

# Implementation of SPA

Yoan Martin

School of Computer and Communication Sciences

Semester Project

December 2017

**Responsible**  
Prof. Serge Vaudenay  
EPFL / LASEC

**Supervisor**  
Ms. Handan Kiliç  
EPFL / LASEC

# Chapter 1

## Introduction

Beyond users' lack of knowledge on computer science, the problems can occur because of unawareness of possible outcome of an action or complicated interfaces.

PEBCAK [1]: Problem exists between chair and keyboard. This quote is frequently used to explain that many problems come from the user. Unfortunately, users do not always have an advanced knowledge in computer science. ~~They need simple interfaces. However, computers are extremely complex and providing a simple way to use them is a real headache.~~ Passwords are the perfect example where simplicity meets complexity.

*why simple* } *It is not clear what you mean he is.*  
*why complex* } *Be more explicit.*

A common user would create an easy-to-remember password. For instance, he can use the name of his pet. *This solution is easy for the user, but not secure.* In addition, users usually generate passwords following the same behaviour: For instance, the research of the SURPASS novel [2] found that most of the uppercase letters appeared in the first position and that most of the symbols appears at the last position. Consequently, an attacker can use a dictionary

*Dictionary attack*

*Dictionary*

attack [2]. It consists of understanding this password generation behaviour to create a dictionary of common passwords. Then, the attacker tries these passwords until finding the good one. If the password is too short or too common, it would be broken easily. The attack is even faster if the attacker creates a dictionary containing element of the user's personal life.

*Unfortunately*

*What is not enough?*

*is it the problem?*

*what is the problem and so solution?*

Unfortunately, this is not enough, a common user logs into many accounts. The solution is simple: using the same password for each account. Again, this solution is easy but not secure.

Even if the password is strong, if the server, where the password is used, is compromised by an

adversary, then this adversary can access to the other accounts of the user. An attacker can

also find the user's password using phishing attack [3]. This attack consists of stealing the

identity of a trusted service. Then, the attacker requests personal informations about the user

using, for example, fake mails and fake websites. The attacker can also create a service with useful features. Then, if the user registers to this service, the attacker has access to his password. This attack is called honeypot attack [4]. Sadly, remembering a strong password for each account is too hard for a common user.

An existing solution is the password manager [5]. This software acts as locker and store all these hard-to-remember passwords. When the user connects to a service, he just needs the key

of the locker, called the master password. This solution is really comfortable but not perfect, because

this solution is a single point of failure. Password manager can still be victims of security vulnerability. For instance, an attacker can install a keylogger [6] on the user's machine. This

software saves every keyboard inputs, hence the password too. Consequently, if an attacker manages to find the master password, he has access to all the accounts of the user.

Does it exist a perfect solution to provide security and simplicity to a user? This is the purpose of the single password authentication (SPA) protocol. It implements authentication with a third party storage like a cloud or a smartphone. The user only needs a unique password to

*What is the difference between SPA and password manager?  
Why SPA is not vulnerable to single point of failure?*

connect to all his online services. The idea is to create a different key for each online services, then the user uses his unique password to encrypt it and finally to store it into the storage. This solution is simple due to the uniqueness of the password. It also provides security because the online service never learns the user's password or find any way to impersonate him. Furthermore, this solution is not vulnerable to dictionary attack. If an attacker wants to test every possible passwords, it needs to be connected to the storage and to test them in the online service. This problem can be fixed by allowing a small number of attempts to connect. Consequently, since an attack cannot be made offline, dictionary attacks are useless.

This protocol is not the first attempt to provide authentication with single password. example of PAKE is Bellovin and Merritt's Encrypted Key Exchange (EKE) [4] permits to create a secure channel if the client and the server share a password. The problem is that the server learns the user's password. Asymmetric Password-Authenticated Key Exchange (APAKE) [5] is a solution to this problem. The client has a password and the server has a function to verify the password. Unfortunately, this solution is vulnerable to dictionary attack. An attacker can steal the function of the server to test hundreds of passwords. The solution is to authenticate with multiple server. This is the case of SPA which uses a cloud or a mobile storage. The only condition is that the storage and the server do not interact.

α

*↳ condition of what?*

The aim of this project is to implement the four versions of the SPA protocol and compare their performance.

## Chapter 2

# Cryptography

Before going into the protocol details, it is necessary to introduce some ~~advanced~~ notions of cryptography. *that are used in the SPA implementation.*

**Symmetric encryption with one time pad:** Encryption with one time pad consists of generating a key  $k$  of the same length as the message  $m$ . Then the encryption is made using a simple xor operation:

$$c = m \oplus k$$

Then the message can be easily recovered:

$$m = c \oplus k$$

This encryption is really fast and provides perfect security if it respects three conditions:

- 1) The key as the same length as the message.
- 2) The key is used only once.
- 3) The key is truly random.

**Hash:** A hash function  $H$  computes the fixed-length hash value of a given input of an arbitrary length. A ~~good~~ hash function can have ~~one of~~ the following properties:

- 1)  $H$  is collision resistant: ~~Two different messages cannot have the same hash value.~~ *Secure It is hard to find two message which have the same hash value.*
- 2)  $H$  is resistant to the preimage: Given the hash value, it is ~~impossible~~ *hard* to recover the original input.
- 3)  $H$  is resistant to the second preimage: Given the original input, it is ~~impossible~~ *hard* to find another input with the same hash value.

**Digital signature:** The digital signature is a cryptographic technique which authenticate the signer of a message. This signature can be based on RSA. Plain RSA signature is vulnerable, *because* an attacker can combine two signatures to create a new valid signature. Hence, the

RSA-PKCS#1-V1.5 algorithm is implemented *as a* for digital signature. This algorithm uses a hash function and padding to solve this vulnerability. Imagine that Bob sends a message to Alice and that Alice wants to verify the identity of Bob. Alice has the public key of Bob  $(N, e)$  and Bob has his private key  $(d)$ . Bob signs the message  $m$  and then he sends  $m$  and the signature  $s$  to Alice:

$$s = \text{pad}(\text{hash}(m))^d \mod N$$

Then, Alice can unsign  $s$  to authenticate Bob:

$$m' = s^e \mod N = \text{pad}(\text{hash}(m))^{de} \mod N = \text{pad}(\text{hash}(m))$$

Finally, Alice can compute  $\text{hash}(m)$ . If  $m'$  contains  $\text{hash}(m)$  and contains the correct padding format, Alice knows that Bob is the sender of the message.

**MAC:** The Message Authentication Code (MAC) is used for the authentication between users who shared a symmetric key. In this project, we use Hash-based message authentication code (HMAC) [6]. Again, imagine that Bob sends a message to Alice and that Alice wants to verify if the message is sent by Bob. Alice and Bob share a secret key  $K$  and use a hash function  $H$ . Bob computes the MAC  $s$  from the message  $m$  and sends them to Alice:

$$s = h((K \oplus \text{opad}) || h((K \oplus \text{ipad}) || m))$$

$||$  denotes concatenation and opad/ipad denotes a padding of the size of one block. For instance, the block size of SHA-256 is 512 bits.

Then, Alice can compute the MAC with  $m$  to obtain  $s'$ . If  $s = s'$ , Alice knows that Bob is the sender of the message.

**Blind signature:** The blind signature [7] is an extension of the digital signature. It permits to sign a message without revealing the message to the signer. The concept can be hard to understand because it is difficult to guess why a signer would sign a message without knowing it. Let's consider this simple example: Imagine a political party which wants to hold an election. The party wants to authenticate each vote but each elector does not want his vote to be known. The blind signature is a solution to this problem.

In this project, we use a blind signature based on RSA:  $(N, e)$  is the public key and  $(d)$  is the private key of the RSA signature scheme. The idea is to enclose the message using a blind factor  $r$ . Since the signer does not know  $r$ , he is unable to read the message.

The original message is represented as  $m \in \mathbb{Z}_n$  and the signature as  $s$ . First, we add the blind factor  $r \in \mathbb{Z}_n^*$  to the message.

$$m' = r^e m \mod N$$

Then, the signer signs the blinded message  $m'$  using its private key.

$$s' = (m')^d \mod N = r^{ed} m^d \mod N = r m^d \mod N$$

Finally, we remove the blind factor using the inverse in order to obtain the signature of  $m$ .

$$s = r^{-1} s' \mod N = r^{-1} r m^d \mod N = m^d \mod N$$

**Blinding attack:** Nevertheless, the RSA blind signature is not perfect because it is vulnerable to the blinding attack. This comes from the fact that RSA uses the same key to sign or to decrypt a message. Let's consider an example where Alice is a server. She has a single private key to sign and to decrypt messages. Imagine that Bob sends a message  $m$  to Alice using Alice's public key.

$$m' = m^d \mod N$$

Now, imagine that Oscar wants to read this message. If Oscar can intercept the message from Bob and then he can send it to Alice by asking to sign it. Since Alice is using the same private key to sign a message, Oscar gets the original message of Bob.

$$m = m'^d \mod N = m^{ed} \mod N$$

In other words, Alice decrypts the message when she signs it. Therefore, a different key pair must be used for signing and exchanging messages. In this project, we do not use RSA encryption. Therefore, the blinding attack does not concern us.

**Oblivious transfer** The oblivious transfer[8] is a protocol which permits to exchange data between a client and a server. Let's imagine that the server has  $n$  tuples of the form  $(w_i, c_i)$ ,  $w_i$  is an index and  $c_i$  is the corresponding data. The particularity of this protocol is that the client can retrieve the data  $c_j$  from the index  $w_j$  without revealing the value  $w_j$  to the server and without learning any value  $c_i$  where  $i \neq j$

In this project, the oblivious transfer protocol by Ogata and Kurosowa. Let  $H$  be a hash function and  $G$  be a pseudo-random generator. We assume that the server has a private key ( $d$ ) and the client has the related public key  $(N, e)$ . The server starts by creating  $n$  keys using the indexes  $w_i$ .

$$K_i = (H(w_i))^d \mod N$$

Then it uses these keys to encrypt each data  $c_i$  in the following way:

$$E_i = G(w_i \| K_i \| i) \oplus (0^l \| c_i)$$

To execute the xor, each part must have the same length. Hence, the server adds zeros in front of  $c_i$  to respect this property. Concretely,  $l$  is the length of the output of  $G$  minus the length  $c_i$ . Then, the server sends every  $E_i$  to the client. Now, let's consider that the client wants to retrieve data  $c_j$ . Using blind signature, the client is able to reconstruct the key  $K_j$  corresponding to  $w_j$ . So, using a blind factor  $r$ , the client generates:

$$Y = r^e H(w_j) \mod N$$

Then, the client asks to the server for a blind signature. This way, the server cannot learn the value  $w_j$  chosen. At the end, the client received the value:

$$K_j = r^{ed} H(w_j)^d \mod N = H(w_j)^d \mod N$$

Finally, he can decrypt the corresponding data using:

$$(a_i \| b_i) = E_j \oplus G(w_j \| K_j \| i)$$

If  $a_i = 0^l$ ,  $b_i$  corresponds to the data wanted by the client. Since, the client has only one  $K_j$ , he can only decrypt the data  $c_j$  corresponding to  $w_j$ . So, he cannot learn any other value of  $c_i$  with  $i \neq j$ .

## Chapter 3

### Protocol description

The single password authentication protocol (SPA) [9] lets a client (Alice) to connect to every website (Bob) using the same password. The password is used to encrypt a secret stored in an untrusted storage or a trusted mobile device (Carol).

#### 3.1 Overview

First, Alice invents a username and a password which must be as strong as possible: it must look like random. These will be ~~the same~~ <sup>used</sup> for each website Alice wants to connect to. <sup>and Alice only needs to remember them.</sup>

**Registration:** When she registers to Bob, she generates a secret key  $sk$ . She sends her username and  $sk$  to Bob. Then, Alice registers to Carol. She generates an  $id$  and she uses her password to encrypt  $sk$ . Finally, she sends her  $id$  and  $sk$  encrypted to Carol. At that point, Alice only needs to remember her username and her password.

<sup>which is use to connect Bob later (it can be considered as password to connect Bob)</sup>

**Connection:** When Alice wants to connect to Bob, she first needs her username and her password. She uses the username to obtain a challenge from Bob. Then, Alice generates  $id$  again and sends it to Carol to recover  $sk$  encrypted. Now, Alice can decrypt  $sk$  and calculate the response of the challenge. Then Alice sends it to Bob. Finally, Bob verifies the response and accepts the connection if it is correct.

SPA describes four different protocols: server optimal cloud SPA, storage optimal cloud SPA, privacy optimal cloud SPA and mobile SPA.

#### 3.2 Server optimal cloud SPA

**Registration (Figure 1):** First, Alice generates two RSA key pairs. ( $ssk$ ,  $svk$ ) for digital signature used with Bob and ( $bsk$ ,  $bvk$ ) for blind signature used with Carol. To register to Bob, Alice sends him her username and  $svk$ . Then, to register to Carol, Alice starts by signing her password with  $bsk$  to get  $sig = BSign(bsk, Hash(pwd))$ . Then, she uses  $sig$  to encrypt  $ssk$  and to obtain  $ctext = Encrypt(Hash(sig), ssk)$ . Then, Alice generates an  $id = Hash(Alice, Bob)$ . Finally, Alice sends to Carol  $id$ ,  $bsk$  and  $ctext$ . Now, she can forget everything except her username and her password.

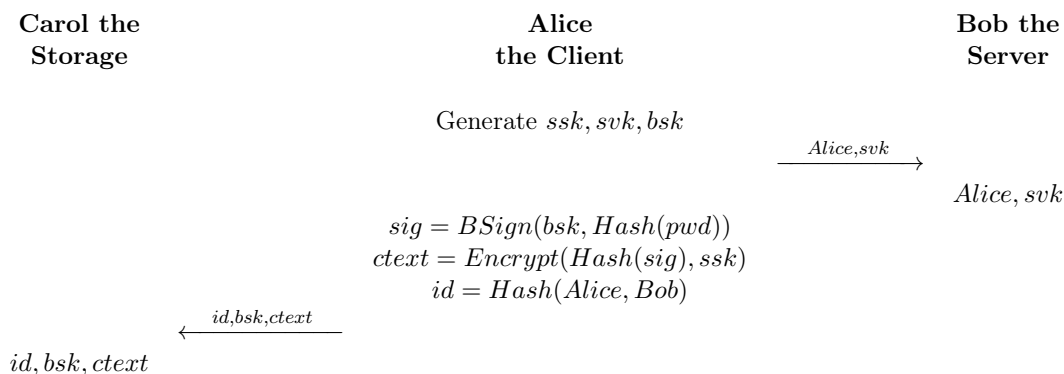


Figure 1: Server optimal cloud SPA registration

✗ **Connection (figure 2):** When Alice wants to connect to Bob, she only sends him her username and gets a challenge. Then, Alice must recover  $ssk$  to compute a response. She begins by recovering her  $id = Hash(Alice, Bob)$  and sends it to Carol. She asks Carol to sign her password by using a blind signature scheme (BS):  $sig = BSign(bsk, Hash(pwd))$ . Finally, Carol sends Alice  $ctext$ . Now, that Alice has  $sig$  and  $ctext$ , she can compute  $ssk = Decrypt(Hash(sig), ctext)$ . Then she can compute the response  $R$  and sends it to Bob which accepts if it is correct.

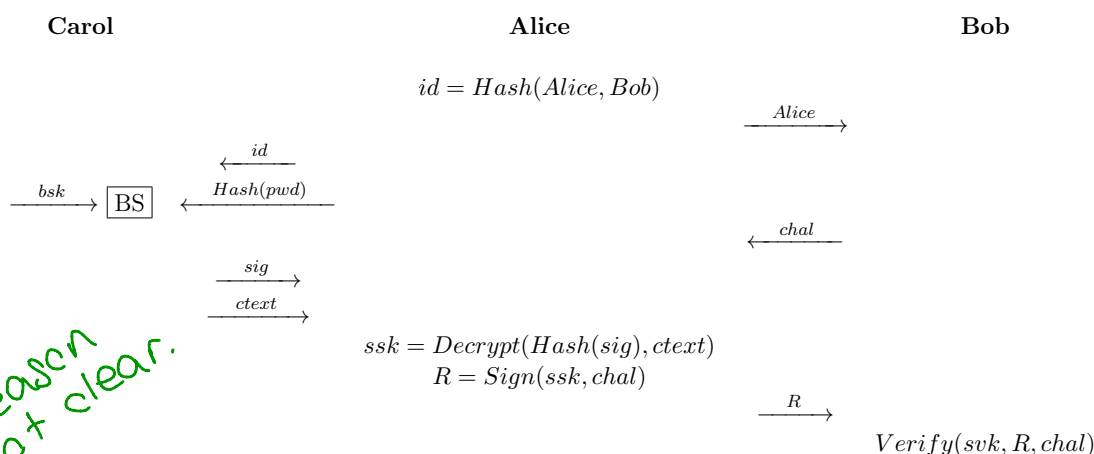


Figure 2: Server optimal cloud SPA connection

This version of the protocol is the most efficient for the server. If Bob is malicious, Alice is safe because Bob never learns Alice's password. Furthermore, challenge-response is used for each connection, hence, Bob cannot create a dictionary attack. If Carol is malicious, Alice is not anonymous nor unlinkable. Alice is not anonymous because Alice's username is calculated using  $Hash(Alice, Bob)$ . So Carol can launch a dictionary attack to find it. Furthermore, Alice is linkable by linking the found username with the id provided. Regarding Bob, he never learns Alice's password. In addition, since the challenge-response is used for each connection, Alice is protected against dictionary attack.

### 3.3 Storage optimal cloud SPA

**Registration (figure 3):** As Server optimal cloud SPA, Alice first generates two RSA key pairs  $(ssk, svk)$  and  $(bsk, bvk)$ . To register to Bob, Alice sends him the username,  $svk$  and  $bsk$ . Then, to register to Carol, Alice begins by generating her  $id = BSign(bsk, Hash(pwd))$ . Then, she encrypts  $ssk$  with her password to get  $ctext = Encrypt(Hash(pwd), ssk)$ . Finally, Alice sends  $id$  and  $ctext$  to Carol.



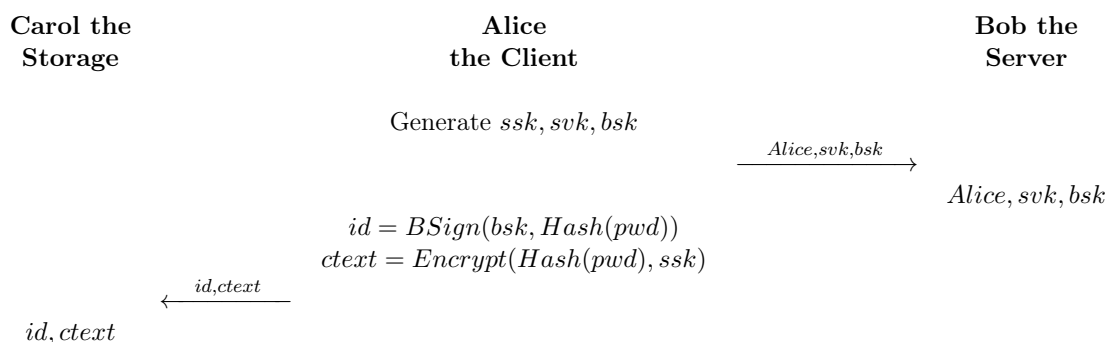


Figure 3: Storage optimal cloud SPA registration

**Connection (Figure 4):** When Alice wants to connect to Bob, she first sends him her username. Then, she asks Bob to sign her password (BS):  $id = BSign(bsk, Hash(pwd))$ . Finally, Bob sends Alice a challenge. Then, Alice needs  $ssk$  to compute the response. Now that she has recovered her  $id$ , she sends it to Carol and Carol sends her the corresponding  $ctext$ . Then, Alice can recover  $ssk = Decrypt(Hash(pwd), ctext)$  and compute the response  $R = Sign(ssk, challenge)$ . At the end, Alice sends the response to Bob which accepts if it is correct.

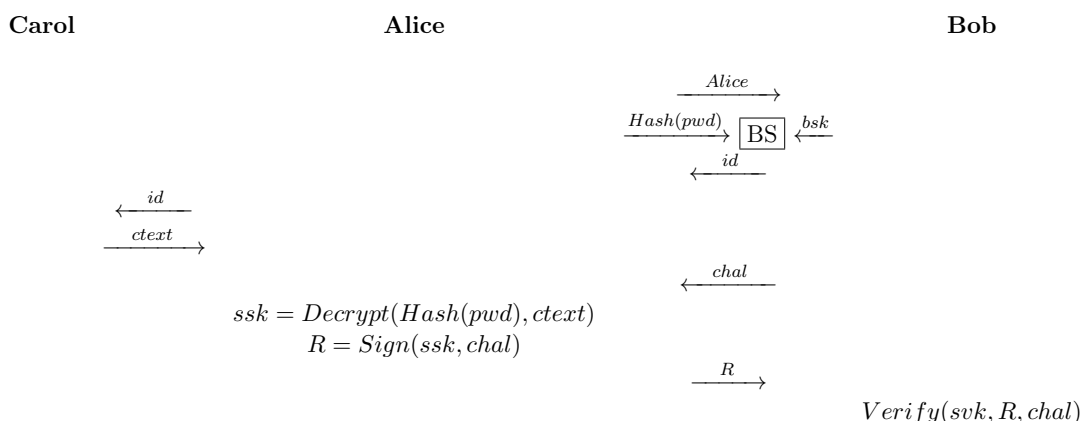


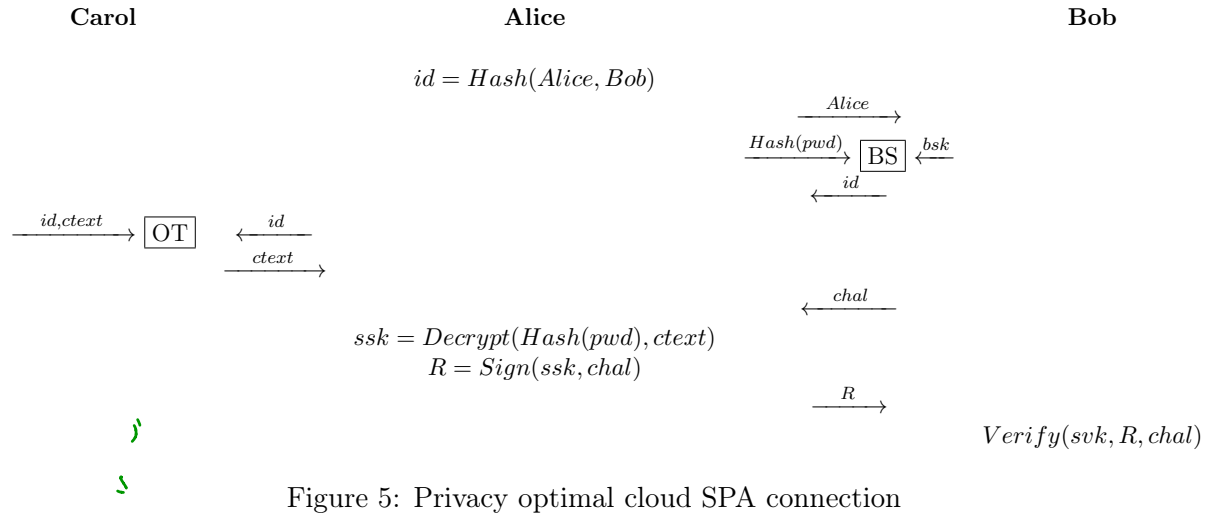
Figure 4: Storage optimal cloud SPA connection

This protocol is the most efficient for storage. If Bob is malicious, Alice is safe like the server optimal protocol. If Carol is malicious, Alice is anonymous but still linkable. Alice is anonymous because Alice's  $id$  is a signature. Hence, Carol cannot recover the  $id$  using a dictionary attack. Alice is linkable because Carol still has access to the  $id$ . As server optimal, Bob never learns Alice's password, so Alice is also protected against a dictionary attack from Bob.

### 3.4 Privacy optimal cloud SPA (figure 5)

Privacy optimal protocol is the same as storage optimal. Privacy means that this protocol provides anonymity and unlinkability from Bob and Carol. The only difference is made during the connection. When Alice retrieves  $ctext$  from Carol, they use oblivious transfer (OT). If Bob is malicious, Alice is safe like the storage optimal protocol. If Carol is malicious, Alice is anonymous and unlinkable. Alice is anonymous because the Alice's  $id$  is a signature. Then, Alice is unlinkable because Carol never learns the  $id$  of Alice due to oblivious transfer.

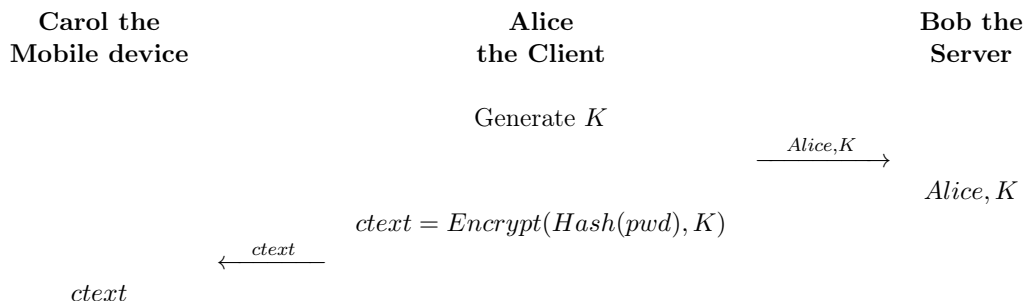
explain who is sender-receiver in OT. What are the inputs of OT



### 3.5 Mobile SPA

Mobile SPA does not require a cloud storage. It uses a trusted mobile device instead. The name of Carol is kept to describe the mobile device. This protocol requires a connection between Alice and Carol. It can be a direct internet connection (SSL) or an indirect internet connection (Bluetooth, Wi-Fi, USB). Carol must also be able to display and analyze a human-readable information, like a QR code or a sound. The main advantage of this protocol is that Alice can connect to any terminal, even untrusted. Indeed, she types the password on her mobile. Consequently, the untrusted terminal is unable to obtain the password. However, the mobile can be lost or stolen. In this case, Carol becomes malicious. Fortunately, the data stored is encrypted. So, an attacker does not have access to Alice's secret. In addition, Alice's username is not stored in the mobile. Hence, Alice is anonymous and unlinkable to an attacker. In addition, if Bob is malicious, he never learns Alice's password and cannot mount a dictionary attack since each connection uses challenge-response.

**Registration (figure 6):** Alice starts by generating a symmetric key  $K$ . To register to Bob, she sends him her username and  $K$ . Then, to register to Carol, Alice encrypts  $K$  using her password:  $ctext = \text{Encrypt}(\text{Hash}(pwd), K)$ . Now, Alice can forget everything except her username and her password.



**Connection (figure 7):** When Alice wants to connect to Bob, she first sends him her username to get a challenge. This challenge can be sent to Alice (QR code or audio file) or directly to Carol (SMS). If the challenge is sent to Alice, she uses her mobile device to read it.

Then, Alice provides her password to Carol. Now, Carol can recover  $K = \text{Decrypt}(\text{Hash}(\text{pwd}), \text{ciphertext})$  and computes the response  $R = \text{MAC}(K, \text{challenge})$ . Carol shortens the response using a *TRIM* function and displays it to Alice. Now, Alice can provide  $R$  to Bob which verifies it and accept the connection if it is correct.

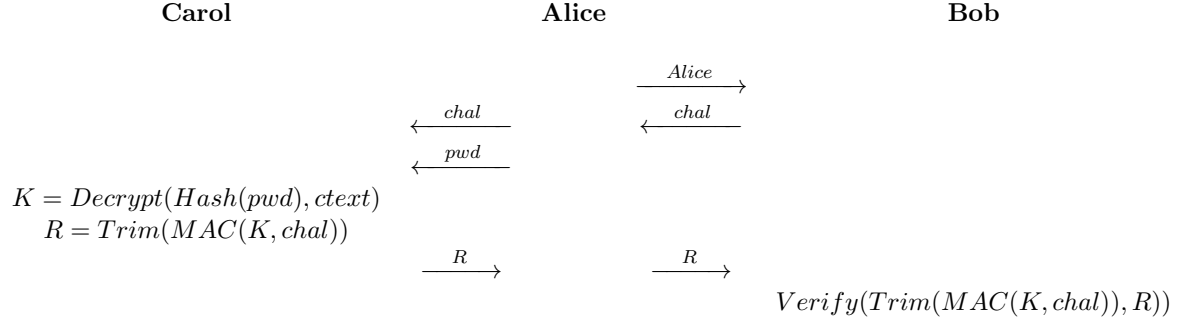


Figure 7: Mobile SPA connection

# Chapter 4

## Implementation

I decided to implement this project in Java. This programming language is really complete and offers a wide range of libraries and documentation. Since this project requires many features (socket programming, website creation, cryptographic elements, user interface conception, mobile application), Java is one of the best choice. Furthermore, it is the programming language I feel the most comfortable with. I started my implementation by creating my own cryptographic library.

### 4.1 Cryptographic library

α **Symmetric algorithms** The first type of algorithms implemented ~~is~~ <sup>are</sup> contained in the SymmetricEncryption.java class. It implements functions to encrypt, decrypt or MAC a given message using a symmetric key. The symmetric ~~algorithm~~ used is one time pad. It is very fast since it uses one xor operation. It also gives perfect security because:

- We use uniformly distributed random key
- Every message in SPA protocol is encrypted with a new key
- The key size and the message size are the same.

Regarding MAC, the implemented algorithm is HMACSHA256. It generates MAC efficiently and in a very secured ~~a~~ way. } encryption

**Asymmetric algorithms** The second type of algorithms implemented are contained in AsymmetricEncryption.java class. This class only implements RSA. The choice of this algorithm comes from the fact that blind signature can be based on it. So, the class contains functions to encrypt, decrypt, sign and verify the signature of a message. It also implements functions for blind signature: generating a random blind factor, blind and unblind a message. Like, SymmetricEncryption.java, there exist libraries [10] to implement these functions. Nevertheless, I prefer to make my own implementation. It makes the code more readable and it permits to understand in details how blind signature is implemented.

α **Key generation:** Encryption can be relatively easy to implement, but key generation is more sophisticated. Therefore, I decided to generate keys using the existing java libraries [10] in MyKeyGenerator.java class. So, the generation is secure ~~it~~ and efficient. Concretly, AES, RSA and HMACSHA256 keys are generated using these libraries. The key of one time padding is generated using a secure random generator [11]. Each key can be generated in three ways: from random, from a password or into a file which includes the key.

**Hash** The class Hash.java implements SHA256 using java libraries [10]. SHA256 provides the right balance between security and performance.

**Oblivious transfer** Since oblivious transfer is used by a server and a client, I did not choose to create a static library. I created two objects OTSender and OTReceiver which can be instantiated by a server and a client to execute the oblivious transfer protocol. OTSender needs a Map with  $w_i$  the keys and  $c_i$  the corresponding data. It also requires an RSAPrivateKey.

OTSender automatically generates the keys  $K_i$  when it is instantiated. Then it provides two functions. One to encrypt the data using  $K_i$  by generating  $E_i$  and one to generate  $K_j$  from the index  $Y$  received from the client. OTReceiver needs an index  $w_j$  and an RSAPublicKey to be instantiated. Then it provides four additional functions. The first one blinds the index using a given blind factor  $r$  to generate  $Y$ . The second one unblinds the value  $K_j$  received by the server. The third one is used to decrypt the data  $E_i$  received by server using  $K_j$ . And finally, the last one checks the data generated during the decryption to find and return the corresponding value.

## 4.2 Implementation of the storage

**Database connection:** After implementing the necessary cryptographic components, the next step was to find an efficient database for the storage. This database must have many properties. It must be free, fast and it must offer a performance report. I chose to use the SQL database of the Microsoft Azure platform [12]. It is free for academic purposes [13]. It is fast because their servers are available in Europe. Furthermore, their platform automatically provides many statistics about the database. The only disadvantage is the storage size since there is only 35 MB available. In the concern of this project, it is enough, since any user store only a small amount of data.

The implementation has two parts: SQL and java. The SQL part means creation of tables and testing queries. I created two tables. Storage\_server\_optimal for the server optimal protocol. It contains three columns (id, bsk and ctext), each one has type varbinary(256) to store 2048 bits:

id	bsk	ctext
0x582A749BFF...	0x80128461BB...	0x4816B4F323...
0x769398B3F3...	0x68154A3FF1...	0x729BA5BB72...

Storage\_storage\_optimal for storage optimal and privacy optimal protocols. It contains two columns (id, ctext), each one is also of type varbinary(256):

id	ctext
0x549748104A...	0x6381036A24...
0x32F38BA233...	0x67C7471B31...

The SQL code can be found in the SQLCommands.txt file.

The java implementation is done using the JDBC library [14] created by Oracle and modified by Microsoft for their database. I provide database connection with an object written in the DatabaseConnector.java class. When the object is instantiated, the user provides which protocol he uses and then the corresponding connection is automatically created with the database. Then, I provide functions to execute SQL queries: insertion, search and deletion. For privacy optimal protocol, I also provide a function to fetch every elements of the table in a random order.

**Connection between storage and client:** When the database was fully implemented, the user must be able to send and retrieve data. Of course, a simple database does not provide features like blind signature or oblivious transfer. So, I first implemented a server for the storage. This server, contained in the Storage.java class, offers an interface between the client and the database. It only includes a main method which creates an SSL socket and wait for a

client connection. When the server starts, it creates a pool of 20 executors [15], so 20 clients can connect at the same time. When a client wants to connect, the server executes one of the existing thread which takes care of him.

This server uses two enumerations to classify each client. The first one is in ProtocolMode.java class. It contains 4 elements: Server\_optimal, Storage\_optimal, Privacy\_optimal and Mobile. These represent each protocol of chapter 3. Of course, Mobile is not used in this implementation since this protocol does not require a storage. However, it is useful for the implementation of the website.

The second enumeration is found in ClientToStorageMode.java. It contains 2 elements: Store and Retrieve. These represent the possible user action: Store during the registration phase and Retrieve during the connection phase.

The thread which takes care of a client is contained in ClientAdministratorThread.java. It first receives the user protocol and the user action. Therefore, it identifies the client and acts consequently:

Server\_optimal and Store: The thread receives *id*, *bsk* and *ctext* and stores them in the database.

Server\_optimal and Retrieve: The thread receives *id* and *Hash(pwd)*, fetches *bsk* and *ctext* from the database and uses the blind signature scheme to send *sig* and *ctext* back.

Storage\_optimal and Store: The thread receives *id* and *ctext* and stores them in the database.

Storage\_optimal and Retrieve: The thread receives *id*, fetches *ctext* from the database and sends it back.

Privacy\_optimal and Store: The thread receives *id* and *ctext*, stores them in the database and sends the public key for oblivious transfer back.

Privacy\_optimal and Retrieve: The thread receives *id*, fetches the Storage\_optimal table and uses oblivious transfer to send *ctext*

When the server-side of the storage was implemented, I implemented the corresponding client-side. It is contained in the StorageClient.java class. This object acts as an interface for the user. When it is instantiated, it offers two methods: one to store data into the storage during the registration phase and one to retrieve the data during the connection phase. This object contains two constructors: one for the Server\_optimal protocol and one for the Storage\_optimal and Privacy\_optimal protocols.

**Data storage:** When the client wants to store data, this object creates an SSL connection and runs two threads. One to send the data (ClientSenderThread.java) and one to receive a response from the server (ClientReceiverThread.java). These threads are executing with a single executor [15] to avoid parallelism errors.

In the Server\_optimal implementation, the client sends *id*, *bsk* and *ctext* and receives an acknowledgement.

In the Storage\_optimal implementation, the client sends *id* and *ctext* and receives an acknowledgement.

In the Privacy\_optimal implementation, the client sends *id* and *ctext* and receives the public key for oblivious transfer.

**Data retrieval:** When the client wants to retrieve the data, the object also creates an SSL connection but this time, it runs one thread (ClientRetrieverThread.java). This thread fetches the data from the server computes and returns the private key *ssk* wanted by the client.

In the Server\_optimal case, the client sends *id*. Then, he executes the blind signature scheme to obtain *sig*. Finally, he receives *ctext* and computes *ssk*.

In the Storage\_optimal case, the client sends *id*, receives *ctext* and computes *ssk*.

In the Privacy\_optimal case, the client sends *id*. He uses oblivious transfer to obtain *ctext* and computes *ssk*.

### 4.3 Implementation of the mobile application:

The mobile protocol requires a mobile application. I chose to develop it on Android. This platform is based on java and has a lot of documentation. For this project, this application must read a challenge and display the response to the user. Since Google provides a free and well-documented API to read barcodes [16], I decided to display the challenge as a QR code. The application uses its camera to read it and to display the response on the screen.

I first cloned a basic application from the official github repository of Google [17]. This application permits to take a picture using the camera and to display its data. Then, I modified the code to work with this project. Now, when the user starts the application, he first needs to enter his password. Then, like the cloud-based protocols, he can either register to the website or connect to it.

If the user wants to register, he has to enter the IP address of its computer. This IP address will be given by the computer software. Then, an SSL connection is opened between the mobile application and the computer software. The latter automatically sends *ctext*. Finally, the android application displays a pop-up to tell the user if the registration has succeeded. Concretely, the application writes *ctext* into a file and store it in the smartphone storage.

If the user wants to connect to the website, the application starts to scan a QR Code. When it finds a QR Code, it recovers the challenge from it, computes and displays the response. This is done without any user interaction, contrary to the original application.

### 4.4 Implementation of the website:

**Developping website in Java:** Java permits to create a website using Java EE [18]. I used this platform to create a dynamic website. *Html* pages are generated using *jsp* files, a format similar to *php*, and the management of these files are made using a java object called a *servlet* [19]. Concretely, if a user connects to a precise link, he does not directly obtain the corresponding *html* file. The link is related to a servlet which treats the request, generates the response and return a *jsp* file. This implementation provides security against vulnerabilities like cross-site scripting (XSS) [20].

**Website access:** In this project, the website is accessed in two ways. If the user wants to register or to obtain a challenge, he will use the computer software. If the user wants to submit the response, he will use his web browser. Consequently, I implemented two java servlets. The first one is contained in RegistrationServlet.java, it is related to the url: <https://128.178.73.85:8443/pro/register> and can only be accessed from the client software because it only implements the http post request. When a user wants to register, this

class checks the values received and create a new client. When a user wants to connect, this class checks that the user is registered and sends him a challenge. The second one is contained in ConnectionServlet.java, it is related to `https://128.178.73.85:8443/pro/connect` and can be accessed from a web browser. When the user wants to connect, he needs to give a username and the response of the challenge. If the response is correct, the servlet generates a new page to show the user that he is connected.

**Client management:** The web-server also implements a class to represent a client. This class is contained in Client.java. A client can be instantiated using three constructors: one for Server\_optimal protocol, one for Storage\_optimal protocol and one for Mobile protocol.

In the case of Server\_optimal, the client is instantiated with a *username* and his public key *svk*.

In the case of Storage\_optimal, the client is instantiated with a *username*, his public key *ssk* and his private key for blind signature scheme *bsk*.

In the case of Mobile, the client is instantiated with a *username* and his secret key *K*.

Note that there is no difference between Storage\_optimal and Privacy\_optimal for the server. Therefore, both are represented with the Storage\_optimal protocol.

This class also implements an enumeration to represent the state of the client. When the client is created, his state is set to *REGISTERED*. When a challenge is sent to the client, his state is set to *READYTOAUTH* and finally, a client which provides the good response is set to *AUTH*. Furthermore, like the implementation of the storage, the web-server uses the ProtocolMode.java class to enumerate each possible protocol. Finally, the class implements other functions: one to set a challenge to a registered client, a getter for each value of the client and one to change the state. More details can be found in appendix (?). The website generates a dynamic Map to store the connected clients. The username are used as keys and the client objects are the corresponding values.

**Website deployment:** When the website was developped, it was exported in a WAR file [20]. Then Apache Tomcat [21] was installed on the web-server. This software uses the WAR file to implement the website and to run it.

**Client-side of the website:** When the server-side of the website was implemented, I implemented the corresponding client-side. This part is really similar to the client-side of the storage. The only difference is that the request are sent using http request [22]. This implementation is contained in ServerClient.java. It contains three constructors, one for each protocol used by the server. Furthermore, this object provides two methods: one to register to the website and one to get a challenge.

When the user wants to register to the server, the object creates a new thread (ClientSenderThread.java) which sends the data to the server. Then, the thread waits for an acknowledgement and return it.

In the Server\_optimal implementation, the client sends *username* and *svk*.

In the Storage\_optimal implementation, the client sends *username*, *svk* and *bsk*.

In the Mobile implementation, the client sends *username* and *K*.



When the user wants to obtain a challenge, the object also creates a new thread. The latter sends the username, waits for a challenge and returns it.

In the `Server_optimal` implementation, the client sends *username* and receives a challenge.

In the `Storage_optimal` implementation, the client uses blind signature scheme to blind his password. Then, he sends *username* and *password* and receives *id* and a challenge.

In the `Mobile` implementation, the client does not connect using the client software. He simply types his username on the website to obtain the challenge in the form of a QR Code.

## 4.5 Note on SSL implementation:

In a normal implementation of SSL, the server sends its public key and a certificate to the client. The latter checks that the certificate is trusted by a certification authority. In this project, the RSA key pairs used are self-generated. Hence, an external user cannot trust my certificate and must reject the connection. For this reason, the certificate is stored within the project to accept the connection.

For the connection with the storage and the android application, I use SSL socket [23] to create my own secure connection in an unused port. I get the certificate from a file to create it. This implementation can be found in `SSLClientUtility.java` and `SSLServerUtility.java`. These classes are inspired from [24].

For the connection with the website, I cannot create my own connection since https works with port 8443. Consequently, the certificate must be imported inside the Java Virtual Machine (JVM) and the web-browser. //TODO: Explain how to import certificate

## 4.6 Implementation of the user interface:

When the website was implemented, I implemented a user interface. This interface is necessary for two reasons: First, it permits to send and retrieve data with the storage. Second, it is a user-friendly way to interact with the server. For instance during the connection phase of the storage optimal version of the protocol, the user gives his password to this user interface and then the blind signature scheme is invisible for him. This interface is implemented using SWT [25], it permits to design a basic interface easily. It contains four input fields: one to select the protocol, one to type the username, one to type the password and one to select the website. Then, it contains two buttons: one to register and one to connect. Finally, it contains two outputs: one to display a QR code if the mobile protocol is used and one to display informations, so that the user understands what happens.

**Registration:** When the user wants to register, he first needs to fill the different input fields. Then, when he clicks on the register button, the software instantiates two objects: `ServerClient` and `StorageClient`.

When the `Server_optimal` protocol or the `Storage_optimal` protocol is selected, the software simply registers to the server and to the storage using the corresponding methods.

When the `Privacy_optimal` protocol is selected, the software registers to the server and to the storage. In addition, it receives the public key for oblivious transfer and store it into a file.

When the `Mobile` protocol is selected, the software registers to the server, displays its IP address and waits for a connection with the android application.

**Connection:** When the user wants to connect, he clicks on the connect button. Like registration, the software instantiates ServerClient and StorageClient.

When the Server\_optimal protocol is selected, the software retrieves the *ssk* from the storage and obtains a challenge from the server using the corresponding methods. Then, it computes the response and copy it into the clipboard. Finally, it display a message to tell the user that he can connect on the website.

When the Storage\_optimal protocol is selected, the software retrieves *id* from the server and obtains a challenge. Then, it uses this *id* to retrieve *ssk* from the storage. Finally, it computes the response and copy it into the clipboard.

When the Privacy\_optimal protocol is selected, the software retrieves *id* from the server and obtains a challenge. Then, it recovers the public key for oblivious transfer from the file. Then, it uses this key and *id* to retrieve *ssk* from the storage. Finally, it computes the response and copy it into the clipboard.

When the Mobile protocol is selected, the software obtains a challenge from the server. Then, it computes a QR code and displays it. Finally, the response can be computed using the android application.

## 4.7 Performance

Now that the four protocols are implemented, I compared their performance. To compute the performance of each protocol, three times are calculated: the time between the client and the website (Alice-Bob), the time between the client and the storage (Alice-Carol) and finally the total time. The registration time represents the time needed to send data to the storage or the website. The connection time represents the time needed to retrieve the data and compute the response. The total time represents the overall time considering client-server connection and the necessary computations. The following arrays display the average time between 25 tests in milliseconds when the RSA key-length is 1024 bits.

**Server optimal:** Here is an array representing the average time for the server optimal protocol:

	Alice-Bob	Alice-Carol	Total
Registration	15	35	50
Connection	22	218	245

During the registration phase, we see that the registration lasts longer with Carol. This result is not surprising. Indeed, to register with Carol, Alice must compute *sig*, *ctxt* and *id* before sending the data. When Alice registers with Bob, she just needs to send *svk*.

During the connection phase, the connection lasts longer with Carol. Once again, this result is not surprising. When Alice retrieves the data from Bob, she uses the blind signature scheme, contrary to the connection with Bob which only requires to receive a challenge.

Connection requires more calculations than registration, this is why the total time of the connection is bigger than the sum of the connection time between Alice and Bob and the connection time between Alice and Carol. Nevertheless, this extra time is insignificant.

**Storage optimal:** Here is an array representing the average time for the storage optimal protocol:

I didn't understand if you add communication through network in your performance analysis. I think you should separate computation time for Alice, Bob and Carol as well so that we see how heavy the cryptographic operations are.

	Alice-Bob	Alice-Carol	Total
Registration	29	36	65
Connection	237	41	284

During the registration phase, we see that the registration lasts a bit longer with Carol. Like the server optimal protocol, this result is not surprising because Alice must do some computations before sending the data to Carol.

During the connection phase, the connection lasts longer with Bob. Once again, this result is normal. In the storage optimal protocol, the blind signature scheme is used with Bob and not with Carol. This requires more computations and consequently more time.

Like server optimal, connection requires more calculations than registration, this is why the total time of the connection is bigger than the sum of the connection time between Alice and Bob and the connection time between Alice and Carol. Nevertheless, this extra time is insignificant.

**Privacy optimal:** Here is an array representing the average time for the privacy optimal protocol:

	Alice-Bob	Alice-Carol	Total
Registration	26	38	64
Connection (0 element)	225	48	279
Connection (200 elements)	218	277	505
Connection (500 elements)	223	1773	2004
Connection (1000 elements)	229	3164	3407

Can you draw a graph?

The privacy optimal protocol is similar to the storage optimal protocol. When the storage is empty, the result is really close to storage optimal. But, due to the obvious transfer, the storage needs to send all its data to the client. Bigger the database is, longer the connection will last. We see this result in the array: when the storage has a thousand clients, the connection lasts more than three seconds.

**Mobile:** Here is an array representing the average time for the mobile protocol:

	Alice-Bob	Alice-Carol	Total
Registration	21	127	148
Connection	15	1362	-

Computing the performance of the mobile protocol is more complicated since it requires more user interactions. During the registration phase, the time to register with the storage is calculated when the user enters the correct IP address. We see that the registration with the server is really similar to server optimal. This is not surprising since Alice only sent her username and her key to Bob. The registration with the storage is longer, this shows that a connection with a smartphone lasts longer.

The connection time is calculating directly on the server and on the smartphone. It represents the time that the server needs to generate a challenge and the time that the smartphone needs to read this challenge and compute the response. We see that generating a QR code is really fast. In addition, reading the QR code and compute the response lasts much longer. This result is not surprising since launching the smartphone camera is a long process.

When Alice wants to connect, she needs to use her smartphone to read the QR code. Since this

user interaction is not measurable precisely, I do not consider the total time for this protocol.

**2048 bits length-key:** Here is an array representing the average connection time for the server optimal and storage optimal protocol, both requires blind signature and digital signature with an RSA key-length of 2048 bits:

	Alice-Bob	Alice-Carol	Total
Server optimal	23	258	287
Storage optimal	282	43	338

As expected, a connection with a longer key increase the connection time. Nevertheless, this increase is quiet small and reasonable.

**Personal conclusion:** From the result above, the difference between 1024 and 2048 RSA key-length is small. To enforce security, I think that 2048 bits key-length should be used. In addition, server optimal and storage optimal have fast and similar results, hence both are good. α Privacy optimal enforces ~~security~~ <sup>privacy</sup> but the connection time increases quickly. Consequently, the storage becomes vulnerable. An attacker can register a thousand times to the storage to decrease its performance. Consequently, I think that this protocol is not usable. Finally, the mobile protocol is longer than server optimal or storage optimal. However, this protocol is still fast and it permits to connect from any terminal. Hence, this protocol should also be used.

## 4.8 Conclusion:

# Chapter 5

## References

- [1] PEBCAK — Wiktionary. Available from: <https://fr.wiktionary.org/wiki/PEBCAK>
- [2] What is dictionary attack? - Definition from WhatIs.com. SearchSecurity. Available from: <http://searchsecurity.techtarget.com/definition/dictionary-attack>
- [3] Password manager. In: Wikipedia. 2017. Available from: [https://en.wikipedia.org/w/index.php?title=Password\\_manager&oldid=812886172](https://en.wikipedia.org/w/index.php?title=Password_manager&oldid=812886172)
- [4] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In 1992 IEEE Symposium on Security and Privacy, pages 72–84. IEEE Computer Society Press, May 1992.
- [5] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In V. Ashby, editor, ACM CCS 93: 1st Conference on Computer and Communications Security, pages 244–250, Fairfax, Virginia, USA, Nov. 3–5, 1993. ACM Press.
- [6] Hash-based message authentication code. In: Wikipedia. 2017. Available from: [https://en.wikipedia.org/w/index.php?title=Hash-based\\_message\\_authentication\\_code&oldid=815040921](https://en.wikipedia.org/w/index.php?title=Hash-based_message_authentication_code&oldid=815040921)
- [7] Kelum Senanayake. Blind Signature Scheme. Available from: <https://fr.slideshare.net/kelumkps/blind-signature-scheme>
- [8] W. Ogata and K. Kurosawa. Oblivious keyword search. Journal of Complexity, 20(2-3):356–371, 2004. Available from <http://eprint.iacr.org/2002/182/>
- [9] Acar T, Belenkiy M, Küpçü A. Single Password Authentication. 2013. Report No.: 167. Available from: <http://eprint.iacr.org/2013/167>
- [10] javax.crypto (Java Platform SE 7 ). Available from: <https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html>  
java.security (Java Platform SE 7 ). Available from: <https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html>
- [11] SecureRandom (Java Platform SE 8 ). Available from: <https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html>

- [12] SQL Database – Cloud Database as a Service | Microsoft Azure. Available from: <https://azure.microsoft.com/en-us/services/sql-database/>
- [13] Azure in Education | Microsoft Azure Available from: <https://azure.microsoft.com/en-us/community/education/>
- [14] mssql-jdbc: The Microsoft JDBC Driver for SQL Server is a Type 4 JDBC driver that provides database connectivity with SQL Server through the standard JDBC application program interfaces (APIs). Microsoft; 2017. Available from: <https://github.com/Microsoft/mssql-jdbc>
- [15] Executor (Java Platform SE 8 ). Available from: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html>
- [16] Barcode API Overview | Mobile Vision | Google Developers. Available from: <https://developers.google.com/vision/android/barcodes-overview>
- [17] android-vision: Sample code for the Android Mobile Vision API. Google Samples; 2017. Available from: <https://github.com/googlesamples/android-vision>
- [18] Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network | Oracle. Available from: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>
- [19] HttpServlet (Servlet API Documentation). Available from: <https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServlet.html>
- [20] WAR (file format). In: Wikipedia. 2017. Available from: [https://en.wikipedia.org/w/index.php?title=WAR\\_\(file\\_format\)&oldid=811702248](https://en.wikipedia.org/w/index.php?title=WAR_(file_format)&oldid=811702248)
- [21] Apache Tomcat®. Available from: <http://tomcat.apache.org/>
- [22] HttpServletRequest (Servlet API Documentation). Available from: <https://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/http/HttpServletRequest.html>
- [23] SSLSocket (Java Platform SE 7) Available from: <https://docs.oracle.com/javase/7/docs/api/javax/net/ssl/SSLSocket.html>
- [24] gpotter2/SSLKeystoreFactories: Util class to generate SSLSocket and SSLServerSocket from a Keystore cert file. Available from: <https://github.com/gpotter2/SSLKeystoreFactories>
- [25] Guindon C. SWT: The Standard Widget Toolkit. Available from: <https://www.eclipse.org/swt/>