DEA d'Informatique Fondamentale et Applications
Université de Marne la Vallée
77454 Marne la Vallée FRANCE

LIX - Laboratoire d'Informatique
École polytechnique
91128 Palaiseau FRANCE

Traineeship report

Digital Certificates and the Feige-Fiat-Shamir zero-knowledge protocol

Daniele Raffo

Supervisor: François Morain

July 11, 2002

To Cécile / Cilcee

Contents

Chapter 1

Introduction

This report describes the traineeship that I have carried out from March to July 2002 at LIX, the Computer Science Laboratory of the École polytechnique in Palaiseau. This was in the framework of my studies in the "DEA d'Informatique Fondamentale et Applications, filière Réseaux" (Master Thesis in Computer Science, specialization in Networking) at the Université de Marne la Vallée.

The traineeship has been supervised by François Morain, head of the Cryptology team of LIX.

Two main subjects have been developed during this period.

During the first part of the traineeship, I have studied the Digital Certificates infrastructure used for secure connection over the World Wide Web. I have examined their use in the Netscape browser, and I have investigated the possibility of implementing a different authentication system to substitute the one embedded onto Navigator. For this purpose, the Open Source version of the Netscape browser, Mozilla, has been studied. At the end, I explain my conclusions and show the possible alternatives.

In the second part, I have examined the zero-knowledge protocols, with particular reference to the Feige-Fiat-Shamir identification scheme. I have implemented in C language the Feige-Fiat-Shamir protocol using the GMP library. The aim was to analyze the possibility of using this protocol to build an alternative authentication scheme for SSH, the Secure Shell.

This traineeship gave me the possibility to explore and work on several security systems and protocols as those utilised in the WWW and the Internet, and greatly improve my knowledge of these subjects; even if the short time allowed

did not suffice for a complete research in a broad field like Cryptography. This period has been very formative to me, and I have enjoyed very much writing this report.

1.1 Acknowledgements

First, I thank my supervisor, François Morain, for offering me this traineeship, being my guide, and giving me the possibility to learn more about the amazingly interesting world of Cryptography.

I want also to thank all my colleagues and members of the LIX for their help and their good advices during my traineeship; especially Régis Dupont, Andreas Enge, Francesco Logozzo, Emmanuel Thomé, and the sysadmin Houy Kuoy.

Finally, I would like to thank my family, for their constant moral and material support during these years of study.

Part I

Digital Certificates

Chapter 2

Public Key Cryptography

2.1 Internet security and PKC

Most of the secure communications today on the Internet use *public key cryptography* (PKC), also called asymmetric cryptography. This is a kind of cryptography that uses pairs of keys; a *public key* for the encryption, that must be known from the world, and a *private key* for the decryption, that needs to be kept secret. This is opposed to *private key cryptography*, or symmetric cryptography, where the same key is used for encryption and decryption. While private key ciphers were widespread since the ancient times, public key cryptography has been discovered only in 1976, by Whitfield Diffie and Martin Hellman.

Examples of private key cryptographic algorithms, or *ciphers*, are DES (Data Encryption Standard) with its improvement Triple DES, IDEA (International Data Encryption Algorithm), Lucifer, and Skipjack; and the more simple and ancient class of substitution and transposition ciphers, like Caesar and Vigenere. Examples of public key cryptographic algorithms are RSA (Rivest-Shamir-Adleman), Knapsack, ElGamal, and DSA (Digital Signature Algorithm). The most spread PKC cipher today is the RSA. [1]

In PKC, the key pairs have the property that the encryption key is different from the decryption key; furthermore, both are obtained together from a mathematical function, and it is computationally infeasible to guess a key knowing the other.

In a PKC protocol, the sender picks the public key of the recipient, encrypts

[1] For an introductory book about cryptography, read [SS99].

the message with this public key and sends it. The recipient then decrypts the message with his private key.

The big advantage of this system is that it simplifies key management, as there is no single encryption/decryption key that needs to be shared between sender and recipient, and that therefore might be intercepted.

Unfortunately, PKC is usually much slower than symmetric cryptography; for instance, RSA is about from 100 to 1000 times slower than the symmetric cipher DES. For this reason, when two parties want to communicate securely, one party computes a random session key that is sent via PKC to the other party. The rest of the communication is then encrypted with a symmetric cipher, using the session key.

Public and private keys may be used also to generate an *electronic signature*, that ensures authentication, integrity and non-repudiation of a message (but not confidentiality).

First, the message is fed to a hash function[2] like Message Digest 5 (MD5) or Secure Hash Algorithm (SHA) to generate an unique hash value called *message digest*. The message digest is then encrypted by the sender with his private key and appended to the message. The recipient will use the public key of the sender to decrypt the message digest, and then will compute again, with the same hash function, a message digest of the message received; if the two message digest do not match, this means that the message has been altered by a malicious third party during the transfer. (See Figure 2.1.)

An electronically signed message is readable also by those recipients who do not have access to cryptography functions, even if at this point they will be unable to check the integrity of the message. Also, encrypting the message digest takes much less time than encrypting the whole message.

Of course, it is possible to combine both digital signature and encryption in the same message, to ensure confidentiality too. In this case, the message is being signed, and then the message and its signature are encrypted in bulk. Signing the message after encrypting it would leave a door opened to different kind of attacks: an eavesdropper who intercepted the message could easily replace the sender's signature with his own signature. In this case the recipient might believe

[2] An *hash function* is a mathematical function which, given a long message, distills it into a small number, typically 128 (for MD5) or 160 (for SHA) bytes long. This function has the property that every bit of the small number is potentially influenced by every bit of the message; and, knowing the small number, it is computationally infeasible to find the originating message.

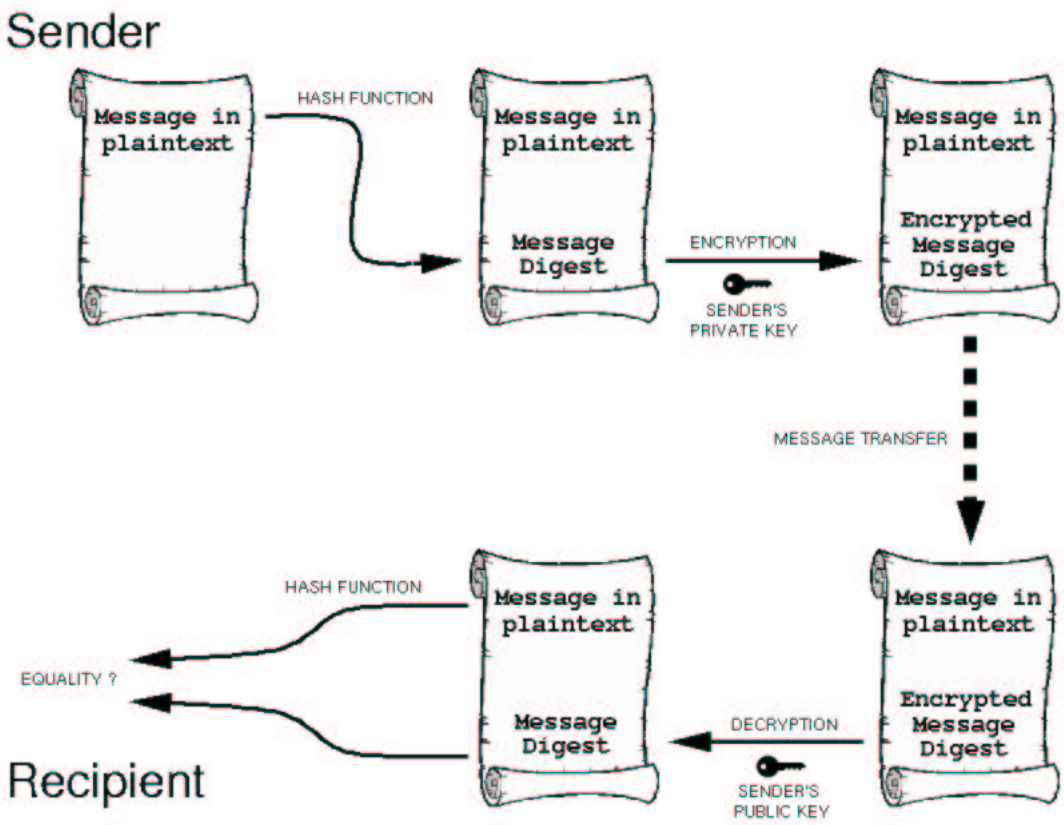

Figure 2.1: How electronic signatures work.

that the eavesdropper was the sender, and might send him confidential information [IM90]. [AN95] explains another kind of falsification in which the recipient can forge a new message and prove that the sender signed the forged message instead of the original one.

One problem of the PKC is the difficulty to bind a public key to the legal owner - that is, being sure that a specific public key is owned by a specific person, and not by an impostor that could then decrypt the private messages supposed to be sent to the person. The Digital Certificates system makes some effort in the aim to solve this problem.

2.2 RSA

The RSA cipher, invented by Ron Rivest, Adi Shamir and Leonard Adleman in 1978, is the easiest and the most famous public key algorithm.

The public and private keys are generated as follows:

I. Pick two large prime numbers p and q, and compute $n = pq$ and $u = (p-1)(q-1)$.

II. Choose a number e which is relatively prime to u. This is the encryption key.

III. Compute, via the Euclid's algorithm, $d = e^{-1} \bmod u$. This is the decryption key.

IV. e and n are the public key; d is the private key.

A plaintext message m is then encrypted[3] into a ciphertext c by the formula $c = m^e \bmod n$. Similarly, to decrypt a ciphertext c into a message m, use the formula $m = c^d \bmod n$. In these encryption and decryption formulas, the values of e and d may be switched as well [BS94].

The security of RSA comes from the difficulty of factoring large numbers. If an attacker could factor n into p and q, he would compute d and e and therefore break the cipher. Nowadays are utilised values of n which are 1024 bits of size; while multiplying two large prime numbers to obtain n is an operation relatively fast, finding the two factors of a 1024 bits long n is computationally infeasible. With the improvement of computing speed, certainly this key size will need to be augmented in the future.

[3]It is necessary that $m < n$; if this is not the case, you have to divide m in numerical blocks such that each block is $< n$, and encrypt each block individually.

Chapter 3

Digital Certificates

3.1 What Digital Certificates are

A *Digital Certificate* (or *Digital Passport*, *X.509 Certificate*, *Public Key Certificate*, *Security Certificate* or *Digital ID*) is a kind of online passport which contains the identity name of a computer or a person, and its public key. It is issued by a Certification Authority (CA), a trusted third party which creates a certificate upon request and signs it with its private key.

For instance, from the VeriSign web site[1] it is possible to obtain immediately a personal certificate (VeriSign Class 1 Digital ID, free 60-days trial edition), for use with Netscape Navigator.

3.2 The X.509 v3 standard

The standard used nowadays is the X.509 v3, by the International Telecommunication Union Telecommunication Standardization Sector. Used in the SSL, S/MIME, IPSec and Secure Electronic Transaction protocols, this standard defines eleven fields as follows [HB01]:

Version: The certificate format.

Serial Number: An unique integer value issued by the CA.

Signature Algorithm Identifier: Ignored by most implementations, this field repeats the information in the **Signature** field [ST00].

[1] `https://digitalid.verisign.com`

Issuer Name: The X.500 name of the CA.

Period of Validity: The first and last date for which the certificate is valid.

Subject Name: The name of the certificate's owner. This is the name and email address (and possibly other information) if the certificate is issued to a person, or the IP address if the certificate is issued to a server.

Subject Public Key Information: The public key of the certificate's owner, and the cryptographic algorithm of its public key.

Issuer Unique Identifier: Optional. Identificator of the CA.

Subject Unique Identifier: Optional. Identificator of the certificate's owner.

Extensions: Extension fields added in version 3 of the X.509 standard.

Signature: The signature of the CA, and the cryptographic algorithm of its public key.

3.3 Secure Socket Layer

Secure Socket Layer (SSL) is a separate protocol layer for security, developed by Netscape in 1994. The development of SSL became later the responsibility of the Internet Engineering Task Force, that renamed it as Transport Layer Security (TLS). SSL inserts itself between the Transmission Control Protocol layer and the application layer, as shown in Figure 3.1.

HTTPS, the SSL-secured version of HTTP, uses the Digital Certificates infrastructure to secure the communications in Web browsers.

The browser has a list of Certification Authorities he trusts; i.e. he has in its database all the public keys, needed to verify their signatures, of those CAs. These public keys are declared in CA certificates, typically self-signed[2], stored in the browser's database. The contents of a typical CA certificate, as it is stored in the database of the Mozilla browser, are shown in Figure 3.2.

[2]So the CA certificate is signed by the very authority (and public key) that it purports to identify. This may seem weird, but actually the CA certificates comes inside well-known programs like Netscape Navigator or Microsoft's Internet Explorer; this means that they are trusted by the people who created the programs [GS02].

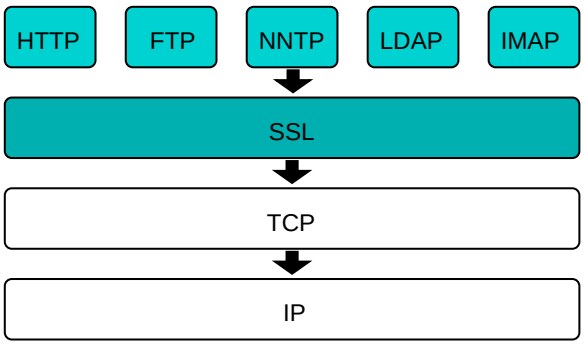

Figure 3.1: SSL stays between TCP and applications.

VeriSign was the only CA recognized in the first version of the Netscape browser; nowadays, many more others CA exist. Figure 3.3 shows a part of the list of CAs trusted by Mozilla v0.9.9.

When a browser client opens a connection to a secure WWW server (whose URL starts with `https://`), the server sends its certificate to the client, to supply it with the server's public key. Further communications are then encrypted, in a transparent way for the user. This one-way use of certificates is the most common in the World Wide Web.

If the site presents a certificate signed by a Certification Authority that is not in the browser's database, this means that the public key included in the certificate belongs to the server, but the server's identity is certificated by an unknown authority. The user is given the choice to accept or refuse it, and should decide depending of the security level he requires. The site may also present a certificate signed by itself; this leads to the same security problems about host authentication in SSH (see Chapter 8).

The user may own a personal certificate, whose purpose is to identify him in front of a WWW server like those of a bank or a corporation; in this case, the certificate is stored in the browser's database, and submitted upon request to the server to identify the user.

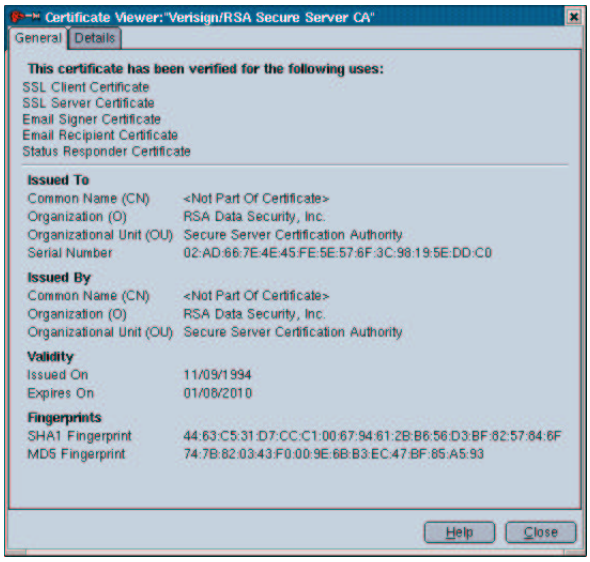

Figure 3.2: Details of the fields of a CA certificate.

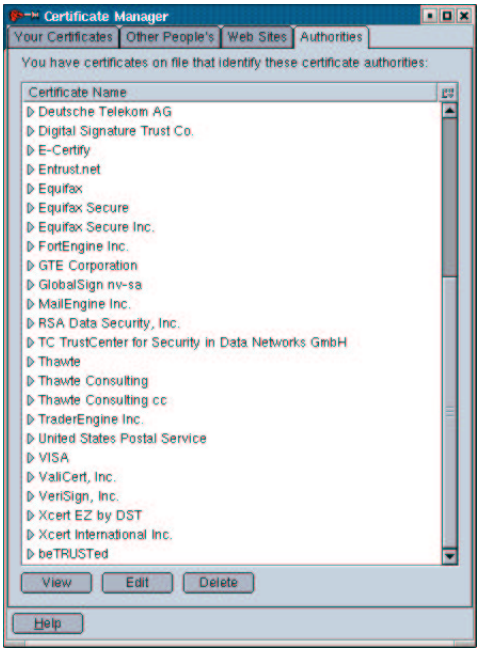

Figure 3.3: The window of the Mozilla browser listing all trusted CAs.

To sum up, there are four types of certificates: *server certificates*, which are assigned to a Web site and allow to identify it in front of a user; *personal certificates*, which are assigned to a person and allow to identify him in front of a Web site; and *CA certificates*, which are self-assigned to and self-signed by a Certification Authority, and allow a browser to verify the signature of all other types of certificates signed by the CA. The fourth type of certificate are *software certificates*, which are assigned by a developer to the software he makes and distribute, like ActiveX components or downloadable executables [GS02].

All these certificates are memorized in Base64 encoding.

3.4 Secure MIME

The Digital Certificates may also be used to send secure mail via Secure MIME (S/MIME). In order to send a signed email message, one must first own a personal certificate, that will be stored automatically by the mailclient. The certificates of other peoples, needed to decrypt their encrypted emails or to verify their signed emails, are also automatically stored in the database.

Chapter 4

Netscape/Mozilla

To investigate the possibility of substituting some module in a piece of software, it is essential first that we are granted access to code source; this is why Netscape has been chosen.

4.1 The Mozilla project

On 1998, Netscape made freely available to all users the code source for Navigator, and announced the creation of Mozilla.org[1], a non-profit organization devoted to the development of the new browser Mozilla. After the version 4, all further versions of Netscape would have been built over the basis of Mozilla.

At the creation of Mozilla.org, the code of the browser (mostly in C and C++) was completely rewritten from scratch to ensure a true cross-platform development.

Mozilla is consequently the Open Source version of the Netscape browser.[2] In its user graphical interface, it looks like a clone of Netscape Communicator (see Figure 4.1). It has some more functionalities, like some Preferences privacy settings, and is shipped without some proprietary plugins which anyway may be installed by hand by the user. The latest release of Mozilla is the 1.0 (June 5, 2002), which corresponds to version 6 of Netscape.

[1] `http://www.mozilla.org`

[2] For an explained approach to the philosophy of Open Source, see [ESR99].

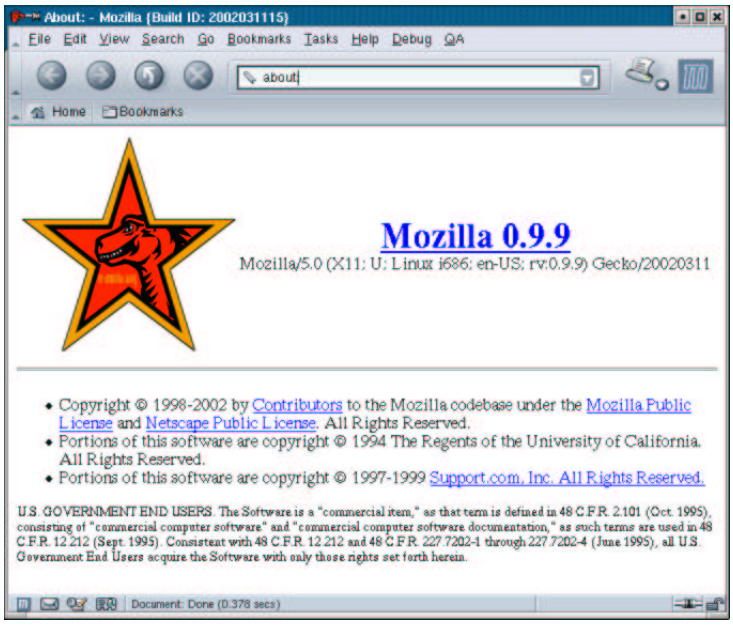

Figure 4.1: Mozilla's "about" page.

4.2 Open Source security libraries in Mozilla

The Network Security Services (NSS) is a set of libraries, APIs and utilities designed to support cross-platform development of security-enabled client and server applications. It provides a complete Open Source implementation of the cryptographic libraries used by Netscape (and other companies) in the Netscape 6 browser. Amongst others, NSS supports SSL v2 and v3, TLS, S/MIME and X.509 v3 certificates.

The Personal Security Manager (PSM) is a set of libraries and a daemon designed to the same aim, and are built on top of NSS.

Both NSS and PSM are shipped with Mozilla. One should look in these libraries in order to find the code that implements certificate management. More information may also be found on `netscape.public.mozilla.crypto` and `netscape.public.mozilla.security` newsgroups. Unfortunately, only SSL and IMAPS are integrated in Mozilla, not yet S/MIME; so signed and encrypted mail are not supported.

Chapter 5

Conclusions

I have investigated the possibility to substitute the modules that implement encrypted communications in the Netscape/Mozilla browser.

The cryptography system used in Netscape/Mozilla uses Digital Certificates, standard X.509. This standard is based on public key cryptography and digital signatures, that represent today the most secure protocol for communications confidentiality and integrity.

The X.509 standard is widely spread on the Internet and is used in the SSL protocol. Hence, this system is recognized by all the mainstream WWW browsers and mailclients. Because of incompatibility, it might not seem convenient to adopt another cryptography standard.

For these reasons, the base of this security system should not be modified. However, it could be convenient for a closed environment to rely on its own Public Key Infrastructure (PKI).

5.1 Quality problems on today's CAs

Unfortunately there are some security problems with CAs today [GS02]. These problems include inconsistencies in the X.509 Subject and Issuer fields, the difficulty of a non-ambiguous identification of a CA, the existence of "ghost CAs"[1],

[1] In [GS02], Garfinkel points out the case of the Colegio Nacional de Correduria Publica Mexicana, A. C., whose web site is blank and "under construction" since the year 2000. Yet this CA is happily trusted by Internet Explorer.

and certificate expiration dates unrealistically far in the future[2]. Correct identification is very important, as a successful attack on a CA will allow an enemy to impersonate whomever he wants, by binding from the compromised CA a certificate with public key of the enemy's choice to the name of another user.

The creation of a proprietary PKI would permit to resolve most of these problems.

5.2 Creating a proprietary PKI

Instead of relying on a external Certification Authority, a closed environment (laboratory, university, organization, enterprise...) could decide to build up its own and independent Public Key Infrastructure [JA00].[3] This involves the creation:

- of a Registration Authority that verifies the identity of the unit (person or machine) that wants to obtain a certificate, picks up its public key and transmits all these data to the CA;
- of a Certification Authority that creates the certificates, signs them, and sends a copy of them to the demander and another copy to the Publication Service;
- of a Publication Service that publishes the certificates and makes them available to anyone, by maintaining a LDAP (Light Directory Access Protocol) server or a Web server.

Of course, many questions need to be answered, many facts to be decided and many problems to be solved, following the needs:

- what kind of PKI to build (number of CAs, certificate policies...)
- to whom/what assign a certificate (researchers, students, workers, agents, groups, operational units, servers...)
- which kind of information are stored within the certificate

[2]Some VeriSign certificates shipped with Internet Explorer have expiration date in the year 2028.

[3]For instance, the CNRS has decided to set up its own Public Key Infrastructure on June 2000. See [CNRS00] for details.

- what is the validity of the certificate (validity time, expiration date)
- which standard to use for the certificates (X.509 or its own standard)
- which ciphers and key lengths to use for the certificate
- how to manage the possession of the private keys (backup, copies...)
- what policy is applied for the security and protection of the private keys
- how to manage the LDAP server
- how to manage the list of revoked certificates
- and so on...

Note again that the X.509 standard does not specify how to create or certify a Digital Certificate; the implementation is left open.

Furthermore, it is feasible to change the encryption algorithms embedded in the Digital Certificates system. The X.509 standard does not specify the ciphers that are to be used, even if it makes some recommendations; hence, it is possible to substitute them, or use the same with a longer key for improved security. Nowadays, the Certificate signature algorithms that are used by Mozilla are mostly SHA-1 (an improved version of SHA), MD5 and MD2 for hashing, and RSA for encryption. These algorithms at this time assure the best performances in terms of speed/security.

Part II

The Feige-Fiat-Shamir Identification Protocol

Chapter 6

Zero-knowledge protocols

6.1 Zero-knowledge interactive proofs

A *zero-knowledge interactive proof* is a protocol between two parties in which one party, called the *prover*, tries to prove a certain fact to the other party, called the *verifier*. A zero-knowledge interactive proof takes usually the form of a three-way protocol:

> **Witness:** The prover chooses a random number and sends to the verifier a proof of knowledge of this secret number. The number defines a class of questions to which the prover is supposed being able to answer.
>
> **Challenge:** The verifier chooses randomly a question in that class and sends it to the prover.
>
> **Response:** The prover answers the question to the verifier, using his secret.

If necessary, this protocol may be repeated for multiple rounds to reduce the probability that the prover matches by chance the correct answer; this is done until the requested level of security is reached.

Such a proof has the following properties: the verifier always accepts the proof if the fact is true (*completeness*) and he always rejects the proof if the fact is false (*soundness*).

The *zero-knowledge* property is the most interesting. In a zero-knowledge proof, the verifier learns nothing from the prover about the fact being proved that he could not already learn alone, except that it is correct. This is very useful as it resolves one of the biggest problems in Cryptography, that is: how the prover can

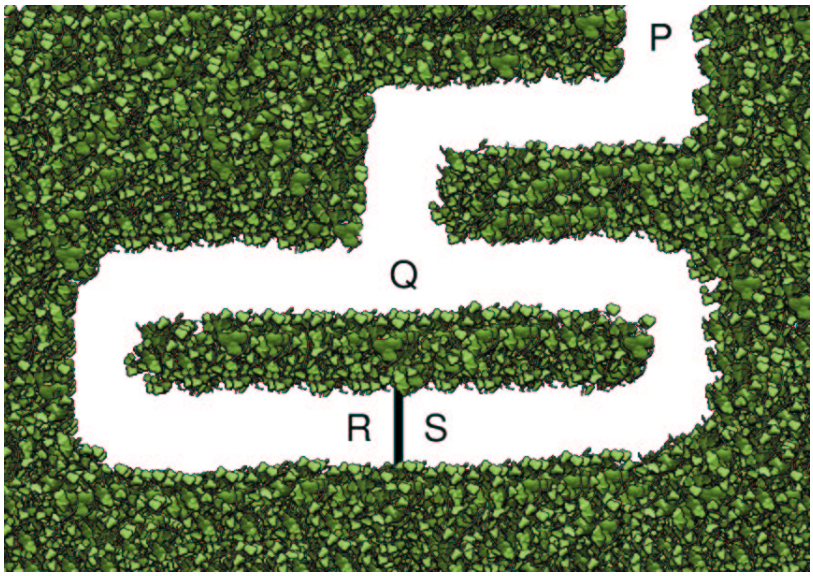

Figure 6.1: The Ali Baba's Cave.

prove that he knows a secret without actually disclosing it. In a zero-knowledge proof, the verifier cannot even prove the fact later to anyone else.

6.2 The Ali Baba's Cave

The zero-knowledge proof has been modeled by Quisquater and Guillou [QG89] in the example of the Ali Baba's Cave. This example may be explained as follows; see Figure 6.1.

From now on, I will use the mnemonic name of Peggy for the prover and the mnemonic name of Victor for the verifier, as in most cryptography literature.

Peggy (the prover) wants to prove to Victor (the verifier) that she knows the magic word that opens the portal at the points R-S, but she does not want to reveal that secret to Victor.

A round of the protocol takes place as follows: Victor goes to P and waits while Peggy goes unseen to R or S. Then Victor moves to Q and shouts to Peggy to come out either from the left side or the right side of the tunnel (*cut-and-choose* technique). If Peggy does not know the magic word, she has only a 50% proba-

bility to come out from the good side.

The protocol may be repeated as many times as wanted until Victor is certain that Peggy does know the magic word. If the protocol is repeated k times, the probability of Peggy to be cheating is 2^k. And it does not matter how many times the protocol is repeated, Victor does not learn the secret.

The main idea behind this reasoning is this: Peggy wants to prove a certain fact F_1 but she does not want to disclose the proof. She then finds another fact F_2, that may be publicly disclosed, such as F_2 is true if F_1 is true (necessary condition). So she "delegates" the proof of F_1 by proving actually F_2 only. In this example, F_1 is the knowledge of the magic word, and F_2 is the ability of appear from any of the sides of the tunnel. If Victor agrees that F_2 cannot be true without F_1 being true too, then the protocol may start.

Chapter 7

The Feige-Fiat-Shamir identification protocol

7.1 The Feige-Fiat-Shamir zero-knowledge proof of identity

Uriel Feige, Amos Fiat and Adi Shamir introduce in [FFS88] their identification scheme based on a zero-knowledge protocol. This is the best-known zero-knowledge proof of identity.

The Feige-Fiat-Shamir identification scheme uses a public-private key pair. It has the advantage of requiring only a few modular operations; hence, it is quite fast and may be implemented on the weak microprocessors embedded in smart cards.

FFS is a mere identification protocol; it may serve for login procedures. Unlike RSA, it is not possible to use it also for data encryption. However, its advantage over RSA is that it's much computationally lighter; FFS computations involve just multiplications, while RSA uses raisings to power.

In the scheme, a trusted center publish a modulus n which is the product of two large primes of the form $4r + 3$. n is a Blum integer, and hence it has the property that -1 is a quadratic nonresidue (i.e. $x^2 = -1 \bmod n$ has no solution) and its Jacobi symbol is 1 mod n. The difficulty of extracting square roots mod n makes the private key infeasible to guess.

Once the modulus n has been published, the center can be closed as it has no further role in the protocol.

First, Peggy generates her public and private keys as follows:

I. She chooses k random numbers $S_1, \ldots, S_k$ in Z_n. The $S = S_1, \ldots, S_k$ is the private key.

II. She chooses k random values I_j as $I_j = \pm 1/S_j^2 \bmod n$. The $I = I_1, \ldots, I_k$ is the public key.

The identification protocol develops itself as follows:

1. Peggy chooses a random number R, and sends $X = \pm R^2 \bmod n$ to Victor.
2. Victor chooses k random booleans $E_1, \ldots, E_k$ and sends them to Peggy.
3. Peggy sends the value $Y = R \cdot \prod_{E_j=1} S_j \bmod n$ to Victor.
4. Victor verifies that $X = \pm Y^2 \cdot \prod_{E_j=1} I_j \bmod n$. If and only if this is true, the proof is accepted.

To increase security, these steps may be repeated t times. The probability that Peggy can fool Victor is then 1 in 2^{kt}.

7.2 Alternate versions

An alternate version of the Feige-Fiat-Shamir identification scheme is given by Schneier [BS94]. In Schneier's version, Peggy chooses k random numbers $S_1, \ldots, S_k$ where S_j is a quadratic residue mod n, and publishes them as the public key. Then she calculates the smallest I_j such as $I_j = \sqrt{1/S_i} \bmod n$, and keeps $I_1, \ldots, I_k$ as the secret key. The following four steps of the protocol take place as explained above.

The disadvantage of this version of the protocol lies in the difficulty of computing quadratic residues.

7.3 Problems with zero-knowledge authentication protocols

The basic problem with this type of identification technique is that it is subject to *man-in-the-middle attacks*, in which a dishonest verifier Victor makes a copy of

the proof of identity given by Peggy, to misrepresent successfully him to another verifier Vincent. This is done by Victor relaying every single message from Peggy to Vincent and vice versa.[1]

The counterattack for this kind of attack is a strong synchronization: a certain time limit is imposed for the replies, in the purpose that there will not be enough time for relaying the communications. Another counterattack, which may be used in addition to the first, is to require all identifications to take place inside protected zones (shielded rooms, Faraday cages) to prevent communication relay.[2]

[1] [BS94] gives an example of this kind of attack to show how you can defeat a chess grandmaster. Challenge both Gary Kasparov and Anatoly Karpov to a chess game, at the same time but in two different rooms, without telling neither of them about the other. Have Karpov play white and Kasparov play black. You record Karpov's move, go in the room where Kasparov is and repeat the same move on Kasparov's chessboard. Then wait for Kasparov's reply, record his move, walk to the room where Karpov is and repeat the move on the chessboard. And so on, until you defeat one of the grandmaster (or both games ends in a draw, in which case both of them are going to be very impressed of your skill anyway). In fact, Kasparov is playing against Karpov, and you act solely as a middleperson. You don't even need to know the rules of chess!

[2] In the chess grandmasters problem, this could be done by requiring that all moves are made immediately, and/or by locking from outside the room where you and one of the grandmasters are playing.

Chapter 8

Secure Shell

SSH, the Secure Shell, is a security application (not a shell!) which allows encrypted communications over a network. SSH has a client/server architecture; it covers both user authentication, data encryption and data integrity, by the use of public key cryptography.

SSH permits secure remote logins, file transfer and remote command execution. Its aim is to replace the traditional rlogin, rsh, rcp, ftp and telnet commands, which are unsafe because they send all passwords and other data in cleartext; these data are therefore subject to capture, should an attacker activate a packet sniffer on the network.

The first version of the Secure Shell protocol, SSH-1, was developed in 1995 by Tatu Ylönen who founded later SSH Communications Security, Ltd.[1], for the maintaining and the further development of the project. In 1998 the second version of the protocol, SSH-2, was released.

The two version are incompatible with each other: SSH-1 uses DES, Triple DES, IDEA and Blowfish ciphers for encryption and RSA for authentication; SSH-2 uses Triple DES, Blowfish, Twofish, Arcfour and CAST for encryption and DSA for authentication [A01].

SSH was born and is primarily used on UNIX machines, nevertheless many implementations of SSH exist for several platforms: for instance SSHOS2 for OS/2, NiftyTelnet SSH for Macintosh, AmigaSSH for Amiga, SSHDOS for MS-DOS, PuTTY for Windows, SSH for BeOS, and Top Gun SSH for PalmOS. Many of them are Open Source projects [BS01].

[1] `http://www.ssh.com`

8.1 Establishing a secure connection

The protocol that allows a client to establish an encrypted connection to a SSH server takes place as follows:

1. The client sends a connection request (usually on TCP port 22) to the server.
2. The client and server communicate each other the versions of the SSH protocol they support; if the client supports only an older version than the one the server declared, they disconnect.
3. The client and server switch to a packet-based protocol over the underlying TCP connection.
4. The server sends to the client:

   its host public key, for the identification of the host;

   the server key, which is a temporary (regenerated every hour) asymmetric key used to improve encryption security;

   eight random check bytes, used as a protection against IP spoofing attacks;

   the list of encryption, compression and authentication methods that the server supports.

   Until now, all communications are still carried in cleartext.
5. If the client accepts the host key as valid (see the next section), the connection continues.
   The client generates a random session key for a cipher supported both by the client and the server, and encrypts the session key twice, with the host public key and with the server key.[2] The client then sends to the server:

   the encrypted session key;

   the eight check bytes;

   a choice of algorithms from the list of methods sent in the previous step.

[2]Encrypting the session key a second time with the server key ensures that an enemy that might capture the host private key in the future, cannot decrypt recorded sessions (*perfect forward secrecy*). This because the server key is temporary and never stored on disk.

6. From now on, all communications are encrypted with the session key and the selected cipher.
   As authentication, the server sends a confirmation message. The secure connection is now established.

Immediately after, the client authenticates itself to the server, trying in the order the following methods until one succeeds: Kerberos, Rhosts, RhostsRSA, public key, or password.

8.2 The host public key

As said before, in step 4 of the protocol the server tells to the client its host public key, just at connection time. There is not any public directory listing the public key for a particular host: SSH does not use Digital Certificates, nor any type of Certification Authority.

When the client receives the host key, it checks if it has ever communicated to that server before. This is done by consulting its database, which lists all known hosts with their public keys.

If the host name is not in the database, the user is prompted whether he wants to add it with its key, continuing the protocol, or break the connection.

If the host name is already in the database and the public key presented by the server matches the corresponding public key in the database, the protocol continues.

If the host name is in the database but the corresponding public key does not match, this may mean two things: either the host has legally changed its host public key, or the host address has been spoofed and the connection is under a man-in-the-middle attack.[3] The user is prompted whether he wants to update the key entry in the database with the new key, continuing the protocol, or reject the new key and break the connection.

[3] In the computer-network version of this attack, an enemy is sitting between the client and the server, relaying their communications. He is masquerading as server (with a spoofed host address and its own public key) with the client, and as client with the server. The data from the client are intercepted, decrypted and recorded, and then sent to the true server after being encrypted again; the same goes for the data from the true server to the client. Neither the client nor the server notice the presence of the intruder lurking between them.

```
zamok.crans.org 1024 35 140711467759341474750902704506473123403245100857699295877
602606848847059638973856005166694031558052334217088337121577128188223484923334764
513390452025518235208448987584688222938591491797488139077897547355638547623116665
243920022218479522299780093062519443684662062250875447196332810591450918439748251
547116479
yoko.rip.ens-cachan.fr 1024 41 156643291889081973677001060888346254299427434522l5
622372720016934458073511973537159285946050649562183541836791367598456497159279390
856304524274103409113414830315697862344011647707737124619247685043611677890951547
323342444535115187517557012014194258006599095045523194105738237788718597835250086
6899277946867993
crete.polytechnique.fr 1024 37 137013616904785131675077691846366226450158886027O8
319980706482785673864021633100092976151937714914250057653595033879423592243706026
362801712797975885799722390885449055492142134431088166120833997092537541923920473
502664299089969704261991743773580705060813670725911179853872990614476514743675334
8869717394254589
lxplus.cern.ch 1024 35 132405367977312799613369783536498990590680141744818722l333
222302192028703895347517759585199323001097670988414163821547670239455248852765194
391976132152372535421484579024834485470709758452548392697901737883658010649404104
139210073901958168230236684307545695498816570524071482094114090628404541162427376
88223581
```

Figure 8.1: An example of SSH's known hosts file.

In SSH-1, the public keys are of RSA type: each entry of the database contains the host address, key length, public exponent e and modulus n. (See Figure 8.1.) In UNIX systems using SSH-1, the database is stored partly in the global file `/etc/ssh_known_hosts`, and partly in the user's personal file `$HOME/.ssh/known_hosts`. SSH-2 uses DSA keys instead.

8.3 Security flaws in SSH

The policy of exchanging the host public key just "on demand" may produce security problems.

In fact, server authentication is one of SSH's weaknesses. There is no third party which may certify the possession of a certain public key by a certain host; users are supposed to trust the host at connection time.

When the user is alerted that the host key has changed, he cannot know if this is due to an host spoofing from an attacker, or not. Even worse, if an host spoofing is in progress the first time the user is connecting to a host with whom he has never spoke before, he will probably accept the (spoofed) host key without caring too much; the next time he will connect to the genuine host, he will notice the key change, and would possibly consider the new (genuine) host key as the fake one.

In the purpose of making connections to server more reliable, I have searched possible solutions offered by the FFS protocol.

Chapter 9

Implementation of the FFS scheme

To test properly the FFS protocol, I wrote a small implementation of it in C language, on a Linux machine. The source code is printed in Appendix A.

My program is based on the original paper [FFS88] by Feige, Fiat and Shamir. It uses a 1024-bit number for n and a default value of 5 for k. These are the values suggested in [BS94], and may be changed with little modifications to the source code. To keep the implementation simple, I let $t = 1$, that is the program makes only one round of the protocol.

As C's primitive types of data were too small to handle big cryptographic numbers, I have utilized the GNU Multiple Precision Arithmetic Library, by Torbjörn Granlund.

9.1 Implementation details

9.1.1 Architecture

This software implementation is composed by four modules.

`center.c` picks two 512 bits prime numbers and multiplies them to obtain the modulus n.

`zkp.c` simulates the communications between prover and verifier, and handles the whole protocol. Exchanged and shared data, like X, Y and the vector of E_js, are stored as global variables. It contains also the function which sets the seed for all random number generators.

`prover.c` chooses the k components of the public key and computes the corresponding components of the private key. It contains the functions for steps 1

and 3 of the protocol.

`verifier.c` provides the functions for steps 2 and 4 of the protocol.

9.1.2 Prime numbers

To find prime numbers I have used a GMP builtin function. It does some trial divisions followed by some Miller-Rabin probabilistic primality tests [TG02]. The probabilities of picking mistakenly a composite number instead of a prime are very small.

9.1.3 Randomness

An important problem in Computer Science is how to generate random numbers, needed in large amounts when working with cryptography [GS96]. The output of a computer program is predictable by definition; by definition, a random number is unpredictable. For this reason, to generate real random numbers, one should use external sources like radioactive decay measured by Geiger counters, cosmic rays, or thermal noise in electrical circuits. When this is not possible, there are algorithms that produce pseudo-random sequences. Although these sequences are always periodic and therefore predictable, when the period is very long the numbers in the sequences may be used acceptably by any application needing quite good quality randomness.

My program uses the pseudo-random number generator from the GMP library, initialized with a 1024 bit seed from the device `/dev/random`. This device provides an interface to the Linux kernel's random number generator, which gathers environmental noise from device drivers and other sources into a 512 bit sized entropy pool. From this entropy pool, random numbers are created.[1]

When the program freezes, it is because the entropy pool is empty. In this case it is necessary to generate more system events, by reading a directory or pressing some keys, or better by moving the mouse if the machine is running X Windows.

For use on slow machines, and for the purposes of testing the protocol, I have provided a "fast seed" option. If this option is enabled the seed, instead of being created from the data in the entropy pool, is calculated in a much faster way (and

[1] Linux Programmer's Manual, command `man -S 4 random`.

much lesser randomly!) as a combination of the machine time[2], the process ID of the program, and the process ID of the program's parent (usually the shell).

To enable the "fast seed" option, run the program with some argument in its command line. The generated pseudorandom seed is a very badly chosen random number, and may easily be cracked; do not use this option for real cryptographic applications.

9.1.4 Output

The program makes one round of the FFS protocol and prints on screen the results. Are printed, in order:

- the modulus
- the components of public and private keys
- the witness from prover
- the challenge from verifier
- the response from prover
- the verification value computed by the verifier
- the result of the authentification: successful if the verification value matches the witness.

Figure 9.1 shows a screenshot of the output.

9.2 Conclusions

FFS is an identification scheme based on public key cryptography; in this respect, it is similar to SSH's original authentication protocol, which uses RSA. Hence FFS does not solve the problems of key management; actually, the FFS protocol assumes that the public key are published somewhere by a trusted third party.

The original paper by Feige, Fiat and Shamir suggests a modification to the protocol. The trusted center checks properly the physical identity of Peggy (the

[2]This is the time since 00:00:00 UTC, January 1, 1970, measured in seconds.

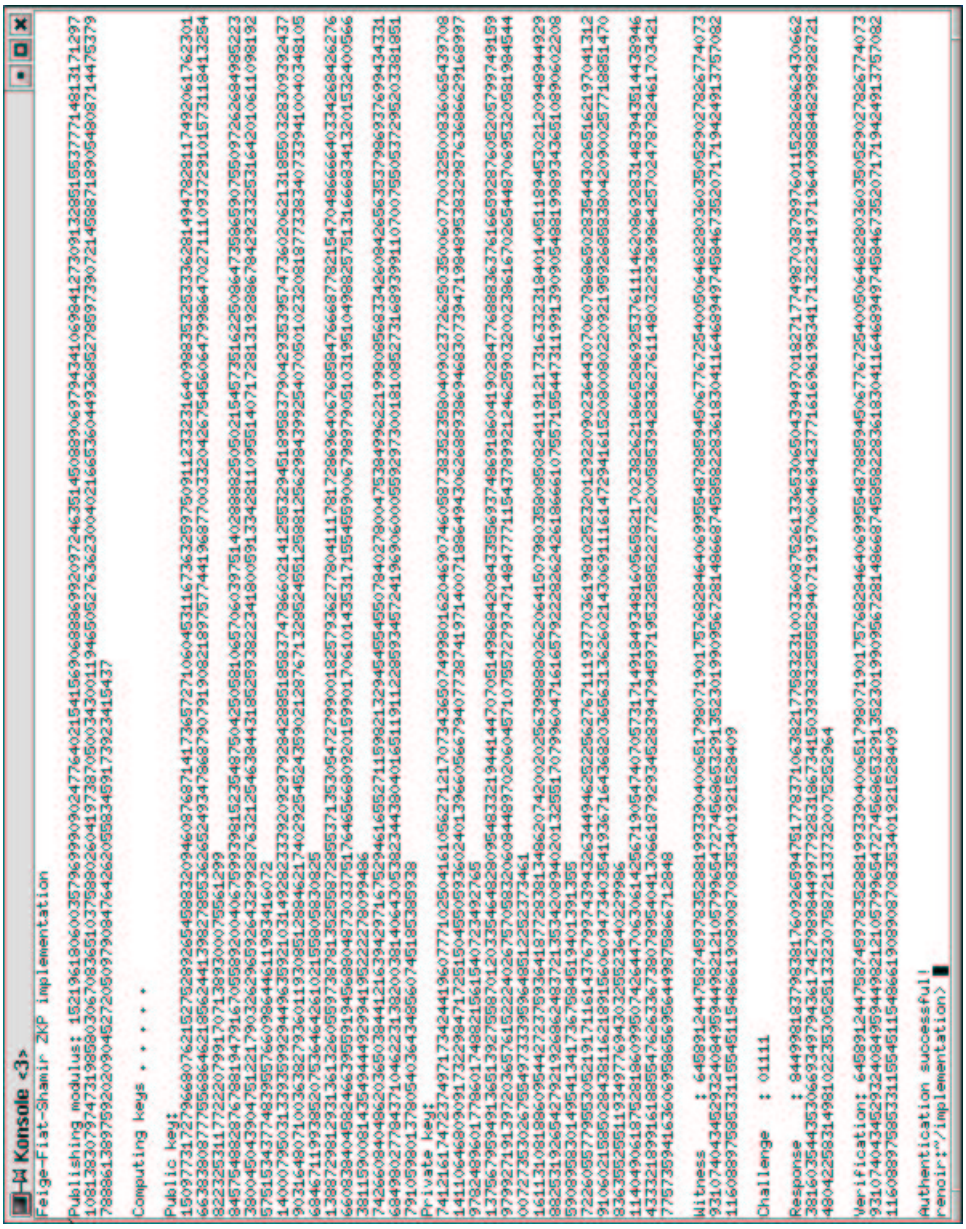

Figure 9.1: Snapshot of the program's output.

prover), and creates a string which contains the user's name, ID, and every other information necessary to identify her. The center also computes and stores the secret key $S_1, \ldots, S_k$ in a smart card, issued to Peggy. The smart card should be cracker-proof, in such a way that the content of its memory cannot be read or copied even by Peggy. Once that done, the center may be closed.

From the string, Victor (the verifier) can derive the public key $I_1, \ldots, I_k$ by applying a publicly known deterministic transformation $T(I, j)$. The identification protocol follows, as in the original scheme.
Even this version of the protocol nevertheless requires the presence of a trusted center issuing the smart cards.

It is then necessary to approach the problem from another point of view.

It would be feasible, for instance, to rely on trusted external servers ("Directory Servers") storing the public keys for multiple hosts. These servers could be organized hierarchically, like DNS[3], with every server operating on a subdomain of a network.

The first recording of the public key H_k of a host H would be done manually via a secure channel; it could be similar to a InterNIC registration, which is made to set up a new domain. During this recording, the Directory Server gives its own public key D_k to the host. In a second time, when the host wants to change its record (i.e. its public key H_k), he contacts the Directory Server and sends it securely the new data, encrypted with D_k. Therefore, nobody but the host who registered amongst the Directory may modify its public key H_k.
When an user wants to connect securely to host H, he retrieves directly H_k from the Directory Server. In the case of a host spoofing, the attacker cannot decrypt the data sent by the user because he does not know the corresponding private key.

The records in the Directory Servers could be basically of the same form of a SSH's known host file.

In an alternative version of this system, the Directory Servers store only a hash value of the public key, instead of the complete public key. This is done to improve transfer speed from the Directory Server to the user.

When the user connects to host H, the host sends to the user its public key H_k, as in the canonical SSH connection protocol. The user then computes the hash value of H_k and compares it with the corresponding one retrieved from the

[3]DNS (Domain Name System) is an Internet service which provides translation from hostnames to IP addresses; e.g. `lix.polytechnique.fr` refers to IP address `129.104.11.2`.

Directory Server. If they do not match, the user ends the connection.

Chapter 10

Other zero-knowledge implementations

Zero-knowledge protocols have already been used to build security applications; Stanford's SRP is one of these.

10.1 The Secure Remote Password protocol

The Secure Remote Password protocol [TW98] is the core technology behind the Stanford SRP Authentication Project[1], an Open Source initiative that integrates secure password authentication into existing network applications.

The SRP protocol is specifically an authentication protocol, that provides strong two-party mutual identification. It is based on asymmetric key exchange protocols coupled with zero-knowledge theory, but uses modular exponentiations while Feige-Fiat-Shamir uses multiplications. The SRP implementation provides both secure client and servers for telnet and FTP protocols [BS01].

10.2 Concepts under the SRP protocol

The SRP protocol is based on a construction called Asymmetric Key Exchange, which is a generalized form for a class of verifier-based protocols. AKE does not use encryption; instead, it uses predefined mathematical relationships to combine exchanged ephemeral values with established password parameters. This permits

[1] `http://srp.stanford.edu`

to keep the protocol simple, cipher-independent, safe, and suitable for use in those areas of world in which implementations of encryption algorithms are subject to legal restrictions.

In an AKE, each party computes a secret and then applies a one-way hash function to this secret to generate a verifier, which is shared with the other party. The verifier needs always to be kept secret to prevent *dictionary attacks*, but even if it is accessed by an enemy, it does not suffice to impersonate another user; the corresponding secret password is still needed.

10.3 How the SRP protocol works

The following shows how this protocol permits Peggy to authenticate to Victor.

Beforehand, two values are agreed and known by both parties: n, a large prime number, and g, a primitive root (generator) of the finite field GF(n).

Peggy knows her plaintext password P. She picks a random salt[2] s, and computes $x = H(s, P)$ and the password verifier $v = g^x \bmod n$, where $H()$ is a one-way hash function. Victor stores v and s.

The protocol then takes place as follows:

1. Peggy sends to Victor her username.
2. Victor looks Peggy's password entry and retrieves her corresponding v and s, sending s to Peggy.
3. Peggy computes her long-term private key $x = H(s, P)$. She chooses a random number a in Z_n, $a > 1$, computes her ephemeral public key $A = g^a \bmod n$ and sends it to Victor.
4. Victor chooses a random number b in Z_n, $b > 1$, computes his ephemeral public key $B = v + g^b \bmod n$ and another random number u, and sends them to Peggy.
5. Peggy computes $S = (B - g^x)^{a+ux} \bmod n$ and Victor computes $S = (A \cdot v^u)^b \bmod n$. If Peggy's password in step 2 matches the password she originally used to generate v, then the two common values S will match.

[2] A *salt* is a small random value added during encryption functions to improve security; the UNIX passwords system uses salt, too.

6. Peggy and Victor create both a session key $K = H(S)$.

7. Peggy sends $M_1 = H(A, B, K)$ to Victor as evidence that she has the correct session key K. Victor computes M_1 by himself and verifies that it matches Peggy's value.

8. Victor sends $M_2 = H(A, M_1, K)$ to Peggy as evidence that he too has the correct session key K. Peggy computes M_2 by herself and verifies that it matches Victor's value.

Once all protocol steps are completed, Peggy and Victor use K to encrypt subsequent messages.

It is worth noting that only the password verifier is stored, not the plaintext password itself; this means that in case of the verifier being revealed to an attacker, the system is not directly compromised as it would be if the password itself is revealed. And a legitimate authentication attempt on a compromised system does not disclose the password to the attacker. This design makes the protocol resistant to dictionary attacks, allowing even weak passwords to be used safely.

Mutual authentication may be obtained by requiring Victor to keep Peggy's verifier v secret.

Part III

Appendices

Appendix A

FFS source code

zkp.h

```
#define TRUE 1
#define FALSE 0

/* Multiplicity of challenge */
#define K 5
/* Number of rounds of the protocol */
#define T 4


mpz_t n;            /* modulus (a Blum integer) */
mpz_t i[K];         /* Prover's public key */
mpz_t rndseed;


void setrndseed();
void publish_modulus();
void compute_keys();
```

zkp.c

```
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include <time.h>
#include "zkp.h"

/*************************************************************************
  Feige-Fiat-Shamir Zero-Knowledge Identification Scheme: an implementation

          by Daniele Raffo, 25 JUN 2002 - LIX, Ecole Polytechnique
 *************************************************************************/
```

```
int fastseed = FALSE;



int main(int argc, char **argv) {
  mpz_t x;            /* witness */
  unsigned int e;     /* random boolean vector (challenge) */
  mpz_t y;            /* response */
  int proof;

  mpz_init(x);
  mpz_init(y);
  mpz_init(rndseed);

  printf("Feige-Fiat-Shamir ZKP implementation\n");
  if (argc > 1) {
    fastseed = TRUE;
    printf("Warning: fastseed enabled, using a bad random seed value!\n");
  }
  printf("\n");

                                /*        How the protocol works:        */

  publish_modulus();            /* Center publishes modulus n            */
  compute_keys();               /* Prover chooses public/private keys    */
  witness(x);                   /* Prover sends the witness              */
  e = challenge();              /* Verifier sends the challenge          */
  response(y, e);               /* Prover sends the response             */
  proof = verify(x, y, e);      /* Verifier verifies if response matches */

  if (proof)
    printf("Authentication successful!\n");
  else
    printf("Authentication failed!\n");

  mpz_clear(x);
  mpz_clear(y);
  mpz_clear(rndseed);
  return (0);
}



/*
  Set the random seed from /dev/random
*/
void setrndseed() {

  FILE *rnd;
  mpz_t rndtmp;
  unsigned long int idx;
  time_t t1;

  if (!fastseed) {
    mpz_init(rndtmp);
```

```
    rnd = fopen("/dev/random", "r");

    for (idx = 0; idx < 128; idx++) {
      mpz_set_ui(rndtmp, (unsigned long int) getc(rnd));
      mpz_mul_2exp(rndtmp, rndtmp, idx * 8);     /* left shift */
      mpz_add(rndseed, rndseed, rndtmp);
    }

    fclose(rnd);
    mpz_clear(rndtmp);
  }

  else {
    /* Set a faster seed. Do not use this for cryptographic purposes! */
    mpz_set_ui(rndseed, (unsigned long int) time(&t1));
    mpz_mul_ui(rndseed, rndseed, (unsigned long int) getpid());
    mpz_mul_ui(rndseed, rndseed, (unsigned long int) getppid());
  }
}
```

center.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <gmp.h>
#include "zkp.h"



/*
  Publish the modulus (a Blum integer which prime factors
  are randomly chosen and 512 bits long)
*/
void publish_modulus() {
  mpz_t rand, tmpprime, tmp, prime1, prime2;
  gmp_randstate_t state;

  mpz_init(rand);
  mpz_init(tmpprime);
  mpz_init(tmp);
  mpz_init(prime1);
  mpz_init(prime2);
  mpz_init(n);

  gmp_randinit_lc_2exp_size(state, 128);

  /* computes 1st prime */
  setrndseed();
  gmp_randseed(state, rndseed);
  mpz_rrandomb(rand, state, 512);
  while (TRUE) {                          /* repeat until prime is of form 4r+3 */
    mpz_nextprime(tmpprime, rand);
    mpz_sub_ui(tmp, tmpprime, 3);
```

```
    if (mpz_divisible_ui_p(tmp, 4)) break;
    mpz_set(rand, tmpprime);
  }
  mpz_set(prime1, tmpprime);

  /* computes 2nd prime */
  setrndseed();
  gmp_randseed(state, rndseed);
  mpz_rrandomb(rand, state, 512);
  while (TRUE) {
    mpz_nextprime(tmpprime, rand);
    mpz_sub_ui(tmp, tmpprime, 3);
    if (mpz_divisible_ui_p(tmp, 4)) break;
    mpz_set(rand, tmpprime);
  }
  mpz_set(prime2, tmpprime);

  /* computes modulus */
  mpz_mul(n, prime1, prime2);
  gmp_printf("Publishing modulus: %Zd\n\n", n);

  mpz_clear(rand);
  mpz_clear(tmpprime);
  mpz_clear(tmp);
  mpz_clear(prime1);
  mpz_clear(prime2);
  gmp_randclear(state);
}
```

prover.c

```
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include "zkp.h"

static mpz_t s[K];   /* private key */
static mpz_t r;      /* random number */



/*
  Choose private and public key
*/
void compute_keys() {
  int index = 0, index2, flag;
  mpz_t candidate, inverse;
  gmp_randstate_t state;

  printf("Computing keys ");

  gmp_randinit_lc_2exp_size(state, 128);
  setrndseed();
  gmp_randseed(state, rndseed);
```

```
  mpz_init(candidate);
  mpz_init(inverse);

  while (index < K) {
    printf(". ");
    mpz_init(i[index]);
    mpz_init(s[index]);

    mpz_urandomm(candidate, state, n);

    /* test if candidate has already been chosen as key component */
    flag = FALSE;
    for (index2 = index - 1; index2 >= 0; index2--) {
      if (mpz_cmp(s[index2], candidate) == 0) {
        flag = TRUE;
        break;
      }
    }
    if (flag == TRUE) continue;

    mpz_mul(inverse, candidate, candidate);
    mpz_mod(inverse, inverse, n);
    if (mpz_invert(inverse, inverse, n) == 0) continue;

    mpz_set(s[index], candidate);
    mpz_set(i[index], inverse);

    index++;
  }

  printf("\n\nPublic key:\n");
  for (index = 0; index < K; index++) gmp_printf("%Zd\n", i[index]);
  printf("Private key:\n");
  for (index = 0; index < K; index++) gmp_printf("%Zd\n", s[index]);

  mpz_clear(candidate);
  mpz_clear(inverse);
  gmp_randclear(state);
}



/*
  Pick a random number and send the witness x (step 1 of the protocol)
*/
void witness(mpz_t x) {
  gmp_randstate_t state;

  mpz_init(r);
  mpz_init(x);

  gmp_randinit_lc_2exp_size(state, 128);
  setrndseed();
  gmp_randseed(state, rndseed);

  mpz_urandomm(r, state, n);
  mpz_mul(x, r, r);
```

```
  mpz_mod(x, x, n);
  gmp_printf("\nWitness     : %Zd\n\n", x);

  gmp_randclear(state);
}



/*
  Send the response y (step 3 of the protocol)
 */
void response(mpz_t y, unsigned int e) {
  int index;

  mpz_set(y, r);

  for (index = 0; index < K; index++) {
    if (e & (0x1 << index)) mpz_mul(y, y, s[index]);
  }
  mpz_mod(y, y, n);

  gmp_printf("Response    : %Zd\n\n", y);
}
```

verifier.c

```
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include "zkp.h"



/*
  Send a random bit vector as the challenge (step 2 of the protocol)
*/
unsigned int challenge() {
  int index, bit;
  unsigned int bitmask = 0x0;

  setrndseed();
  srandom((unsigned int) mpz_get_ui(rndseed));

  printf("Challenge   : ");
  for (index = 0; index < K; index++) {
    bit = (int) (random() % 2);
    if (bit) bitmask |= (0x1 << index);
    /* challenge is printed starting from LSB */
    printf("%d", bit);
  }
  printf("\n\n");

  return (bitmask);
}
```

```
/*
  Verify the response from Prover (step 4 of the protocol)
*/
int verify(mpz_t x, mpz_t y, int e) {
  int index, result = FALSE;
  mpz_t test;

  mpz_init(test);

  mpz_mul(test, y, y);
  for (index = 0; index < K; index++) {
    if (e & (0x1 << index)) mpz_mul(test, test, i[index]);
  }
  mpz_mod(test, test, n);

  gmp_printf("Verification: %Zd\n\n", test);
  if (mpz_cmp(x, test) == 0) result = TRUE;
  mpz_clear(test);

  return (result);
}
```

Appendix B

Bibliography and references

[A01] Anonymous, *Maximum Security*, Sams, 2001

[AN95] Ross Anderson and Roger Needham, "Robustness principles for public key protocols", *Advances in Cryptology – CRYPTO '95*, Vol. 963 of *Lecture Notes in Computer Science*, 27-31 August 1995, Springer-Verlag, 1995

[BS01] Daniel Barrett and Richard Silverman, *SSH, The Secure Shell*, O'Reilly, 2001

[BS94] Bruce Schneier, *Applied Cryptography*, John Wiley & Sons, 1994

[CNRS00] Centre National de la Recherche Scientifique – Bulletin Officiel, 000381BPC, 13 June 2000
`http://www.dsi.cnrs.fr/bo/2000/08-09-00/`→
`416-bo080900-dec000381bpc.htm`

[ESR99] Eric S. Raymond, *The Cathedral and the Bazaar*, O'Reilly, 1999
`http://www.tuxedo.org/~esr/writings/cathedral-bazaar/`

[FFS88] Uriel Feige, Amos Fiat and Adi Shamir, "Zero-Knowledge Proofs of Identity", *Journal of Cryptology*, Volume 1 Number 2 1988, Springer-Verlag, 1988

[GS02] Simson Garfinkel and Gene Spafford, *Web Security, Privacy & Com-*

merce, O'Reilly, 2002

[GS96] Simson Garfinkel and Gene Spafford, *Practical UNIX & Internet Security*, O'Reilly, 1996

[HB01] Jonathan Held and John Bowers, *Securing E-Business Applications and Communications*, Auerbach, 2001

[IM90] Colin I'Anson and Chris Mitchell, "Security Defects in CCITT Recommendation X.509 – The Directory Authentication Framework", *Computer Communication Review*, April 1990

[JA00] Jean-Luc Archimbaud, *Certificats (électroniques): Pourquoi? Comment?*, CNRS/UREC, 2000
`http://www.urec.cnrs.fr/securite/articles/`→
`certificats.kezako.pdf`

[QG89] Quisquater and Gillou, "How to Explain Zero-Knowledge Protocols to Your Children", *Advances in Cryptology – CRYPTO '89*, Vol. 435 of *Lecture Notes in Computer Science*, 20-24 August 1989, Springer-Verlag, 1990

[SS99] Simon Singh, *The Code Book*, Fourth Estate, 1999

[ST00] Stephen Thomas, *SSL and TLS Essentials*, John Wiley & Sons, 2000

[TG02] Torbjörn Granlund, *The GNU Multiple Precision Arithmetic Library*, Edition 4.0.1, Free Software Foundation, 20 January 2002

[TW98] Thomas Wu, "The Secure Remote Password Protocol", in *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, March 1998
`ftp://srp.stanford.edu/pub/srp/srp.ps`