Lectures on distributed systems

Cryptographic communication and authentication

Paul Krzyzanowski

Introduction to cryptography

We will refer to a message that is readable, or not encrypted, as **plaintext**, **cleartext** and denote it with the symbol M. The process of disguising a message to hide its substance is called **encryption**. We will represent this operation as E(M). The encrypted message, C=E(M) is called **ciphertext**. The process of turning ciphertext back into plaintext, M=D(C), is called **decryption**. **Cryptography** is the art and science of keeping messages secure. In addition to providing confidentiality, cryptography is also used for:

authentication: receiver can determine the origin of the message and an intruder cannot masquerade.

integrity: receiver should be able to verify that the message has not been modified in transit. An intruder cannot substitute a false message for the original.

nonrepudiation: a sender should not be able to falsely deny that he sent a message.

confidentiality: a message may be encrypted so that others cannot read its contents.

A **cryptographic algorithm**, or **cipher**, is the mathematical function used for encryption/decryption. If the security of an algorithm is based on keeping it secret, it is a **restricted cipher**. Restricted ciphers are historically interesting but not adequate today. With a changing user community, all's lost if thee wrong party discovers the cipher. Moreover, there is no ability to have quality control on the algorithm since it must be kept hidden. Far more preferable are ciphers that rely on a publicly-known algorithm that accepts a secret parameter, or **key**, for encryption and decryption. If the encryption and decryption keys are the same (or mathematically derivable from each other), the algorithm is known as a **symmetric algorithm** (DES is an example):

$$C = E_K(M)$$
$$M = D_K(C)$$

If the key used for encryption is different from the key used for decryption, then the algorithm is a **public-key algorithm** (RSA is an example). The decryption key cannot be

calculated from the encryption key in a reasonable amount of time (and vice versa). The reason it is called a public-key algorithm is because the encryption key can be made public. A stranger can thus encrypt a message with this public key but only the holder of the decryption key (private key) can decrypt the message. A message can also be encrypted with the private key and decrypted with the public key. This is used as a basis for digital signatures. Anyone can decrypt the message with the public key but by doing so, they know that only the possessor of the private key was able to encrypt it (and hence created the message).

One type of function that is central to public key cryptography is the **one-way function**. This is a function where it is relatively easy to compute $f(x)$ but extremely difficult to compute x, given $f(x)$ (that is, the inverse function $f^{-1}(x)$). By extremely difficult, we mean a complexity that would take millions of years if all the computers in the world were assigned to the problem. One often-cited way of thinking about a one-way function is to think of breaking a glass, or a plate. It's infinitely easier to break it than it is to put it back together.

So what good are these one-way functions? We can't use them for encryption (nobody would be able to decrypt the message). One particular form of a one-way function is the **one-way hash function**. This is also known as a message digest, fingerprint, cryptographic checksum, integrity check, manipulation detection code (MDC). The function takes variable-length input (the message) and computes a generally smaller, but fixed length, output which is the hash value. This value indicates whether the pre-image (the original message) is likely to be the same as the real message. While it is easy to compute the hash from a pre-image, it is (nearly) impossible to generate a pre-image that results in a given hash. The hash itself is a public function. No secrecy is needed. Its one-wayness is its security. One way hashes are used for fingerprinting files. A variant of the one-way hash is the **encrypted hash**, known as a Message Authentication Code (MAC) or a Data Authentication Code (DAC). This is the same as the hash but is also a function of a secret key so that only the possessor of the key can verify the integrity of the message. This means that if the message is intercepted and altered, the encrypted hash cannot be computed by the perpetrator who does not possess the key.

Classical cryptosystems

The goal of cryptography has been to render messages unintelligible and their use certainly predates digital computer systems. The earliest documented use of ciphers was in the Roman army under Julius Caesar around 60 B.C.. This was a simple **substitution cipher** where each letter in plaintext is replaced by some other letter. In this case, each letter was replaced by one n positions away from it, modulo the alphabet size. Even this simple cipher, known as a *Caesar Cipher*, still lives on in the domain of netnews, where it is known as ROT13. Each letter is shifted by 13 positions (A becomes N, B becomes O, etc.). Needless to say, decryption is trivial and its only use is to avoid the case of someone reading a message inadvertently and getting offended. The general case of a substitution cipher is to maintain a secret substitution alphabet – a random scrambling of the alphabet that will be used as a lookup table for performing the substitution. Both sides would need to have a copy of this alphabet. The problem with substitution ciphers is that they are vulnerable to frequency analysis. Each language has a characteristic distribution of

letters (for example, if we look at Shakespeare's English, we'll see that 'e' occurs 11.8% of the time, 'o' occurs 8.3% of the time, and 'x' occurs 0.14% of the time). By looking at the frequency of letters in the ciphertext, it is generally easy enough to decipher most of the substitution alphabet.

To thwart a frequency analysis attack, **polyalphabetic ciphers** were developed. Here, different ciphertext symbols can represent the same plaintext symbol. The earliest of these ciphers was created by Leon Battista Alberti and consisted of two concentric disks, one smaller than the other. The inner disk as the alphabet along its circumference. The outer disk has a substitution alphabet along its circumference. To start encryption, the disks are aligned with a predetermined line-up of an inner letter to an outer letter. Then, the ciphertext is generated by finding the corresponding cyphertext character on the outer disk that lines up with the plaintext character on the inner disk. After *n* symbols, the disk is rotated to a new alignment (say, shifted by one position).

Another basic form of cryptography is the *transposition cipher*. This involves permuting the letters in the plaintext message according to some set of rules. Knowledge of the rules will allow the message to be decrypted. Here is a slightly sophisticated example.

Suppose we have a message *If she weighs the same as a duck, she's made of wood* and a key "31415927", we can arrange the plaintext message under the key, wrapping around as needed:

```
31415927
IFSHEWEI
GHSTHESA

MEASADUC

KSHESMAD

OFWOODXX
```

To transmit the coded message, we read out the text column-first, sorting the columns by the elements in the key (numbers in this case), obtaining:

```
FHESFHTSEOESUAXIGMKOSSAHWEHASOIACDXWEDMD
```

A recipient would simply arrange these in column-first order to fill eight columns and then move each column into its unsorted position. Both the number of columns and the positions are functions of the secret key. Transposition ciphers may be combined with substitution ciphers to yield even stronger algorithms (for example, the German ADFGVX cipher used in World War I). The problem with using a good transposition cipher is that these ciphers generally require a lot of memory and may require that messages be of certain lengths. If a cipher requires that a message be a multiple of a certain size, it is known as a **block cipher** and encryption is performed a block at a time. If the message is performed character by character and there is no requirement that a message be a specific size, the cipher is a **stream cipher**.

As mechanical techniques improved and better encryption was demanded, a class of cryptographic engines known as *rotor machines* emerged (around 1917). A rotor machine contains a set of independently rotating cylinders through which electrical pulses flow. Each cylinder has an input and an output pin for each letter of the alphabet (e.g. 26 input pins and 26 output pins). The cylinder also has internal wiring that connects each input pin to a unique output pin. The simplest machine would contain a single cylinder. A letter is associated with each input and output pin. For example, an operator may depress a key for 'P' that may be wired to the 13^{th} input pin and the 9^{th} output pin, producing 'L'. After each key is depressed, the cylinder rotates one position, so that all the internal connections are shifted by one. After 26 characters have been entered, the cylinder is back in its original position.

A single-cylinder rotor machine yields a polyalphabetic substitution cipher with a period of 26, which is not a formidable challenge to a cryptoanalyst. The machine is improved by adding multiple cylinders, such that the outputs of one cylinder feed the inputs of another. The cylinders operate similar to an odometer. With each keystroke, the one farthest from the input pin rotates one position. For every complete rotation, the one next to it rotates one position, and so on. With three 26-character cylinders, there are 26^3=17,576 different substitution alphabets before the system repeats. With 5 cylinders, there are 26^5=11,881,376 possible substitution alphabets.

Communication

We can engage in secure communication using symmetric cryptography, public key cryptography, or a hybrid system.

Communication with symmetric cryptography

To communicate using symmetric cryptography, both parties have to agree on a secret key. After that, each message is encrypted with that key, transmitted, and decrypted with the same key.

Key distribution must be secret. If it is compromised, messages can be decrypted and users can be impersonated. However, if a separate key is used for each pair of users, the total number of keys increases rapidly as the number of users increases. With n users, we would need $[n(n\text{-}1)]/2$ keys. Secure key distribution is the biggest problem in using symmetric cryptography.

Communication with public key cryptography

Public key cryptography, by using a different key for decrypting than encrypting solves problems of key distribution. If Alice and Bob wish to communicate, Alice sends Bob her public key and Bob gives his public key to Alice. Alice then encrypts her message to Bob with Bob's public key, knowing that only Bob, the possessor of Bob's private key, can decrypt the message. Likewise, Bob encrypts his messages to Alice with Alice's public key. Public keys may be stored in a database or some well-known repository so that the keys do not have to be transmitted. Not only does public key cryptography solve key

distribution, it also solves the problem of having $[n(n\text{-}1)]/2$ keys for n users. Now we only need $2n$ keys (n public and n private).

Communication with hybrid cryptosystems

Wonderful as public key cryptography may be, a problem with public key algorithms is that they are currently considerably slower than symmetric algorithms (at least 100 times slower in software and 1000 times slower in hardware). Public key algorithms can be vulnerable if the message is one of several known plaintext messages. An analyst needs to only encrypt (with the readily available public key) each of the possible messages and compare the result. She won't discover the key but will know the message. Because we would like to use a different key for each communication session (session key) , we would have to generate one on the fly. Generating an RSA key is an extremely computationally expensive process compared to generating keys for symmetric algorithms, which basically involves picking a pseudo-random number.

A common use of public key cryptography is to encrypt symmetric keys to solve the key distribution problem. It also enables a communicating party to pick a random key that will be valid for only one communication session. Suppose Alice and Bob wish to communicate. Alice sends Bob her public key. Bob then generates a random **session key**, encrypts it with Alice's public key, and sends it to Alice. Alice is now the only one who can decrypt the session key since only she has her private key, which is needed to decrypt the session key. After that, messages can be encrypted with the randomly generated session key. This type of cryptosystem, which relies on both public key and symmetric algorithms, is known as a **hybrid cryptosystem**.

The randomly generated key just mentioned is known as a **session key**. Session keys are useful because, since their lifetime is only for one conversation session, the covertness of future messages is ensured even if one key is compromised since future conversations will be encrypted with a different session key. The less data that is encrypted with one key, the less the chance that a key will be pentrated. Session keys can also be distributed to a group to allow for secure group communication. Suppose Alice wishes to multicast a message to a group containing Bob, Charles, and David. She can follow this procedure:

1. Pick a random session key, K.
2. Get the public keys of Bob, Charles, and David.
3. Encrypt K for each party: $E_B(K)$, $E_C(K)$, $E_D(K)$.
4. Encrypt the message with K: $C = E_K(M)$.
5. Send out a message containing $\{E_B(K), E_C(K), E_D(K)$, and C$\}$.

Charles, for example, will only be able to decipher $E_C(K)$ using his private key, thus obtaining K. He can now decrypt C using the key K to obtain the original message M.

Digital Signatures

We use signatures in today's society to identify ourselves uniquely. A signature has several important properties. A signature is considered to be:

- Authentic – the recipient knows that the signer deliberately signed the document.
- Unforgeable – the signature is proof that the signer signed the document.
- Not reusable – the signature is part of the document and cannot be copied onto another document.
- Unalterable – once a document is signed, it cannot be changed.
- Nonrepudiatable – the signer cannot claim that he/she didn't sign the document.

It is these properties that make signatures so indispensable in society. Unfortunately, all of these properties are completely untrue. Luckily, forging signatures and altering documents is often quite a bit of trouble. In the digital domain, however, copying and modifying files is trivially easy. We'd like the properties of signatures without the problems. For example, Alice should be able to "sign" a document so others would know that it was really Alice and that the document was not altered after she signed it.

Arbitrated protocol

We can turn to a trusted third party (or arbiter) to authenticate our messages. In this case, a third party, Trent, has the symmetric key of every user. Trent is a trusted party – he will not forge messages or give away keys. If Alice wishes to send a message to Trent, she composes a message, including the destination (Bob), and encrypts it *for herself*: $E_A(M)$. She then sends this message to Trent. Since Trent has all the keys, he can decrypt the message and know that it could have originated only from Alice (since Alice is the only other party that has Alice's key). Trent then generates a statement of receipt, adds it to the message, and encrypts the message for Bob: $E_B(M)$. He may also choose to log a record of this message along with a hash of the message. When Bob receives the message, he uses his own key to open it, knowing that it could have come only from him (it didn't) or from Trent (nobody else has Bob's key). Since Trent is a trusted party, Bob trusts Trent's attestation that the message really did originate from Alice.

Going one step further, if David wants to forward the message to Charles, he encrypts the message (along with Trent's attestation to the origin of the message from Alice) along with a destination using his own key. The encrypted message is sent to Trent, who can then look up the hash of the message in his database to ensure that it was not altered by Bob. Trent then adds his attestation that the message was also "signed" by Bob, encrypts it for Charles and sends it to him.

Digital signatures and public key cryptography

With public key cryptography, encrypting a message with one's private key is the same as signing the message! Anybody with access to the public key can decrypt the message but will know that the document could have been encrypted by the possessor of the private key. No third party is needed.

We can generate a stand-alone fixed-length signature for a message by creating a hash of the document H(M) and then encrypting the hash with our private key. If a recipient wishes to verify the signature, it produces a hash of the document and decrypts the hash we sent by using our public key. If the hashes match, the document has not been altered. This scheme makes it easy to attach multiple signatures: each party computes a hash of the message, encrypts it with its private key, and attaches it to the message. Another advantage is speed: we do not have to encrypt the entire message using public key cryptography.

If nonrepudiation is desired, we will need to turn a third party. The originator will add a header to the message containing identifying information (name, timestamp), sign the message, and send it to the third party. The third party will then add its own timestamp and ID to the message, log the transaction, and send it to both the sender and recipient (so the sender will know if someone is trying to impersonate her).

If secrecy of the message is desired, encryption can be combined with the digital signature, providing privacy as well as proof of authorship. To do this, we can pick a random key, K, with which to encrypt the message (using a symmetric algorithm). This key will then be encrypted with the *public* key of each recipient of the message. A recipient will be able to decrypt K with his private key, then decrypt the message, compute the hash, decrypt the hash attached to the message (decrypted with the sender's public key), and verify the origin and authenticity of the message. Let's look at this again:

Alice has a message, M, to send to Bob. She computes its hash, H(M) and encrypts it with her own private key: $E_a(H(M))$. This is her signature. Secrecy of the message is important in this example, so she will encrypt the signed message with a symmetric algorithm using a randomly generated key, K. The encrypted signed message is $E_K(\{M,E_a(H(M))\})$. Now she has to enable only Bob to be able to decrypt this message, so she encrypts the key, K, with Bob's public key: $E_B(K)$. Finally, she sends out the complete message: { $E_K(\{M,E_a(H(M))\})$, $E_B(K)$ }.

When Bob gets this message. He first decrypts the key, K, using his private key. Now, using K, he can decrypt the entire message with signature. Having done this, he computes a hash of M, H(M). He then decrypts Alice's signature using Alice's public key and compares the two hashes to validate the message.

Authentication

In many environments, it is more important that communications be *authenticated* rather than encrypted. That is, both parties should be convinced of each others identity. We need to establish identity and verify identity before allowing access to resources.

There are three methods we can use to authenticate someone:

1. Use something you have, for example, a key or a card. The problem is that these can be stolen.

2. Use something you know. Passwords and PINs (personal ID numbers) fall into these categories. These can be guessed, shared, and stolen by snooping.

3. Use something you are. This involves biometrics. For example, a system may examine a user's fingerprint or iris pattern. In general, these systems require hardware, can be costly, and are imprecise.

Authentication methods can be combined to strengthen the authentication. Using a single one of these methods is known as **one-factor authentication**. Using two techniques is **two-factor authentication**. Withdrawing cash at an ATM machine is an example of two-factor authentication. To authenticate, you present the ATM card (something you have) and enter PIN (something you know).

Most operating systems maintain a notion of a **user identifier** (user ID) which is a unique token that identifies each user on a system. Typically, systems employ a user name (a unique alphanumeric string that a user may use to identify himself/herself to the system) as well as well as a numeric user ID. The system uses the user ID to store and verify access permissions.

The most common method of authentication is with a simple **password authentication** scheme. The system prompts us for a user name and then for a password. It then looks up the name in a password table and sees if the passwords match. This is known as a **reusable password** since the same password is used for each login. One major weakness here is that if somebody manages to break into the system, she can steal the entire password file.

An enhancement to storing a password in plaintext on a system is to use a one-way hash function. We now have a password file that contains encrypted passwords that *cannot be decrypted*. How do we verify a password if we can't decrypt the one we have saved? When the system prompts for a password, it simply encrypts the string that you entered and compares it with the encrypted password. If they encrypt to the same string, then the system accepts the password. In recent years, passwords were moved from publicly readable files (`/etc/passwd` on UNIX) to files readable only by the administrator files. The reason behind this was that guessing passwords is often too easy and the system is vulnerable to dictionary attacks where a perpetrator would try every word in a dictionary with various modifications by adding numbers and/or symbols hoping to find a password that encrypts to the same value.

A problem with passwords is that they can be stolen through observing a user's session (snooping on packets, for example). A stop-gap measure is to require users to change passwords frequently.

Another method of combating password theft is to turn to two-factor authentication systems or to use **one-time passwords** – a new password must be used for each login.

Two-factor authentication generally involves using some form of "authenticator card". If we assume that the network is vulnerable to eavesdroppers, the card must be capable of doing some computation and cannot be a fixed set of bits as we find in a bank ATM card. One form of **challenge/response authentication** works like this: A user wants to login to a server and provides her user name. She is then given a challenge number from the server with a prompt for the response. This challenge number is entered into a challenge/response unit along with a PIN. This unit (that usually looks like a credit-card sized calculator) generates a response that is a function of the PIN, the challenge, and a key that is stored within the challenge/response unit. The response is copied back to the prompt from the server. The server maintains the user's PIN and the key inside the challenge/response unit and can perform the same calculation and thus verify the response. Any eavesdropper does not get to see two important ingredients: the key and user's PIN.

Another popular two-factor authentication scheme is through a SecureID™ card (from RSA, formerly Security Dynamics). This device maintains a clock and constantly generates a number that is a function of a seed number in the card and the current time. This number is permuted with the user's PIN and sent to the server along with a user name or ID. The server, having the seed of the card, the time, and the PIN can recreate this same number. Any eavesdropper has neither the seed nor the PIN.

Skey authentication

The Skey authentication algorithm is used to provide one-time passwords. It relies on one-way functions (an example is $x^a \bmod b$). Suppose we wish to authenticate Alice for 100 logins. We will pick a random number, R. Then, using the one-way function $f(x)$, we will generate the following list:

$x_1 = f(R)$

$x_2 = f(x_1) = f(f(R))$

$x_3 = f(x_2) = f(f(f(R)))$

. . .

$x_{100} = f(x_{99}) = f(\ \ldots f(f(R))\ \ldots)$

We will also compute $x_{101} = f(x_{100})$ and associate this value with Alice in some database (e.g. .password file). The list of numbers $x_1 \ldots x_{100}$ is given to Alice.

When Alice wants to log in, she'll present the last number on her list (x_{100}) along with her name:

Alice to host: "alice", x_{100}

The host now computes $f(x_{100})$ and compares it with the value of x_{101} stored in its database. If the values match then Alice is authenticated. In that case, the value x_{101} in the database is replaced with the x_{100} provided by Alice. Alice must now cross out the last number from his list. Next time she logs in, she'll provide x_{99} along with her name. The system will compute $f(x_{99})$ and compare it with the value of x_{100} in its database. Each number is used only once, so even if others see the authentication, there is nothing they can do with the data. When Alice uses x_1 she's out of logins and will have to see the system administrator to get a new list.

Skey authentication works only because $f(x)$ is a one way function and there is no known way to compute $f^{-1}(x)$. If somebody sees Alice enter x_{100}, there is no way that they can compute the x_{99} that is necessary for the next login.

Public key authentication

A basic form of authentication can be done with public keys. Suppose a host machine wants to know whether it's really Alice trying to log in. It can generate some random string, S, and present it to Alice. Alice then encrypts the string with her private key and sends it to the machine along with here name: { "Alice", $E_a(S)$ }. The host looks up Alice's public key in a database, decrypts the message and compares it with S. If it matches, then the host knows that only someone with Alice's private key could have encrypted S such that it could be decrypted with Alice's public key. Authentication is complete. The random string, S, is called a **nonce** in cryptographic authentication parlance. It is simply a meaningless bunch of data that is different each time it is used (to prevent replay attacks).

SKID authentication

SKID2 and SKID3 are authentication schemes that use symmetric cryptography and assume a shared secret (key) between two parties. SKID2 authenticates only one party. SKID3 has two final steps to provide mutual authentication. The principle of SKID is that you generate a random token and give it to the other party, which then returns the encrypted token to you. Since the key is shared between the two parties, you can decrypt the key and verify that the other party had the right key to encrypt it. The same protocol is done the other way around to enable the other party to authenticate you. SKID uses an encrypted hash of the random token and some other information instead of just simple encryption. We will denote this as $H_K(x)$, which means that a hash, H(x) is computed from some message(s) x and the hash is encrypted with the symmetric key K.

Here's a scenario in which Alice and Bob mutually authenticate each other:

1. Alice chooses a random number R_A and sends it to Bob.

2. Bob chooses a random number R_B. He computes an encrypted hash of the random number he received, his own, and his name: $H_K(R_A, R_B, \text{"Bob"})$ and sends it to Alice along with R_B.

3. Alice, having received R_B and already in possession of R_A can also compute the hash of { R_A, R_B, "Bob"} and encrypt it with K. By comparing the result with the value received from Bob, she can verify that Bob was indeed able to encrypt the data with K and hence possesses the shared key. Authentication is complete as far as Alice is concerned. This is where the SKID2 protocol ends. SKID3 provides mutual authentication, where Alice now has to convince Bob of her identity.

4. Alice computes a hash of {R_B, "Alice"}, encrypts it and sends it to Bob.

5. Bob computes the hash of {R_B, "Alice"} and compares it with the decryption of the value sent by Alice. This convinces Bob of Alice's identity and authentication is complete.

The essential point to note in the authentication sequence is that each party permutes data generated by the other. In effect, each party is challenging the other with data that will be different each time authentication is needed.

Combined authentication and key exchange

SKID suffers from requiring the two parties to have a shared secret. If we can combine authentication with key exchange, then two parties across a network can exchange keys and be sure that they're communicating with each other.

Wide-Mouth Frog

A protocol that accomplishes key exchange and authentication using symmetric cryptography is the Wide-Mouth Frog algorithm. It uses an arbitrated protocol where one party encrypts a message for itself containing the key and sends it to the trusted third party. This third party decrypts the message and re-encrypts it for the recipient. The problem of having n(n-1)/2 keys is avoided because the secret keys are only for the third party. To prevent replay attacks (somebody snooping on the message and sending it at a later time), a timestamp is added to each message.

If Alice wants to talk to Bob, she sends a message to Trent (the third party) encrypted with her key (A).

Alice to Trent: { "Alice", E_A(T_A, "Bob", K) }

Trent receives the message and sees that it's from Alice. He looks up her key in his database and decrypts the rest of the message. He verifies the timestamp T_A to determine whether to accept the message. Seeing that it's for Bob, he looks up Bob's key and composes a new message (using A new timestamp T_T):

Trent to Bob: { E_B(T_T, "Alice", K) }

Diffie-Hellman exponential key exchange

Diffie-Hellman is the first public key algorithm. Its use is different from RSA public-key cryptography in that it is *only* suitable for key exchange, not encryption. The publicly readable data is not really a key that will be used for encryption or decryption. The algorithm is based on the difficulty of calculating discrete logarithms in a finite field compared to the ease of calculating exponentiation.

Exponential key exchange allows us to negotiate a secret session key without the fear of eavesdroppers.

To perform this algorithm, all arithmetic operations are performed in the field of integers modulo some large number (modulo means that we divide the results by and keep the remainder). Both parties then agree on some large prime number, p, and a number α, where $\alpha < p$ and α is a primitive root of p.[1]

Each party then generates a public/private key pair. The private key for user i is X_i,

which is just a random number less than q. The corresponding public key, Y_i, is computed as:

$$Y_i = \alpha_B{}^{X_i} \bmod p$$

Now, suppose that Alice and Bob wish to talk. Alice has a secret key X_A and a public key Y_A and Bob has a secret key X_B and a public key Y_B.

1. Alice sends Bob her public key, Y_A.
2. Bob sends Alice his public key, Y_B.
3. Alice computes:

$$K = (Y_B)^{XA} \bmod p$$

4. Bob computes:

$$K = (Y_A)^{X_B} \bmod p$$

5. Alice and Bob can now use symmetric encryption using the shared key K.

[1] A primitive root, α, of a prime p is one whose powers generate all integers from 1 to p-1. That is, $\alpha \bmod p$, $\alpha^2 \bmod p$, $\alpha^3 \bmod p$, …, $\alpha^{p-1} \bmod p$ are distinct and consist of integers 1 … p-1 in some combination.

The essential point is that both Alice and Bob could generate a common shared key using their private key and the other's public key but nobody else could do so. The keys are equivalent because:

$$K = (Y_A)^{X_B} \bmod p = (\alpha^{X_A} \bmod p)^{X_B} \bmod p = \alpha^{X_A X_B} \bmod p.$$

Now that two parties can derive a common conversation key (that only they can derive) , one of them can pick a random token and send it to the other for encryption as was done in the SKID/2 or SKID/3 protocol.

UNIX secure RPC and exponential key exchange

By default Sun's Remote Procedure Call interface uses no security. As an option, it can use UNIX security, in which it passes machine name and user name to the remote procedure. Beyond that, a number of secure authentication schemes may be used. One of these is known as Secure RPC. Security data is stored in a credentials structure in the RPC handle.

In secure RPC, all encryption is via DES (Data Encryption Standard). The goal is secure authentication. If a covert conversation channel is needed, the user processes must encrypt their data. Secure RPC is based on the Diffie-Hellman exponential key exchange algorithm and generates a *conversation key*. We create a new random key each time we need a conversation channel. This decreases thee amount of data that we encrypt with our main keys and hence decreases an intruder's chance of collecting enough statistically significant data to decrypt the main keys. RPC relies on access to a database of private keys as well as to the user's private key.

Let's proceed with the authentication sequence. We will use a subscript notation to denote encryption. When we mention a user's name, we refer to a *network name* which is a union of the operating system name, user ID, and domain name. For this system to work properly, clocks on client and server must be synchronized (at least approximately).

Client (A)

1. Generate the common key $C = (K_B)^{P_A}$.
2. Create a random conversation key for this session, *CK*.
3. Select a window value, *W*. This represents the lifetime of the credential.
4. Get the current time, *T*.
5. Create the credentials structure: { user's network name, CK_C . W_{CK} }. Note that *CK*, encrypted with the common key, *C*, means that the only other user that can decipher the value of *CK* is the server (B).

6. Create a verifier structure: $\{T_{CK}, (W+1)_{CK}\}$. The server won't be able to read any of these components unless it succeeded in deciphering the conversation key.

7. Send the credentials and verifier to the server.

Server (B)

8. Read the message: get network name; use that to find that user's public key; generate the common key $C = (K_A)^{P_B}$; decrypt *CK*. Use *CK* to decrypt the verifier.

9. Look at the window value and see whether the window of valid time ($T + W$) expired. If so, reject the request.

10. Use *CK* to decrypt $W + 1$ and verify that the value really is the window plus one. This assures us (the server) that the user sending the request really does know the conversation key.

11. Send back a verifier containing a nickname to use for future requests between the two parties and $(T-1)_{CK}$. We subtract one from the time stamp that we received from the client and encrypt it with the conversation key. This way we get a different bit pattern than if we sent back the original encrypted time stamp (that would be useless) and allow for the client to verify that we really were able to decrypt the conversation key, *CK*.

Client (A)

12. Get the verifier from the server. Decrypt the time stamp using *CK* and verify it. If the remote program was able to figure out the time stamp, then it was able to decrypt *CK* successfully and must therefore indeed be the desired server B.

13. Use the conversation key to authenticate all future messages. The client's credential contains the nickname (used to index into a table to find the window size on the server) and the client's verifier contains the current time encrypted with *CK*. The server sends back a verifier of that time stamp minus one encrypted with *CK*.

Kerberos

Another authentication scheme comes from Project Athena at MIT (1983-1988). Project Athena's goal was to create a computing environment around high-performance workstations with distributed servers. This scheme is based on trusted third party authentication. It assumes that the network is insecure (snoopable). To access a service you must be authenticated to use it and present an authentication *ticket* to the service. Since the network is insecure, passwords are never sent across in cleartext. Every user and every service has a password. Kerberos is a trusted third party that knows all the passwords. We'll take a look at a sample authentication scenario to get a feel for how this

system performs authentication. Let's assume that *Marge* wishes to access the service *Homer*. Both *Marge* and *Homer* have passwords (or keys). For the purposes of this example, let's assume the following keys are assigned (we'll see where the session key and conversation key come from as we go along):

marge	18
homer	57
session key	28

1. Marge contacts the Kerberos authentication server (AS), requesting a "ticket" to communicate with the Homer.

2. The Kerberos Authentication Server looks up Marge and the service Homer to determine whether she is indeed allowed to access it. If she is, the Authentication Server generates a session key (28) and sends it back to Marge along with an identifier of Homer's service (e.g., IP address and port). The entire message is encrypted with Marge's key so that anybody else who sees this message will be unable to make sense of it: $\{\text{hom}er\_service, 28\}_{18}$.

3. The second message that Marge receives from the Kerberos Authentication server is the same session key and her identifier. This time, the message is encrypted with Homer's secret key, so she is unable to decode this: $\{m\text{arg}e, 28\}_{57}$. This message is known as a *sealed envelope* or *ticket*.

4. Marge is now ready to establish communications with Homer. She first decodes the session key (28) and information on how to access the Homer service from the first message that she received (the one encrypted with her secret key). She then sends Homer the sealed envelope that she received from Kerberos along with a timestamp that she encrypted with the session key that she received from the Authentication Server.

5. Homer gets the message and (if the service really is Homer), can decrypt the sealed envelope using his secret key. Doing so reveals the session key and an identifier for Marge. Because he was able to decrypt this message successfully, he knows that it must have been generated by a trusted party – one that has his password. Now that he has the session key, he decrypts the encrypted timestamp that he received from Marge and checks whether it is within a given time window. If it is within a valid range, he realizes that the message is not a *replay attack* and that the message must really have come from Marge because Kerberos would not have divulged the session key to anyone else.

6. Homer now sends a message to Marge to prove to her that he really is homer. This message contains his identifier and the timestamp that he received from Marge. The message is encrypted with the session key: $\{\text{hom}er\_service, T\}_{28}$.

7. When Marge receives this message, she decodes it using the session key. She is convinced that the remote party is Homer because only he would have been able to decode the ticket to extract the session key so that he could decrypt the timestamp she sent.

Now that authentication is ccmplete and both parties have the session key, they can communicate securely, encrypting each message with the session key.

Popular algorithms

Several algorithms for encryption and hashing are very widely used and it is difficult to go to a party or get a haircut without some of their names popping up. This section will present the names but not give algorithmic details. The interested reader can consult a reference like Schnier to appease any curiousity.

MD5

MD5 is a message digest (hash) algorithm created by Ron Rivest of MIT (the 'R' in RSA). Its input is a message of arbitrary length and its output is a 128 bit message digest.

SHA

SHA stands for Secure Hash Algorithm. It is a hash algorithm based on MD4, a precursor to MD5, created by the National Institute of Standards and Technology in 1993. It produces a 160-bit message digest. Because of the longer output, it is harder to produce another message that yields the same digest. On the other hand, it requires more computational steps than MD5 (80 vs. 60), making it approximately 25% slower.

IDEA

The International Data Encryption Algorithm was created by Xuejia Lai and James Massey at the Swiss Federal Institute of Technology in 1990. It uses a 128 bit key to encrypt data in blocks of 64 bits. It is widely used by the European community and by packages such as PGP (Pretty Good Privacy).

DES

The Data Encryption Standard, created in 1977 is one of the most widely symmetric encryption algorithms. The National Institute of Science and Technology (NIST) reaffirmed DES for federal use for another five years in 1994. Under DES, data is encrypted in 64-bit blocks using a 56-bit key. The 56-bit key is cause for worry because trying 2^{56} combinations is not as formidable a task as it once was. To strengthen the encryption, one might wonder whether using two keys would work (encrypt once with one key and then again with the second key). This would be useless if we could always find a single 56-bit key such that:

$$E_{K'} = E_{K_2}(E_{K_a}(M)).$$

Fortunately, this does not hold for DES and multiple encryptions do strengthen the result. However, there is a problem with double-DES known as the "meet in the middle attack". If we know some pair (M, C), then we can (1) encrypt M for all 2^{56} values of K_1 and (2) decrypt C for all 2^{56} values of K_2. For each match of (1)=(2), we can test the

potential key against another (M, C) pair If there is a match, then we can assume that the keys have been found. This attack can be avoided with three stages of encryption and two keys. The approach is known as **triple-DES** and works as:

$$C = E_{K_1}(D_{K_2}(E_{K_1}(M)))$$

The use of decryption in the middle stage allows for compatibility with single DES, where both K_1 and K_2 can be the same.

Skipjack

Bill Clinton said that this algorithm "will bring the Federal Government together with industry in a voluntary program to improve the security and privacy of telephone communication while meeting the legitimate needs of law enforcement". The hardware embodiment of this algorithm is known as *Clipper*. It is stronger than DES, but allows the escrow a pair of keys at two separate escrow agencies. At least once during the message, the sender must transmit a "Law Enforcement Field" (LEAF) that will enable law enforcement agencies to recover the message. This field enables the agency to decrypt the session key if it has the two escrowed keys (for which it would need a search warrant). The algorithm is classified as secret by the U.S. government to "ensure" that inferior or incorrectly-implemented chips are not made.

ISO authentication framework

Public key systems have great appeal for key exchange, but if we want to use them as a basis for authentication, we need something that will bind one's identity to the public key. If you ask for Fred's public key, how can you be sure that it really belongs to Fred and not some imposter? How can you find key identifying information about Fred and his public key without contacting him? One option is to maintain a centralized database of public keys. The problem with this is that one must always turn to this trusted source for the distribution of keys: you cannot trust a key that was passed on to you by an untrusted party.

The International Standards Organization (ISO) introduced a set of protocols known as X.509 protocols to provide standards for authentication across networks. While no standard algorithms are specified, RSA public key encryption is recommended.

The most important part of the ISO framework is the structure for public key **certificates**. Each user has a unique name (called a distinguished name) that is a collection of several attributes including the user's real name, organization, locality, and country. A trusted **certification authority** (CA) issues a signed certificate that contains the distinguished name and the user's public key. This certificate is signed by the CA. A certificate looks like this:

version	serial #	algorithm, params	issuer	validity: from, to	distinguished name	Pub. Key: alg, params, key	signature of CA

If Marge wants to talk to Homer and they have a common CA, she can get his certificate from some database (or some source). She can then verify the signature of the CA (by hashing the contents of the certificate and comparing them with the CA's signature decrypted with the CA's public key). This gives Marge assurance that Homer's certificate was indeed generated by the same certification authority.

A more complicated problem is when a different certification authority certified Homer than Marge. Certification authorities are designed to fit into a hierarchical structure. Each CA has a certificate signed by the CA above it and by the CA below it. If you have a certificate signed by an unknown CA, you can ask the CA for its key and see which higher-level signed it. If that is also unknown, you can repeat the process until a CA is reached (a common root). This is known as *certificate chaining*. If a common root isn't reached then you may choose not to honor it.

The process of certificate discovery is not consistently implemented at this time. One possible mechanism is for Marge to move up her chain of certification authorities and look at the graph of CAs below, searching for the CA which certified Homer. A more reasonable approach may be for Marge to know her entire chain of CAs and follow Homer's chain until a common CA is located and Homer's identity may be trusted.

Certificates can also be revoked by a CA and each CA is responsible for maintaining a certificate revocation list.

Certificate-based authentication is either one-way, two-way, or three-way:

one-way	Marge authenticates herself to Homer
two-way	Mutual authentication: Marge authenticates herself to Homer and gets a reply from Homer to establish his identity
three-way	Another message is added from Marge to Homer to avoid the need for timestamps (and authenticated, synchronized time).

The authentication from Marge to Homer works as follows:

1. Marge generates a random number, R_m. She constructs a message: $M = \{T_m, R_m, I_h, d\}$ where is T_m a timestamp, I_h is an identifier for Homer, and d is arbitrary data.

2. Marge sends her certificate, M encrypted for Homer and sighed by Marge.

3. Homer verifies Marge's certificate and obtains her public key. He then decrypts M using his private key and verifies Marge's signature.

4. Homer checks I_h for accuracy to ensure that Marge really intended to address him and has the right identifier for him. He then checks the timestamp to see whether it's a current message. Optionally, Homer may check R_m in a database of previously used random numbers if he's concerned about a replay attack and doesn't trust the timestamp.

To support two-way authentication, Homer would send back the original random number and a new one. For three-way authentication, Marge would encrypt the random number sent by Homer and send it back to him.

Public Key Cryptography "Standards" (PKCS)

No comprehensive set of standards currently exists to encompass encryption, authentication, certificates, key formats, and data formats. The closest to this is the set of Public Key Cryptography Standards (PKCS) put forth by RSA (now owned by Security Dynamics). These standards are numerically referenced (e.g. PKCS7):

1	A method for RSA encryption/decryption, digital signature construction.
3	A method for implementing Diffie-Hellman key exchange.
5	A method for encrypting messages with a secret key derived from a password (for encrypting private keys in transit).
6	Syntax for public key certificates. This is a superset of X.509 and includes other data such as email address.
7	A general syntax for data that may be encrypted or signed (envelopes or signatures). This is a recursive structure, allowing envelopes or signatures to be nested.
8	Syntax for private key information.
9	Defines attribute types for #6 certificates, #7 messages, and #8 public key information
10	Syntax for certification requests.
11	Cryptographic token API standard; a programming interface (Cryptoki) for portable cryptographic devices.
12	Syntax for storing public keys, private keys, and certificates.

Secure Sockets Layer

The secure sockets layer (**SSL**) is a layer of software that sits on top of TCP/IP to provide authentication via RSA and X.509 certificates. The goal of SSL is to provide an encrypted

(possibly authenticated) channel over which two parties may communicate. It enables conventional insecure services, such as http, telnet, ntp, ftp, smtp to engage in secure, authenticated transactions. SSL is gaining popularity particularly on the web as a layer under the HTTP protocol (https). Browsers such as Microsoft's Internet Explorer and Netscape Navigator support certificate storage and SSL authentication.

SSL authentication is broadly defined and it is up to the server to determine how much authentication needs to be done. The basic protocol is simple:

1. The client sends a "hello" to the server.

2. The server replies with a “hello”. These client/server greetings serve to exchange and establish the protocol version, session ID, cipher suite, and compression method.

3. If the server is to be authenticated by the client, the server sends a certificate (which contains the server's public key to the client). Otherwise, the server will send a basic key exchange message (a public key without a certificate).

4. The server then sends a “hello done” message.

5. If the server requested a client certificate, the client now sends its certificate (or a “no certificate alert” message).

6. The client creates a session key, encrypts it with the server's public key, and sends it to the server. The key transfer is complete.

7. The client may send a “change cipher spec” message to set a specific cipher mode.

8. Data is now encrypted with RC4 symmetric encryption.

Smart Cards

Now that you can have a digital identity (for example, a digital certificate), the issue becomes one of using it. All may be fine if the certificate is loaded into your Netscape browser and that’s all you use, but the problem escalates once you start thinking about using multiple machines or possibly untrustworthy machines; you don’t want to leave your private key all over the place. One approach to this is the use of smart cards. The definition varies. In the simplest case, a smart card can be a device with only memory (ROM, PROM, or EEPROM). This makes them not much different from most credit cards that already have a magnetic strip. More properly, smart cards contain some processing logic on them: either special-purpose security logic or a general-purpose CPU.

A relatively popular early attempt at a standard for smart cards is *Fortezza*. It is targeted toward users of portable computers and PCs as a standard way to attach a public key cryptographic protocol to a computer system. Fortezza provides a set of software interfaces and a cryptographic engine implemented as a hardware device that plugs into

a PCMCIA slot. The goal is that a variety of hardware devices will be compatible with this standard.

The current implementation is that of a credit card size PCMCIA card that uses the Clipper chip. It stores private and public keys for each of its possible users and performs digital signature and hash functions. It also provides block data encryption/decryption at high speed. Other cards may use different underlying methods in the future but the goal is for the software interface to remain the same.

Every user has a PIN and the card cannot be used unless a PIN has been entered recently. The system uses an X.500 compatible directory containing certificates where the issuer of a certificate will be an X.500 server administered by a trusted agency on behalf of the Fortezza authentication domain. Fortezza also has knowledge of the public keys associated with the trusted directory services to enable it to validate certificates from the directory.

In recent years, there have been additional standardization efforts. The International Standards Organization produced a comprehensive set of standards for smart cards, blanketed under ISO 7816. The seven working groups include:

- 1: Physical characteristics and test methods for ID-cards
- 3: Identification cards - Machine readable travel documents
- 4: Integrated circuit card with contacts
- 5: Registration Management Group
- 7: Financial transaction cards
- 8: Integrated circuit cards without contacts
- 9: Optical memory cards and devices

In 1996[2], Sun announced the Java Card application programming interface (API). This allows Java applications ("cardlets") to run on any ISO 7816-4 compliant card. The firmware on the card contains a base operating system with a layer over it containing the Java Card virtual machine. The programmer has access to a number of class libraries: many of the well-known Java libraries as well as card-specific interfaces and security functions. Multiple cardlets can run concurrently and every byte-code is validated during interpretation to ensure that only valid resources are referenced. The operating environment is restrictive compared to running Java on a PC: 24K bytes of ROM, 16K bytes of EEPROM, and 512 bytes of RAM. A limited set of data types are supported and only single-dimensional arrays.

Biometric authentication

A form of identification that is particularly interesting in many environments is biometric authentication: using some aspect of a person's biological information to establish identity. The appeal is obvious: while a password can be stolen or given away, biometric data cannot. A number of human characteristics can be measured in performing such

[2] Version 2.0 was defined in 1997. The current revision is 2.1.

authentication. These include fingerprints, iris scans, retina scans, face pattern, hand geometry, signatures, voice, and DNA. All biometric authentication is based on *statistical pattern recognition*. Direct comparisons cannot be performed and thresholds are used to determine whether the data should be accepted as a match or not. A plot of false rejects vs. false accepts can be associated with each biometric authentication device. This plot is known as a ROC (Receiver Operator Curve) curve. It is up to the implementor to pick a point on this curve that will provide an acceptable level of security. For example, to dramatically decrease the level of false accepts will require that the number of false rejects increases. This will increase user frustration, since valid users will get rejected in higher numbers and will have to re-authenticate.

As current technology stands, fingerprint and iris scan systems provide the best recognition rates. They are also more suited for matching one scan against multiple stored profiles. Face and hand geometry systems are not good at assuring uniqueness and require one-to-one matching against a stored profile. This requires users to enter additional information during authentication, such as a user name or some identifying number. Signature and voice are considered to be behavioral systems rather than physical systems; they can change with a user's demeanor and hence tend to have lower recognition rates.

Appendix 1: How the RSA algorithm works

RSA is a public key algorithm which generates two keys and allows data encrypted with one of them to be decrypted with the other (and vice versa). It was created by Ron Rivest, Adi Shamir, and Leonard Adleman, hence the name RSA. The algorithm is based on the difficulty of factoring large numbers. Public and private keys are functions of a pair of large (100-200 or more digits) prime numbers.

To generate the two keys (a public and a private key):

1. Choose two random large prime numbers, p and q (preferably of the same length).
2. Generate the modulus, $n = pq$.
3. Choose a random encryption key, e, such that e and $(p\text{-}1)(q\text{-}1)$ are relatively prime.
4. Use the extended Euclidean algorithm to compute the decryption key, d, such that:

$$M_i = C_i^d \bmod n$$

thar is,

$$d \equiv e^{-1} \bmod((p-1)(q-1))$$

5. Discard p and q.

To encrypt a message, divide it into numerical blocks smaller than n (i.e., divide it into chunks so that each chunk will be treated as a number whose value is less than n). The encryption of each chunk M_i *is:*

$$C_i = M_i^e \bmod n$$

Decrypting a chunk requires performing the same operation using the other key (d):

$$M_i = C_i^d \bmod n$$

This works because:

$$C_i^d = (M_i^e)^d = M_i^{k(p-1)(q-1)+1} = M_i M_i^{k(p-1)(q-1)} = M_i \cdot 1 = M_i$$

where all operations are modulo n.

Appendix 2: Networked file systems and other services

It is time to revisit a few services mentioned in the past and see how they tie into security.

Sun's NFS

- The administrator of a server may choose which clients can mount a file system.
- The administrator can export any directory on a file system, not just the root
- A file system can be exported as read-only or read-only to some clients and read-write to others.
- System V release 4 supports Secure NFS, which uses secure RPC
- No user ID mapping is available - NFS assumes that user IDs are the same across all machines. ID 0 (root) is mapped into user 60001 (a non-existent user). When exporting a directory, the administrator may map ID 0 from clients to an alternate ID.

System V's RFS

- You can choose which machines may mount resources.
- A directory may be advertised as read-write.
- An optional password mechanism is supported between client and server.
- Optional ID mapping is supported to allow user IDs on the client to be mapped onto local user IDs on the server.

AFS, Coda, DFS

All of these systems use Kerberos for authentication and key exchange. DFS uses the authentication service provided by the Distributed Computing Environment (DCE), which is Kerberos. RPC under DCE uses Kerberos as well.

NTP

The Network Time Protocol supports digital signatures on time stamps to prevent spoofing of time reports. By verifying the signature, you can verify that the time stamp came from a trusted source.

FTP

The file transfer protocol uses simple login/password authentication but some versions also support Skey authentication.

Web access

Web access may use either no authentication, basic authentication (user name, password "encrypted" in base 64), or SSL authentication. SSL may involve mutual authentication (both parties exchange certificates and authenticate), one-way authentication (client authenticates server), or no authentication and only an exchange of public keys to perform a symmetric key exchange.

References

Building Secure and Reliable Network Applications, Kenneth P. Birman, © 1996 Manning Publications Co.

Firewalls and Internet Security, William R. Cheswick and Steven M. Bellovin, Addison-Wesley © 1994 AT&T Bell Laboratories.

An Introduction to Operating Systems, Harvey Deitel, ©1984 Addison Wesley.

DoD trusted computer system evaluation criteria, aka. the "Orange Book", DoD 5200.28.STD, DoD Computer Security Center. Available for ftp from ftp.cert.org as /pub/info/orange-book.Z.

Smart Card Technology: Introduction to Smart Cards, Davod B. Everett, 1998, http://www.smartcard.co.uk/tech1.html.

SET (Secure Electronic Transactions), Jonathan Gorham, ©1997 Jonathan Gorham, http://www.maristb.marist.edu/~kts5/@httpd/set/set.htm

IBM SET: white paper (and other links, IBM Corporation, ©1998, http://www.ibm.com/Security/html/set.html.

The Multics System: An Examination of Its Structure, Elliott I. Organick, ©1972 Prentice Hall, pp. 127-186.

Networking Applications on UNIX System V Release 4, Michael Padovano, ©1993 Prentice Hall.

SET: Secure Electronic Transaction Specification, HTML version 1.0, October 1997, Reimer AG, Switzerland, *http://www.reimer-ag.com/reimer/Finanz/SET/index.html*.

Applied Cryptography Protocols, Algorithms, and Source Code in C, Bruce Schneier, pub. John Wiley & Sons, Inc. (c) 1996 Bruce Schneier

Operating System Concepts, Abraham Silberschatz and Peter B. Galvin, © 1994 Addison-Wesley. Pp. 431-473.

Network and Internetwork Security, William Stallings, ©1995 Prentice Hall.

UNIX Network Programming, W. Richard Stevens, ©1990 Prentice Hall.

Distributed Operating Systems, Andrew Tanenbaum, ©1995 Prentice Hall.

Modern Operating Systems, Andrew Tanenbaum, ©1992 Prentice Hall, pp. 181-202.

Private discussions with Lawrence O'Gorman, Chief Technical Officer of Veridicom, Inc., manufacturers of fingerprint recognition systems.