Yet Another Attempt in User Authentication

Liberios Vokorokos, Adrián Pekár, Norbert Ádám

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovakia
liberios.vokorokos@tuke.sk, adrian.pekar@tuke.sk, norbert.adam@tuke.sk

Peter Darányi

AT&T Global Network Services, Einsteinova 24, 851 01 Bratislava, Slovakia
peter.daranyi@tuke.sk

Abstract: This paper deals with the Encrypted Key Exchange (EKE) authentication method, which offers an opportunity to log on to a server and to authenticate the user itself without using a certificate or any direct transmission of the password. This makes the method an interesting alternative solution, where at the end of the authentication process generated is a key that can be further used for the needs of symmetric encryption. In the paper, an implementation of a client-server application that uses EKE authentication method is described. This application after a successful login process enables the user to transfer files in encrypted form, while the encryption key is generated at the end of the authentication process.

Keywords: authentication; Encrypted Key Exchange; Secure Password Exponential Key Exchange; shared key (secret); password; symmetric key

1 Introduction

The expert community has dealt with the issue of user authentication during login on to a server since the inception of the Internet [21]. The main goal is safe authentication using some of the available methods, such as a password or certificate. These methods must be durable to attacks whose purpose is to obtain passwords or other sensitive data. With this information (user name, password, other data, etc.) the attacker could pretend to be an authorized user and log in on a server. This paper describes one of the newer methods, the Encrypted Key Exchange [2], which combines symmetric and asymmetric cryptography. This fact

led us to implement this method in an application allowing the encrypted transmission of files. At present, this method is not commonly available in any commercial application, which also contributed to the attraction of this topic.

The first chapter contains a brief description of authentication protocols (methods) using a password and their comparison. The second chapter includes a draft of the implemented protocol. In the third chapter the implementation of the program with a justification for the used implementation techniques is described. The last chapter is devoted to the verification of outputs and to evaluation of solutions.

2 Authentication Protocols

Cryptographic authentication often relies on ownership of the key authenticated by a party. Such a key usually has a length from approximately hundred bits to several thousand bits, depending on the used algorithm and the desired security level. Experience showed that people have difficulties remembering passwords having seven or eight characters. When all uppercase and lowercase letters and digits from *0* to *9* are used, a random eight-character password represents less than *48* bits of randomness. Therefore, we can conclude that even a short random key for cryptographic algorithms cannot be reliably memorized by people. Cryptographic keys are frequently stored in secure memory in computers or special equipment, such as cryptographic servers or smart cards. However, there are situations where this form of custody is inconvenient or expensive [12], [18]. For this reason, the capability to establish a secure connection that relies on short passwords easily memorized by people is desired.

3 Comparison of Selected Authentication Methods

Protocols for the creation of the key are designed to be safe in situations where participants share their password with only small entropy. At first glance it may seem impossible to achieve a key using only a short password that would not be vulnerable to brute force attack (a progressive scan of all the possibilities) to find the password. This is probably the reason why the first protocols based on passwords (password-based protocols) appeared only in *1989*. Such first protocols used the additional assumption that the client knows the server's public key without sharing password with the server. Later, Bellovin and Merritt presented a class of protocols that had this additional assumption implemented [1].

The idea of Bellovin and Merritt, the Encrypted Key Exchange (EKE) protocol [1], is that the initiator of the protocol chooses a single-shot public key and uses a shared password to encrypt the key. The respondent decrypts the public key and uses it to send a session (relational) key safely back to the initiator. Assuming that

the public keys are always random strings, an attacker who is trying to sequentially test all the passwords cannot distinguish which single-shot public key was used. Further, even if the correct public key is found, it cannot be used to discover the session (relational) key because from the public key it is impossible to obtain the private key. There are many variations of the Bellovin Merritt EKE protocol as well as many alternative protocols [2], [14], [16]. Recently, the protocol was also expanded with proven security features. The original EKE protocol does not specify the used encryption algorithm, which is how to convert the password into the required key.

3.1 The Original Bellovin and Merritt's EKE Protocol

EKE is closely related to the Diffie-Hellman key agreement, and its basic idea is to transfer transient public keys, which were encrypted by a password as a shared key. Only parties knowing the password are able to complete the transfer. Parameters π (shared password) and L (security parameter) are the shared information. As in the basic Diffie-Hellman [4] exchange, the shared key (secret) is $Z_{AB} = g^{r_A r_B}$, but the algorithm for obtaining the session key K_{AB} from Z_{AB} is not specified. The protocol requires two exponentiations by both parties, which is the same as in case of the basic Diffie-Hellman [4] key exchange.

In this exchange, as described in Protocol 1, first both sides agree on large prime numbers, *p*, *n*, and an element, g *(*$2 \leq g \leq p$ *- 2)*, that generates a subgroup of large order (public parameters). *A* chooses a random number, r_A (public key), generates $g^{r_A} = t_A$, encrypts it with π and sends to *B*. Sharing the password, π, *B* decrypts the message to obtain the shared key (secret), generates $g^{r_B} = t_B$, also generates another random number, $n_B \in_R \{1, \dots, 2_L\}$, for the session key, encrypts them and sends to *A*.

A		*B*
$r_A \in_R \mathbb{Z}_p$ $t_A = g^{r_A}$	$\xrightarrow{A,\{t_A\}_\pi}$	$r_B \in_R \mathbb{Z}_p$ $t_B = g^{r_B}$ $Z_{AB} = t_A^{r_B}$ $n_B \in_R \{1, \dots, 2_L\}$
$Z_{AB} = t_B^{r_A}$ $n_A \in_R \{1, \dots, 2_L\}$	$\xleftarrow{\{t_B\}_\pi,\{n_B\}_{K_{AB}}}$	
$verify\ n_A$	$\xrightarrow{\{n_A,n_B\}_{K_{AB}}}$ $\xleftarrow{\{n_A\}_{K_{AB}}}$	$verify\ n_B$

Protocol 1
The original Bellovin-Merritt EKE Protocol

A decrypts the message to obtain the shared key (secret), also generates a random number for the session key and sends it to *B*. Assuming the final verification is successful, both parties can calculate the true session key that will be used for all future messages between *A* and *B*. [2], [14], [16]

3.2 The Secure Password Exponential Key Exchange Protocol (SPEKE)

The SPEKE protocol was designed by Jablon [7], [8]. Although SPEKE, like EKE, is based on the Diffie-Hellman exchange, the main difference between them is in the password used to determine the base that is used in Diffie-Hellman exchange [4]. Jablon introduced the basic and extended version of the SPEKE protocol.

Prime *p* and group *G* are chosen in a way that $q = \frac{(p-1)}{2}$ will become a prime, and *G* will have a degree of *q*. Password, π,is considered to be a number from $\mathbb{Z} *_p$ and then $P = \pi^2$ will surely lie in group *G* and has a degree of *q* (assuming that π is not equal to *1*, *-1* or *0*). Value *P* is used as a generator of group *G*. [2]

Except this special way of defining the generator of the group, the protocol is exactly the same as the basic Diffie-Hellman key exchange with key confirmation [4]. The shared key (secret) is therefore $Z_{AB} = P^{r_A r_B}$.

The original version of SPEKE has no evidence of safety, but later MacKenzie [11] introduced the proof of a slightly modified version. These changes are:

- *P* is defined to include the identities of *A* and *B*: $P = (H(A;B;\pi))^2$
- The hashes used to form the session key include the identities of *A* and *B*, the exchanged messages, $t_A$and t_B, the password, π, as well as the shared key (secret), Z_{AB}: $K_{AB} = (A;B;t_A;t_B;\pi;Z_{AB})$
- The exponents are chosen randomly: $r_A;\ r_B \in_R \mathbb{Z}_q$

The expanded version of the protocol, with the V_a and V_b verifiers defined in the version proposed by MacKenzie [11], is described in Protocol 2.

The SPEKE protocol consists of two stages. The first one uses Diffie-Hellman [4] to establish the shared secret (key) Z_{AB}. Where SPEKE differs from DHEKE is that, instead of the commonly used fixed primitive base g, it converts with function *P* the password π into a base for exponentiation. The rest of the first stage is pure Diffie-Hellman, where parties *A* and *B* start out by choosing two random numbers, r_A and r_B. *A* computes by function *P* message t_A and sends it to *B*. *B* computes by function *P* message t_B and sends it to *A*. *B* also computes the shared key Z_{AB}. *A* also computes the shared key Z_{AB}. [7]

In the second stage, both *A* and *B* confirm the knowledge of Z_{AB} before using it as session key K_{AB}. *B* sends with message t_B a proof of Z_{AB} (message V_B) – which is obtained by a strong one-way hash function H_1 – to *A*. *A* verifies that V_B is correct

and sends its proof of Z_{AB} (message V_A) – which is obtained by a strong one-way hash function H_2 – to *B*. *B* verifies V_A is correct. Assuming the verifications are successful, the protocol is complete. [7]

A		*B*
$r_A \in_R \{1, \dots, 2_L\}$ $t_A = P^{r_A}$	$\xrightarrow{t_A}$	
		$r_B \in_R \{1, \dots, 2_L\}$ $Z_{AB} = t_A^{r_B}$ $check\ Z_{AB} \notin \{-1, 0, 1\}$ $t_B = P^{r_B}$ $V_B = H_1(t_A; t_B; Z_{AB}; \pi)$
	$\xleftarrow{t_B, V_B}$	
$Z_{AB} = t_B^{r_A}$ $check\ Z_{AB} \notin \{-1, 0, 1\}$ $check\ V_B$ $V_A = H_2(t_A; t_B; Z_{AB}; \pi)$		
	$\xrightarrow{V_A}$	
		$check\ V_A$
$K_{AB} = H_3(Z_{AB})$		$K_{AB} = H_3(Z_{AB})$

Protocol 2

Secure Password Exponential Key Exchange Protocol

In this case, shared information are as follows: subgroup *G* of group $\mathbb{Z} *_p$ degree $q = \frac{(p-1)}{2}$, where p and q are primes; derived password $P = \pi^2$, where π is interpreted as an element of $\mathbb{Z} *_p$; security parameter *L*.

For security reasons of the implemented version, and thus the application, we decided to implement the above-mentioned version of the SPEKE protocol.

4 Draft of the Implementation of Authentication Procedures

Based on [17], the SPEKE protocol was implemented in a program for file transmission. In this way, both possibilities offered by the protocol – authentication and password generation, which will be subsequently used to encrypt the transmitted files – will be used. The application is divided into two parts: Into the Secure File Share (SFS) Server and Client programs.

SFS Server is a server program and SFS Client is a client program in the client-server architecture. After start the server loads the basic settings from the settings file (if it exists), which is saved in the directory where the program runs. Otherwise, an error message appears. Next, the servers search the database for data on previously added users. Then the servers begin to listen on a specified port for user log in requests. Like the server, after the client starts, it loads the basic settings from a settings file (if it exists), and then it tries to connect to the specified IP addresses and port. After successful connection establishment and client login, on the basis of the exchanged messages both parties generate a shared key. From that moment on, until an explicit or automatic logging off, the client can transfer files located in the shared directory of the server. Files transferred over the network are encrypted.

For the implementation of cryptographic protocols, the Java programming language was chosen as it supports very large numbers (several hundred bits), hash functions and cryptographic protocols. In addition, for the widest possible usage of the resulting program, Java is not platform (PC, Macintosh, etc.) or operating system (Microsoft Windows, Linux, Mac OS X, etc.) dependent. Since the protocol is implemented for the users logging in to the server, for the user management a relational database is used, with which, on the basis of the above-mentioned criteria, Java can communicate [15]. The database server is located on the same computer as the SFS Server. The database contains only one single table with fields of all necessary data about the users. For each user registered, there are an identification number, name, password, and a flag indicating whether the user has access to the server. Field ID is the primary key table. To avoid adding more records with the same user, name entry is set as unique. For security reasons, the file transfer was only implemented from the server to the client, which will prevent a possible leakage of the key, followed by logging in of the attacker to the server and transfer potentially dangerous files to the server.

As the symmetric cryptography protocol used to encrypt the transmitted file, Advanced Encryption Standard (AES) protocol was used. AES encrypts by blocks in length of *128* bits and supports key in lengths of *128*, *192*, and *256* bits. After successful logging in using the SPEKE protocol, the client and server agree on the key that can be used for symmetric cryptography. In order to generate a sufficiently strong key and also to ensure security of the logging in to the server, primes of at least *500* bits will be used. The agreed key length has approximately the same length as the length of primes used in the protocol, which is certainly greater than the length of keys supported by AES protocol. For this reason the key for AES will be the first *128, 192,* or *256* bits from the agreed key. In the Protocol, a hash function is also used. In order to ensure the highest security, the SHA-*512* function will be used.

5 Communication Principles

The complete authentication process and activities performed by the individual sides are described in Figure 1. After the initial initialization, the application starts to listen on a port. If someone connects to this port, another thread starts. At the beginning, the tread waits until the connecting side sends its user name. Then it sequentially checks the following things to see whether the user can log in:

- First it checks whether the user is logged in with that user name.
  - If the user is logged in the application sends an error message.
  - If not, it checks if the user name is registered in the database.
- Next, if the user is not registered it sends an error message. If yes, the application checks whether the user has not been banned on the server. When the user has passed the final check he can proceed with logging in.

The next procedure of authentication and generation of a common symmetric key is in conformity to the described SPEKE protocol in the previous chapter. If a problem occurs during the client authentication, the server immediately terminates the connection.

After successful logging in, the server sends the structure of the shared folder to the client. Then, until the end of connection, the server waits for a request from the client to resend the folder structure or for file transfer. Connection can be terminated in three ways: the client logs out, the server terminates an inactive connection after a timeout, or disconnection occurs due to network failure.

Upon termination of the connection the thread ends. In order to facilitate the work, the application was extended with a graphical user interface. On the client side, all communication (logging in, reloading the shared directory tree of the server, file transfer) with the server is performed in separate threads. With this method it is possible to save or load client settings, which consist of an IP address, server port and a local directory in which the files are downloaded.

After successful login, the structure of the shared directory tree of the server is accepted. When the user wants to transfer a file, a separate thread will start and as parameters the data obtained during logging in process are used. That is the key by which the file transfer will be encrypted. In case the connection is not successful, the application displays an error message.

During file transmission, for each file the server first sends a message that indicates the client wants to transfer a file. Pieces of the file are always transmitted in a fixed size (by default *1024* B). Before file transmission, a check is performed as to whether the file exists in the destination directory. If so, the user chooses whether to overwrite it. The user can terminate file transfer at any time.

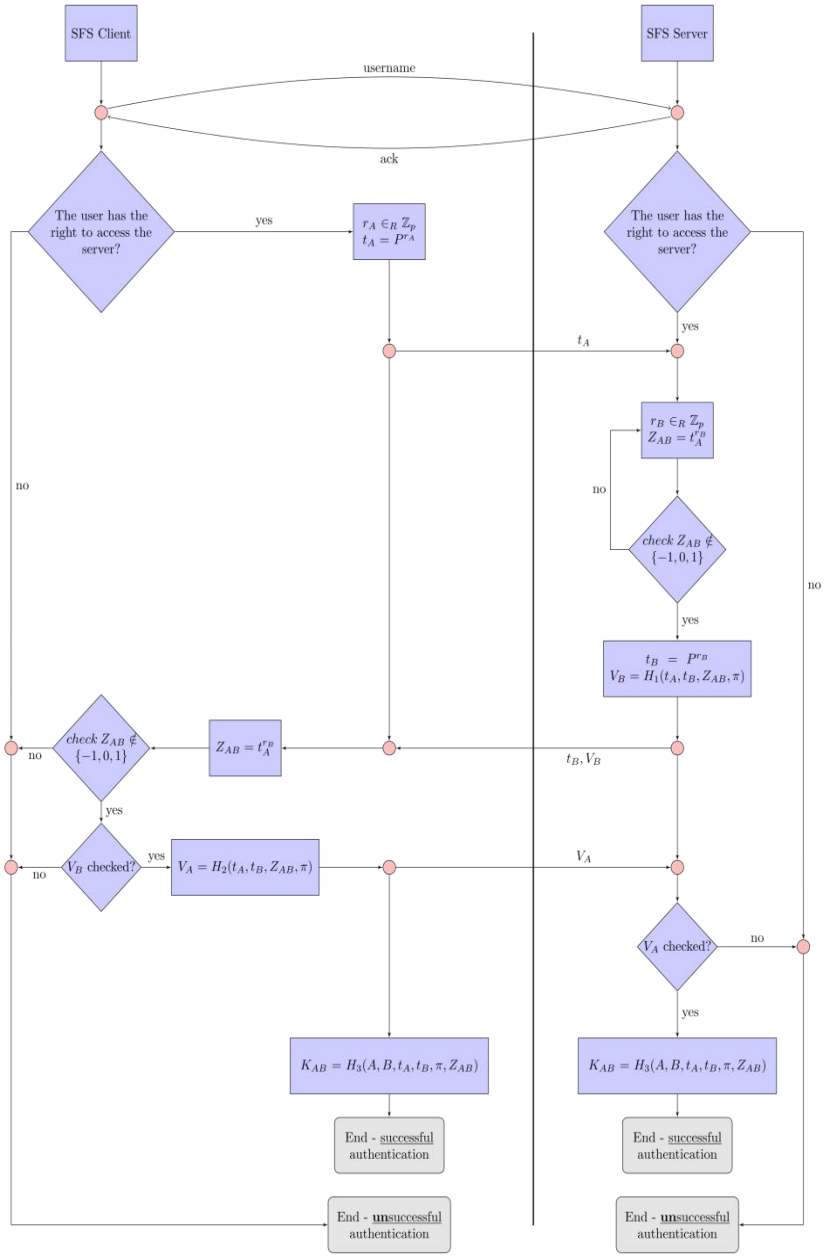

Figure 1
Authentication process

6 Implementation of the Authentication Procedures

Due to the requirements mentioned in the draft section of this paper, we decided to use the Java programming language. This language contains the Java Cryptographic Extension (JCE) extension, by which it is possible to use common cryptographic functions. The Java Virtual Machine (JVM) allows platform independence.

Complying with [22], we chose MySQL as the database server. However, the structure of the database tables is not difficult for administering except that for the command line is also possible to use GUI tools such as MySQL Administrator. Communication with the database server consists of several steps that define the JDBC process.

Communication with the client consists of several parts. First, the message exchange happens in conformity with the SPEKE protocol described in chapter two. During this exchange, the client authenticates itself and a symmetric cryptography key is also generated. After successful completion of this phase of the program until the end of the connection, incoming requests from the client are awaited. The client can request a file transfer or retransmission of the shared directory tree structure of the server. However, if the client does not send the request within a specific amount of time (*15* minutes by default), the connection is automatically terminated. During the process of authentication and symmetric key generation, the *BigInteger* variable type is used. The problem with the variables of the common Math class is that it works with *ints* and *doubles* types. These types can hold only finite numbers and have limited precision. The *BigInteger* class can hold arbitrarily large numbers.

At first, two primes, p and q, are required to satisfy the condition $q = \frac{(p-1)}{2}$. For safety reasons it was decided to use a prime length of *500* bits. Since sending unencrypted primes is a security risk and since the generation of two large primes that meet a given condition may last for an indefinite but certainly long time, these primes are statically assigned and are to be generated by generators of cryptographically strong pseudo-random numbers.

A cryptographically strong pseudo-random number corresponds at least with the test of the statistical random number generator specified in FIPS [6]. Furthermore, *SecureRandom* must give non-deterministic output that is a cryptographically strong sequence of numbers corresponding with the description in RFC 1750, Randomness Recommendations for Security [5]. Non-deterministic output that is a cryptographically strong sequence of numbers corresponds with the description in [5]. In the following, a probable prime number, p, is generated. Then, the variable q is saved as a number, which is the result of p subtracted by *1*. Then, it is divided by *2*. Next, a new number, q, is calculated. This will continue until primes p and q are generated, which satisfy the equality $q = \frac{(p-1)}{2}$.

During initialization it is necessary to establish communication channels through which the process of authentication can be performed. I/O in Java is based on the use of streams. Input data streams read data by bytes and write them by bytes on the output. All classes for working with streams are based on abstract classes *InputStream* and *OutputStream*. After creating the communication channels, user authentication will be performed, which was described in the previous chapter. The generation of the symmetric key $K_{AB} = (A; B; t_A; t_B; \pi; Z_{AB})$ consists of sub-processes that are specific to each number (variable) of the key. After authentication and generation of the symmetric key, an initial AES setup is done.

After successful authentication of the client and preparation of AES cipher, the server starts to wait for requests from the client. The following steps were described in the previous section. Before sending a file, first it is checked whether the file exists and if not the client is informed. In the next step, pieces of the transmitted file will be determined. Subsequently, the file size is sent to the client to know how many bytes to expect. Sending the file itself is performed in a cycle. After the end of file transfer, the input stream associated with the transferred file is closed. Finally, it is checked whether the number of sent bytes is equal to file size.

7 Verifying the Implemented Method

Verification of the solution was divided into three parts: verification of the functionality of the implementation of cryptographic functions; verification of the behaviour of the application during occurrence of errors; verification of overall functionality of both programs (SFS Server and Client). These verification procedures will be the subject of interest in the following sections.

The application was also subjected to performance tests, which showed no noticeable lags.

7.1 Verification of Functionality of the Implementation of Cryptographic Functions

Verifying the functionality of the implementation of the cryptographic protocols was performed in such a way that, instead of the generation of numbers, a message fingerprint fixed assigned numbers were used, which were chosen not to meet the conditions necessary to continue the process of authentication:

- change of properly generated number Z_{AB} to a number *-1*, *0*, or *1*;
- change of the used password *p*;
- change of the messages from which the fingerprints were generated;
- use of different symmetric keys and initialization vectors during the encryption;
- use of user name that was denied access.

Each test resulted in positive results, i.e. the programs behaved in accordance with expectations: as soon as the error was detected the authentication process and consequently the network connection was terminated.

7.2 Verification of Behavior of the Application during Occurrence of Errors

When verifying the behaviour of the programs during error occurrence, the below errors were intentionally created:

- At the start of file transfer, the transmitted file was not in the shared directory of the server;
- During transmission, the network connection was interrupted;
- The network connection was first interrupted for client inactivity, and then the client sent a request;
- Files containing the settings were not in the directory where the program was launched;
- Interrupting the connection between the SFS Server and the database.

7.3 Verification of the Overall Functionality of the Application

When verifying the overall functionality of both programs tested, we used the situations that occur in normal use of the program without errors. The test results were as follows:

- adding and dropping a user from the database, denying and guaranteeing the access, and changing the password of an existing user was always successful;
- user authorized to access was always able to log in;
- user without access was never able to log in and he was always announced about the reason of the failure;
- the file was always transmitted successfully;
- in the Graphical User Interface the state of the component always reflect the actual state of the program.

Conclusion and Future Work

In the fast developing world of the Internet, more and more data and services are being published [9]. The number of services where client authentication is required and secure transmission of data (such as eBanking) is needed are increasing [10]. For this reason, it is necessary to improve authentication and encryption mechanisms and increase their security [19], [3]. Most programs and services that support secure data transfer use the Secure Shell (SSH) or Secure Socket Layer (SSL) technologies. Programs SFS Server and SFS Client are using a new approach, authentication with agreement on the symmetric cryptography.

During the subsequent transmission of encrypted data, the Encrypted Key Exchange (EKE) – specifically the Secure Password Exponential Key Exchange (SPEKE) – variant is used.

The advantage of EKE over commonly used applications is the fact that EKE is less common, which means that the methods of attack on this scheme are not yet known, in contrast with the methods of attack on commonly used methods, which ensures less chance of a successful attack. This advantage, with extension of EKE, will certainly become more pronounced. Another advantage of EKE is the fact that certificates are not required to prove the credibility of the server or the client, which eliminates the need for certificate generation and update.

The disadvantage of EKE against SSL and SSH is its lower versatility. Any application using EKE scheme needs to directly implement it.

Our application was designed to implement the process of authentication using the Secure Password Exponential Key Exchange (SPEKE) scheme as a login mechanism on to a server with file sharing capability, which is transmitted encrypted. Applications successfully meet these requirements. During tests it was confirmed that the processes of authentication and encryption, as well as other parts of the programs, are fully functional. Both programs were developed with regard to portability to other platforms and operating systems.

Although the presented approach should seem unnecessary, actual research and development in the field of secure file downloading mechanisms are leading to the improvement of their abilities, security and flexibility. Our approach offers a prototype of a secure file download system that implements a narrowly spread encryption protocol. The fact that this protocol is not as widely spread makes the system more unique within its application domain.

In the future, the interoperability between the system and the users will be improved by the implementation of Web technologies. The advantage of this implementation to the users is that Web applications do not need any installations on the client side, so they can be run on any system with an appropriate and compatible browser. As mentioned before, in our approach, certificates are not required to prove the credibility of the server or the client. So Web technologies will make the proposed system more flexible. This will also result in the improvements in interoperability with existing systems and clients.

Future work should also be aimed at the enhancement of the security of stored passwords and reduction of the risk that an attacker would obtain the password. This could be done by storing only the password's fingerprint $P = (H(A;B;\pi))^2$ in the database, and not the entire password. Other functions by which the applications could be extended are, for example: the generation of a new key for each file transfer; more detailed information on the client side (file size); management of the shared folder (adding, removal, renaming of files) directly in SFS Server.

Another task in the future is to extend the application by the ability to recognize the count of a session for a given user. At present, more than one user can connect to the server with the same login username and password.

Acknowledgement

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10 (50%). This work is also the result of the project implementation Development of the Center of Information and Communication Technologies for Knowledge Systems (project number: 26220120030) supported by the Research & Development Operational Program funded by the ERDF (50%).

References

[1] Bellovin, S. M, Merritt, M.: Encrypted Key Exchange: Password-based Protocols Secure Against Dictionary Attacks. IEEE Symposium on Research in Security and Privacy, 1992, pp. 72-84

[2] Boyd, C., Mathuria, A.: Protocols for Authentication and Key Establishment. Springer, p. 300, 2003

[3] Fanfara, P., Danková, E., Dufala, M.: Usage of Asymmetric Encryption Algorithms to Enhance the Security of Sensitive Data in Secure Communication. SAMI 2012, Herľany, Slovakia, 2012, pp. 213-217

[4] Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory, 1976

[5] Eastlake, D., Crocker, S., Schiller, J.: Randomness Recommendations for Security. RFC 1750, 1994

[6] Federal Information Processing Standards Publication: Security Requirements for Cryptographic Modules. Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8900, 2001

[7] Jablon, D.: Strong Password-Only Authenticated Key Exchange, ACM Computer Communications Review, pp. 5-26, 1996

[8] Jablon, D.: Extended Password Protocols Immune to Dictionary Attack. Proceedings of the Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE '97) pp. 248-255, 1997

[9] Jakab, F. [et al.]: Rich Media Delivery. Computer Science and Technology Research Survey (CST '2008), Košice, 2008, pp. 31-36

[10] Kopják, J., Kovács, J.: Timed Cooperative Multitask for Tiny Real-Time-embedded Systems, IEEE 10th Jubilee International Symposium on Applied Machine Intelligence and Informatics (SAMI 2012) Herl'any, Slovakia, 2012, pp. 377-382

[11] Mackenzie, P.: On the Security of the SPEKE Password-authenticated Key Exchange Protocol. Cryptology ePrint Archive, Report 2001/057, 2001

[12] Madoš, B., Baláž, A.: Data Flow Graph Mapping Techniques of Computer Architecture with Data-driven Computation Model. SAMI 2011, Slovakia - Budapešť, 2011, pp. 355-359

[13] Michalko, M.: Video Streaming in Wireless Networks Using Avismo Concept, Journal of Information, Control and Management Systems, Vol. 9, No. 2, 2011, pp. 109-117

[14] Raymond, J., stiglic, A.: Security Issues in the Diffie-Hellman Key Agreement Protocol. IEEE Trans. on Information Theory, pp. 1-17, 2000

[15] Roman, S.: Access Database Design & Programming. O'Reilly Media, 3rd edition, 2002, p. 448

[16] Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. Second Edition, Wiley, p. 758, 1996

[17] Szabó, CS., Slodičák, V.: Software Engineering Tasks Instrumentation by Category Theory. SAMI 2011, Slovakia, 2011 pp. 195-199

[18] Tomasek, M.: Encoding Named Channels Communication by Behavioral Schemes. Acta Polytechnica Hungarica, Vol. 8, No. 2, ISSN 1785-8860, pp. 5-19, Budapest, 2011

[19] Vokorokos, L., Ádám, N., Baláž, A.: Application of Intrusion Detection Systems in Distributed Computer Systems and Dynamic Networks. CST 2008, 2008, pp. 19-24

[20] Vokorokos, L., Baláž, A., Madoš, B.: Intrusion Detection Architecture Utilizing Graphics Processors, Acta Informatica Pragensia, Vol. 1, No. 1, 2012, pp. 50-59

[21] Vokorokos, L., Kleinová, A., Látka, O.: Network Security on the Intrusion Detection System Level. INES 2006, 2006, pp. 270-275

[22] Widenius, M., Axmark, D., Dubois, P.: Mysql Reference Manual, O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002