approach (which is used by the secure shell (ssh) protocol[1]) two parties first establish a secure channel by exchanging a session key encrypted using their public keys, and then exchange their authentication information over this secure channel.

Kerberos

Kerberos is a commercial authentication system developed at MIT [SNS88]. It is based on Needham-Schroeder protocol and integrates symmetric key encryption, distribution and authentication into commercial computer systems. It requires a secure central server, but allows for an insecure network (so, never transmits cleartext passwords) and insecure workstations (workstations are shared between users so user passwords are held on workstations only for very short periods). Moreover, it does not hold system keys on workstations.

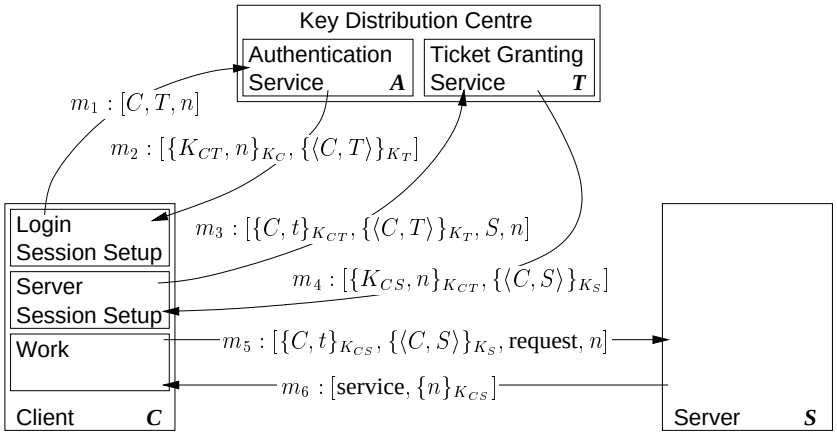

Figure 4: Kerberos protocol

The central server, called the *key distribution centre (KDC)* contains

- an *authentication service A,* which knows all user logins and their passwords (secret keys) as well as identity and key of T, and

[1]See `http://www.ietf.org/html.charters/OLD/secsh-charter.html` for more information about the ssh protocol.

- a *ticket granting service* T, which knows all servers and their secret keys.

The Kerberos protocol has three phases:

1. Login session setup (user authentication)
2. Server session setup (establishing secure channel to server)
3. Client-server RPC

The whole protocol is illustrated in Figure 4.

Kerberos user authentication. At login, the local *login session setup* component proceeds as follows:

1. prompts user for login, C, and password, K_C;
2. sends C, nonce n, and request for server ticket to A;
3. A replies with certified session key K_{CT} for communication with T and *ticket* $\langle C, T\rangle = [C, T, t_1, t_2, K_{CT}]$, valid between t_1 and t_2
4. user's password is used to decrypt certificate$\{K_{CT}, n\}_{K_C}$, this authenticates the user.
5. user's password is erased.

The ticket can now be used to obtain server keys from T. When the ticket times out, users must re-authenticate. The nonce protects against replay attacks pretending to be from A and security is achieved via a *shared secret,* namely user's password.

Kerberos server session setup. When a user wants to communicate with server S, the server session setup component proceeds as follows:

1. sends to T
   - *authenticator* $\{C, t\}$ encrypted with joint key K_{CT}
   - ticket $\langle C, T\rangle$,
   - server name S,
   - new *nonce* n
2. T authenticates the ticket, recovering joint key K_{CT}, and validates the authenticator (and thus client C)
3. replies with key certificate $\{K_{CS}, n\}_{K_{CT}}$ and server ticket $\{\langle C, S\rangle\}_{K_S}$,
4. C verifies key certificate and now possesses key K_{CS} and ticket $\{\langle C, S\rangle\}_{K_S}$ to communicate with server.

Servers only use a session key during the interval specified in the ticket. The time stamp in the authenticator protects against replay attacks pretending to be C and the nonce protects against replay attacks pretending to be from T. A ticket is only needed to open a connection, but the server keeps a timeout value with the key.

	Objects			
Subjects	O_1	O_2	O_3	O_4
S_1	terminate	wait, signal, send	read	
S_2	wait, signal, terminate			read, execute write, control
S_3		wait, signal, receive		
S_4	control		execute	write

Figure 5: Example of an access control matrix

Kerberos client-server RPC. When a user C requests a service from S, the following protocol is used:

1. C sends to S
   - *authenticator* $\{C, t\}$ encrypted with joint key K_{CS}
   - ticket $\langle C, S \rangle$,
   - request,
   - new *nonce* n
2. S authenticates the ticket, recovers the joint key K_{CS}, and validates the authenticator (and thus client C)
3. replies with the result and the encrypted nonce

Finally, C verifies the nonce.

Protection (and Authorisation and Access Control)

Once a principal has been authenticated it is necessary to determine what actions that principal is allowed to perform and to enforce any restrictions placed on it. Restricting actions and enforcing those restrictions allows resources to be protected against abuse. When discussing protection of resources we often refer to *access rights*, which are the rights required to access a given resource. Protection of resources requires, on the one hand, the ability to verify and enforce access rights and, on the other hand, the ability to grant and revoke access rights. The former will be referred to as *access control*, while the latter will be *authorisation*. Note that the three terms, protection, access control, and authorisation are often used interchangeably to refer to the same thing (i.e., protection). The conceptual basis for protection was original set out by Lampson in [Lam71].

Underlying the discussion of protection lies the concept of *protection domains*. A protection domain is an abstraction that represents an execution environment for processes and defines the access rights that processes executing in that environment have. These access rights are defined as object-rights pairs (where an object is a resource in the system). Generally a protection domain is associated with a single principal, that is, it contains all the access rights that that principal possesses. Any process started by that principal will then execute in that protection domain and have the access rights defined therein. It is also common to define protection domains for groups and roles. In this case all processes that belong to a particular group or that have a particular role will be executed in the associated protection domains.

The standard approach to determining the access permissions of a given authenticated subject to a given object is an *access control matrix*, an example of which is displayed in Figure 5. The rows of the matrix define a subject's protection domain and the columns an object's *accessibility*. The access control matrix is a dynamic data structure in that it frequently changes. The change can be permanent, such as that caused by Unix's `chmod` command, or temporary, such as that

caused by Unix's setuid bit on executable files. Because the matrix is very sparse and has many repeated entries, it is usually not stored explicitly.

Before we discuss the two alternatives for implementing the access matrix, let us briefly review some design considerations in a protection system. Later we will evaluate the implementations based on these design considerations. We consider the following points:

Propagation of rights: Can someone act as an agent's proxy? That is, can one subject's access rights be delegated to another subject?

Restriction of rights: Can a subject propagate a subset of their rights (as opposed to all of their rights)?

Amplification of rights: Can an unprivileged subject perform some privileged operations (i.e., (temporarily) extend their protection domain)?

Revocation of rights: Can a right, once granted, be removed from a subject?

Determination of object accessibility: Who has which access rights on a particular object?

Determination of a subject's protection domain: What is the set of objects that a particular subject can access?

Implementation of the Access Matrix

An efficient representation of the access matrix can be achieved by representing it either by column or row. A column-wise representation that associates access rights with objects is called an *access control list (ACL).* Row-wise representations that associate access rights with subjects are based on *capabilities.*

Access Control Lists

Each object may be associated with an access control list (ACL), which corresponds to one column of the access matrix, and is represented as a list of subject-rights pairs. When a subject tries to access an object, the set of rights associated with that subject is used to determine whether access should be granted. This comparison of the accessing subject's identity with the subjects mentioned in the ACL requires prior authentication of the subject.

ACL-based systems usually support a concept of *group rights* (granted to each agent belonging to the group) or *domain classes.* When a large number of objects provide the same rights to a set of subjects, these subjects may be combined into a group, thus reducing the size of the ACL. To reduce the number of groups needed, a system may have the ability to denote negative rights, so that exceptions can be easily expressed. In practice, to keep the ACL small, existing systems simplify ACLs to a fixed-size list (e.g., UNIX's *user-group-others* or VMS' *system-owner-group-world*).

The properties of ACLs, with respect to the previously listed design considerations, are as follows:

- Propagation: the owner of an object can add to or modify entries to the ACL (e.g., owner can `chmod`),
- Restriction: anyone who has the right to modify the ACL can restrict access,
- Amplification: ACL entries can include protected-invocation rights (e.g., setuid),
- Revocation: access rights can be revoked by removing or modifying ACL entries,
- Object accessibility: the ACL itself represents an object's accessibility,
- Protection domain: hard (if not impossible) because the ACLs of all objects in the system must be inspected.

Capabilities

As an alternative to ACLs, subjects may be associated with a list of the access rights that they possess. In this case, a list of a subject's access rights is called a *capabilities list (clist),* which also defines a protection domain. Each capability in a clist can confer a single or a set of rights on an object. Some of these rights may be negative rights. A capability implicitly refers to a single object and can, therefore, be used as an object reference. Because capabilities also implicitly specify access rights for an object, they can be thought of as keys[2] that give access to objects. The possession of a capability can be regarded as sufficient evidence of access permission to the associated object.

The main technical problem in a capability-based system is protecting capabilities against forgery and theft. In return, because a capability alone is sufficient to prove the right for object access there is no need for explicit authentication of subjects upon object access. As a result, the system does not need to trust any intermediary, such as a remote OS kernel.

The properties of capabilities, with respect to the previously listed design considerations, are as follows:

- Propagation: is achieved by simply copying a capability,
- Restriction: may be supported by means of *derived* capabilities,
- Amplification: requires special amplification capabilities,
- Revocation: can be difficult and depends on exactly how capabilities are represented,
- Object accessibility: is hard (if not impossible) because it requires the inspection of every subject's capabilities,
- Protection domain: a clist represents a protection domain.

Implementation of Capability-based Systems

Capabilities are especially attractive for use in distributed systems because no explicit authentication is required to used them. We will, therefore, examine some implementation aspects of capabilities and look at example systems in more detail.

Rights Amplification and Restriction with Capabilities

The amplification of rights is sufficiently easy to realise in server-based systems, where client and server communicate via RPC. In this case, the server performs privileged operations on behalf of client. Rights restriction, on the other hand, is more difficult to realise. In general, a capability system can provide a special mode indicating special privileges. The AS/400 provides *profile adoption*, where the invocation of an object merges the caller's protection domain (PD) with (subset of) object's owner's PD. The caller is allowed to restrict the PD passed to the object (this is called *profile propagation*). In Mungi, *protection domain extension (PDX)* is a controlled temporary extension of caller's PD by a PD registered for the called object. The caller can designate a specific (restricted) PD to be passed to the called object.

Making Capabilities Tamper-proof

There are three basic approaches to making capabilities tamper-proof:

Tagged capabilities: Protection of capabilities is based on special hardware, namely a tag bit, which is controlled by the operating system, such that only the kernel can turn the tag bit on. This is the classical method.

[2] comparable to house keys

Partitioned (segregated) capabilities: All capabilities are held in kernel space and only references are passed to user processes. This approach has been used in Mach, Grasshopper, and EROS.

Sparse capabilities: Capabilities are protected by using a sparse representation that cannot be easily forged. This method is used in the Monash Password Capability System, Amoeba, and Mungi.

We look at all three schemes in more detail.

Tagged capabilities. Tagged capabilities typically consist of (1) a pointer (address), (2) a set of access rights, and (3) a tag bit indicating a capability. Any write operation to a capability turns the tag bit off and only the kernel can turn it on again. Tagged capability can be propagated by copying, but revocation is not easy and restriction requires kernel intervention. Moreover, they cannot be used in a distributed system without a separate mechanism to transfer capabilities securely over the network. The IBM System/38 aka AS/400 (which is the most successful capability system ever) features special capabilities to allow rights amplification ("profile adoption") and allows the caller to designate a subset of a protection domain to be passed on ("profile propagation").

Partitioned capabilities. Partitioned capabilities are stored in a system-protected area (e.g., in the kernel) as clists. User code references capabilities only indirectly (using an index to the clist). Both propagation and restriction require system intervention. For example, in a system where resources are accessed using IPC (interprocess communication) the system can convert clist indices into capabilities on executing an IPC. Rights amplification can be implemented via servers as outlined earlier. Revocation is easily implemented as the kernel keeps track of all the capabilities. Moreover, the scheme allows the use of reference counting to determine when objects become inaccessible. Again, the use in a distributed system requires an additional mechanism to securely transfer capabilities over the network.

Sparse capabilities. Sparse capabilities consist of (1) an object ID, (2) an optional set of access rights, and (3) a "random" bit string for sparsity. Sparse capabilities are simply data that is secured by statistical arguments (i.e., it is infeasible that someone can forge or modify a valid capability), much like secure hashes. Users can freely copy sparse capabilities, which makes propagation easy. However, care must be taken to protect capabilities from theft and there are no means for the system track to who has which capabilities. This makes accessibility impossible to establish. The feasibility of rights restriction and amplification depends on the concrete implementation. One of the most attractive features of sparse capabilities is that they can be easily shared over a network, which makes them well suited for use in a distributed system.

Examples of Sparse Capability Systems

Signature capabilities (first migration scheme). This scheme was originally designed to allow distribution of tagged capabilities. It is depicted in Figure 6. In this scheme, capabilities are made tamper proof via encryption with a *secret kernel key* and can be passed around freely. The two main disadvantages of this scheme are that capabilities have to be decrypted whenever they are validated (which is costly) and that users do not know which object the capability refers to.

Signature capabilities (second migration scheme). This variant addresses the second disadvantage of the first migration scheme and is depicted in Figure 7. It makes the object ID visible, but remains tamper-proof.

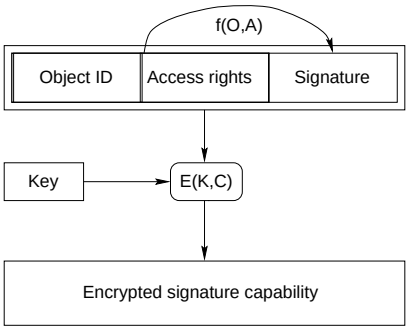

Figure 6: First migration scheme

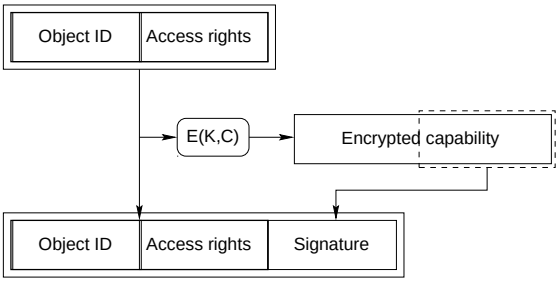

Figure 7: Second migration scheme

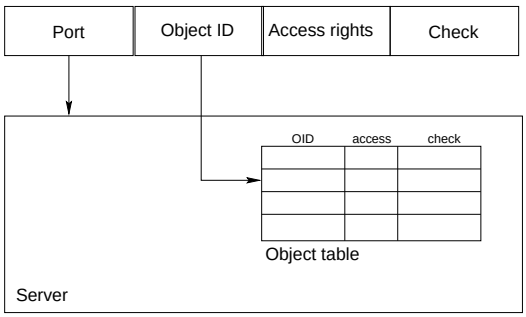

Figure 8: Amoeba's capabilities

Amoeba's capabilities. Amoeba is a server-based distributed OS that uses sparse capabilities as depicted in Figure 8. A *port* in Amoeba identifies a server. Port IDs are large (48-bit) sparse numbers; hence, knowing a port implies send rights. The server uses the OID to look up access rights and compares the check fields to validate the capability.

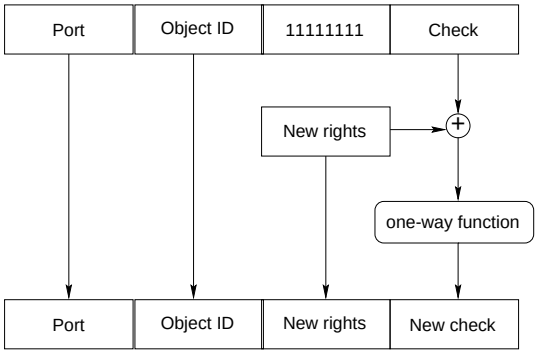

Figure 9: Rights restriction in Amoeba

An original ("owner") capability has all rights granted to it. Rights restriction is achieved by asking the server to derive a lesser capability. The mechanism for rights restriction is illustrated in Figure 9. The check field of the new capability is calculated by XORing the new access rights bitmap with the original check field, and then passing the new value through a one-way function. In order to verify a restricted capability the server must perform the same actions (XOR the access rights with the check field and then pass these through a one-way function) and compare this value with that of the capability. A match implies that the access rights, as specified in the capability, have not been modified and are therefore valid. A capability can be revoked by asking the server to change the check field. This results in access being revoked for all holders of the capability. Rights amplification is provided by the server.

An alternative to the rights restriction scheme displayed in Figure 9 is to use a set of *commuting* one-way functions f_i, one for each access mode, such that $f_i(f_j(x))) = f_j(f_i(x))$. To remove access mode i, a process obtains a new check field $C' = f_i(C)$. This can be done by a user without server intervention, and the server can verify a capability based on the rights field in the capability. This scheme has not been implemented.

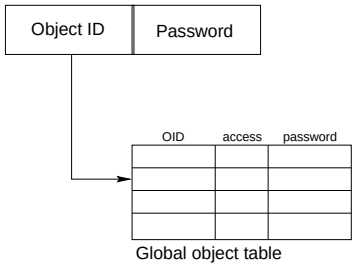

Figure 10: Password capabilities

Password capabilities. This flavour of capabilities was invented for the Monash *password capability system* and is depicted in Figure 10. A random bitstring (not derived from other parts of the capability) is used as a password. Validation requires checking against a global *object table.* As no encryption of parts of the information in a capability is used, validation is fast. However, the password must be hard to guess, which means cryptographic methods are needed for its generation.

Separate passwords may convey different rights, but it is generally useful to include the rights information in the capability to support restriction. An OID with a null password may be used as a

valid capability conferring no rights. Password capabilities are location independent and selective revocation is possible if multiple passwords are used. Rights amplification may be realised via special amplification capabilities (as in Mungi's PDX calls).

Limitations of Access Matrix Model

Lock-key models, which combine ACLs and capabilities, try to overcome some of the disadvantages of the separate models. In these models an object has a *lock* (ACL) and a subject has a *key* (capability). Access is only permitted if both match.

Note that the use of an access matrix does not provide complete protection. In particular it cannot prevent an authorised subject from accessing a resource on behalf of another unauthorised subject. This is called the *confinement problem.* Information may can be leaked via storage channels (e.g. via files), legitimate channels (e.g. during normal communication), or covert channels (e.g. by modulating resource usage), which is very difficult to prevent. These problems can only be tackled by strictly limiting information flow.

We generally distinguish between access-matrix based models that provide *discretionary security*, where the user has control (discretion) over who may access their data and security depends on users being trustworthy and sensible, and *mandatory access control* where security policies are enforced system-wide.

Another problem with the access control matrix model is that it assumes that the access control and authorisation mechanisms cannot be circumvented. Unfortunately, most software contains bugs, and many bugs can be exploited to circumvent protection mechanisms. It is therefore important to make sure bugs are fixed as soon as possible to prevent exploit based attacks.

Firewalls

A different form of protection that can be employed in distributed systems is that offered by *firewalls*. A firewall is generally used when communicating with external untrusted clients and servers, and serves to disconnect parts of the system from the outside world, allowing inbound (and possibly outbound) communication only on predefined ports. Besides simply blocking communication, firewalls can also inspect incoming (or outgoing) communication and filter only suspicious messages. Two main types of firewalls are *packet-filtering* and *application-level* firewalls. Packet-filtering firewalls work at the packet level, filtering network packets based on the contents of headers. Application-level firewalls, on the other hand, filter messages based on their contents. They are capable of spotting and filtering malicious content arriving over otherwise innocuous communication channels (e.g., virus filtering email gateways).

Note that firewalls on their own are not enough to protect a system. The three myths of firewalls[3] summarise the main problems with relying only on firewalls for security. The first myth, *we've got the place surrounded*, refers to the fact that relying on firewalls assumes that the only way into our out of the system is through the firewall. Too often this is not the case, especially if users or processes bore through or avoid the firewall by installing private modems, etc. The second myth, *nobody here but us chickens*, refers to the fact that firewalls only stop external attackers. Many attacks are performed by entities within the confines of a firewall. Finally, the last myth, *sticks and stones may break my bones, but words will never hurt me* refers to the fact that external content that does come through the firewall may still pose a security threat. Examples of this include macro viruses, javascript exploits, and trojan horse programs.

Secure Communication

All the security mechanisms mentioned above (including cryptography) can be combined to provide secure communication via secure channels. A secure channel is a communication channel that is set up between two (or more) entities and provides confidentiality and integrity of communication.

[3] `http://www.mit.edu/afs/athena/astaff/project/kerberos/www/firewalls.html`

It requires authentication and authorisation to ensure that the communicating parties are who they think they are and that they are in fact authorised to communicate together. Cryptography is used to implement both confidentiality and integrity.

A widely used example of a secure channel is provided by the SSL (secure socket layer) protocol[4]. SSL is a flexible application level protocol to establish and maintain a secure channel. The flexibility of SSL allows the communicating processes to negotiate and choose the cipher they wish to use. SSL is widely used to secure HTTP communication in the world wide web and is recognised by the `https` URLs.

As a special case of secure communication is secure group communication. Secure group communication is of special interest in distributed systems as it allows to secure communication between more than just two parties. We discuss two forms of secure group communication: Confidential group communication and secure replicated servers.

The problem of *confidential group communication* is to protect a group of N parties against evesdropping their communication. A simple scheme to ensure confidentiality would be to use a shared secret key and use symmetric encryption to protect messages between group members. This scheme is quite vulnerable to attacks, because the secret key must be shared by all group members, which, in turn, must all be trusted. Alternatively, a separate secret key could be used for each pair of communicating group members. Such a system would be less vulnerable to an attack, but it makes it necessary to maintain $N(N-1)/2$ keys, which is already a difficult problem. A better scheme would be to use public-key cryptography, where each group member has a public and private key. The public key is used by all other members to encrypt communication to that group member.

In a system with *secure replicated servers*, a client sends a request to a group of replicated servers and needs the response from the servers to be trustworthy. Even if the servers are under attack, the client wants to be sure that the returned response has not been subject to a security attack. A simple scheme against such attacks is that the client authenticates each response from all servers. If a majority of the responses are non corrupted, the client trusts the response to be non corrupted as well. This scheme is not ideal, though, because the replicated servers become visible to the client, which violates the replication transparency. Another scheme is to use a shared secret, where all the servers share a part of a secret. Only when all get together can the secret be revealed. With secure replicated servers under attack, it will be difficult (if not impossible) to get a non corrupted response from all servers. It is therefore even better to use a *(m,n)-threshold scheme*. Such a scheme divides the shared secret into n so called *shadows*, but allows the secret to be reconstructed from from any m shadows. Hence, a client can trust a response from a number of servers to be non corrupted, if any combination of m responses reveals the shared secret.

Auditing

Auditing, the last of the four security mechanisms, involves tracing which entities access what. The main approach to auditing is to log all attempts to access the system and its resources. The integrity of these logs must be ensured. In particular it is important that the logs cannot be modified (that is existing entries cannot be removed or modified). This can be achieved by cryptographically ensuring that valid logs can only be created by appending data, and that any attempt to modify existing log contents will invalidate the log. Another approach is to store logs on media that only allows appending (and not overwriting) of data. Furthermore it is important to maintain system integrity and spot (and repair) any unauthorised modifications as soon as they occur.

[4]See `http://wp.netscape.com/eng/ssl3/ssl-toc.html` for more information about the SSL protocol. See also `http://www.ietf.org/rfc/rfc2246.txt` for more information about TLS (Transport Layer Security) which is a standardised secure communication protocol based on SSL.

Untrusted Code

Often systems require the execution of untrusted code, that is, code that originates from a third party and has not been, or cannot be, verified for its security. Examples of this include Java applets, third party applications, bug fixes, drivers, etc. Please read [SMH00] for an overview of approaches to providing protection against possibly malicious untrusted code.

References

[AN96] Martin Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng.*, 22(1):6–15, 1996.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *ieeetit*, 22:644–654, 1976.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4), 1985.

[ET92] Hans Eberle and Charles P. Thacker. A 1 Gbit/second GaAs DES chip. In *IEEE Custom Integrated Circuits Conference*, page 19.7.1. IEEE, 1992.

[LABW92] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4), November 1992.

[Lam71] Butler Lampson. Protection. In *Proceedings 5th Princeton Conference on Information Sciences and Systems*, page 437, Princeton, 1971.

[MH81] R. C. Merkle and M. E. Hellman. On the security of multiple encryption. *Communications of the ACM*, (7), July 1981.

[Nat77] National Bureau of Standards. *Data Encryption Standard (DES)*. Number 46 in Federal Information Processing Standards. US NBS, Washington DC, 1977.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:993–999, 1978.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley, second edition, 1996.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[Sim99] S. Simpson. PGP DH vs. RSA FAQ. `http://www.scramdisk.clara.net/pgpfaq.html`, 1999.

[SMH00] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary.*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101, Saarbrücken, Germany, August 2000. Springer Verlag.

[SNS88] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the 1988 Winter USENIX Technical Conference*, 1988.

[WN94] D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. Technical Report 355, Computer Laboratory, University of Cambridge, 1994.