Summer 2017

UFace: Your universal password no one can see

Nicholas Steven Hilbert

Department:

UFACE: YOUR UNIVERSAL PASSWORD NO ONE CAN SEE

by

NICHOLAS STEVEN HILBERT

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER SCIENCE

2017

Approved by

Dr. Dan Lin
Dr. Wei Jiang
Dr. Jennifer Leopold

ABSTRACT

With the advantage of not having to memorize long passwords, facial authentication has become a topic of interest among researchers. However, since many users store images containing their face on social networking sites, a new challenge emerges in preventing attackers from impersonating these users by using these online photos. Another problem with most current facial authentication protocols is that they require an unencrypted image of each registered user's face to compare against. Moreover, they might require the user's device to execute computationally expensive multiparty protocols which presents a problem for mobile devices with limited processing power. Finally, these authentication protocols will not be able to be implemented in real systems because they take too long to execute. In this paper, we present a novel privacy preserving facial authentication system, called UFace. Not only does UFace limit the amount of computation for a user's mobile device, but it also prevents unencrypted images from leaving a user's possession while finishing the authentication protocol within seconds. Web services can now outsource their authentication protocol to UFace so that each web service only needs to handle its own functionality. UFace guarantees that it can correctly authenticate each user with 90% accuracy, prevent attacks from using online photos and that all data used in the authentication protocol is done on encrypted randomized data. In other words, only the user can see the facial image and feature vector used for authentication; all other parties execute the protocol using seemingly random information. UFace was implemented through two facets: a mobile client application to obtain and encrypt the feature vector of each user's facial image, and a server protocol to securely authenticate a feature vector using secure multiparty computations. The experimental results demonstrate that UFace can be used as a third party authentication tool for any number of web services.

ACKNOWLEDGMENTS

I'd like to thank Dr. Dan Lin for guiding me along my graduate studies, Dr. Wei Jiang for teaching me about all the security protocols and cryptography tools, Christian Storer for designing and implementing the protocol for image comparison, and everyone that provided their facial image for testing.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

Many websites today require users to create an account with a username and password to utilize the web service fully. Statistics [Tagat, 2012] show that each Internet user has an average of 26 different online accounts, with individuals between the age of 25 to 34 having an average of 40 accounts each. With so many different accounts - and thus, many needed passwords - some passwords are bound to be reused or changed ever so slightly due to the challenge of memorizing many different passwords. The surprising fact is that a person uses, on average, just 5 unique passwords for all their accounts [Tagat, 2012]. However, using the same password across multiple accounts has opened the door to attackers and is becoming the main cause of the dramatic rise in online fraud. For example, if 1 web service is hacked and all of the passwords are released, then the attackers could have your password for multiple different websites if the same password was reused.

The question this paper aims to solve: Is there a way that does not require individuals to memorize many different passwords while still preventing attackers from accessing confidential information? Face authentication is one potential solution to this. This tool means users will only need to send an image of their face (or a feature vector representing their face in this case) to prove their identities - much easier than trying to remember the password that correlates with the service being used. Since face authentication is a relatively new technology, it still needs to overcome several critical challenges: maintaining high accuracy of authenticating a user's face, preventing masquerade attacks by using old images found on social media websites, preserving privacy of users' information that have been used for authentication, and accomplishing authentication in real-time. The accuracy of authenticating based on facial recognition is no longer a major concern since certain algorithms can achieve an accuracy of over 90% [Tan and Triggs, 2010]. However, the remaining challenges have not been well addressed. Specifically, in existing face

authentication systems, it is possible for attackers to reuse the photos obtained from social networks and then be authenticated as the photo owners. To prevent such impersonation, the latest technique is face liveness detection [Li et al., 2015]. Unfortunately, the face liveness detection approach has recently also been proved to be vulnerable by researchers [Xu et al., 2016] who can create realistic 3D facial models with a handful of pictures from social media to spoof the face liveness detection.

To overcome these security and privacy challenges during facial authentication and enable its wide adoption in web services, this paper proposes a novel privacy-preserving face authentication system, called UFace, where “U" stands for both “your" and “universal". The main idea is that users will take extremely close-up images of their face that will only be used for authentication purposes using their mobile devices. These close-up images are rarely shared online and can thus be used for authentication purposes. The reason they are not shared online comes from the fact that the focal point is much closer to the user’s face and thus causes the shape of the face to change and appear more narrow. Plus, when people do post close-up pictures online, these images do not match the specifications UFace desires when taking pictures and would not work within the system. The experiments run also show that these close-up images cannot be duplicated by attackers. If the attacker tries to use a device with the same camera capabilities and zooms in to take the victim’s photo from a distance, then the face will have a different appearance due to the camera’s focal point being further from the face. The face will seem to be more round and the UFace system will be able to see this change and prevent authentication. If the attacker tries to crop a face of the user found online, then once again the face will have a different appearance due to the same reasoning as before and thus will prevent authentication. So, now that a specific type of image has been identified for authentication, how can that information remain safe so no attacker can gain access to the data and reuse it? To achieve this, UFace has an efficient and secure privacy-preserving authentication protocol that keeps each user’s image private during the entire procedure. The first step is to convert the image of a user’s face to a feature

vector and delete the original image. Afterwards, the feature vector is then encrypted before being transferred to begin the authentication protocol. Thus, the data stored in the UFace system is an encrypted feature vector of each user's facial image and not the facial image nor the unencrypted feature vector. It is worth noting that this work is unique compared to existing privacy-preserving face authentication approaches [Erkin et al., 2009] which all require the authentication server to maintain a database of each user's facial images. If using an existing face authentication protocols to authenticate close-up facial images, an attacker will still be able to obtain the user's close-up images after compromising the authentication servers and will be able to impersonate the users later on. In UFace, even if the attacker compromises the authentication servers, he/she will only obtain encrypted facial image feature vectors and cannot reuse them for authentication (detailed security analysis will be presented in Section 8).

As shown in Figure 1.1, the UFace system involves four parties: (1) end users, (2) web service providers, (3) UFace data servers, and (4) UFace key servers. UFace is a third party that hosts two authentication servers to facilitate privacy-preserving authentication between web service providers and end users (through mobile devices). An UFace application will be installed on the user's mobile device. When the user wants to log into a web service (already registered with UFace), the user just needs to take a photo while the application creates a unique feature vector and encrypts it in the background. Once the encrypted feature vector is sent to the UFace data server, the UFace system will then carry out a secure multiparty computation with the UFace key server to authenticate the user with the web service. The technical contributions are summarized as follows:

- UFace is built in a multi-cloud environment and can serve authentication for multiple web services simultaneously. Its authentication protocol prevents the disclosure of any users' data to any party participating in the protocol: (1) web services, (2) UFace data servers, and (3) UFace key servers.

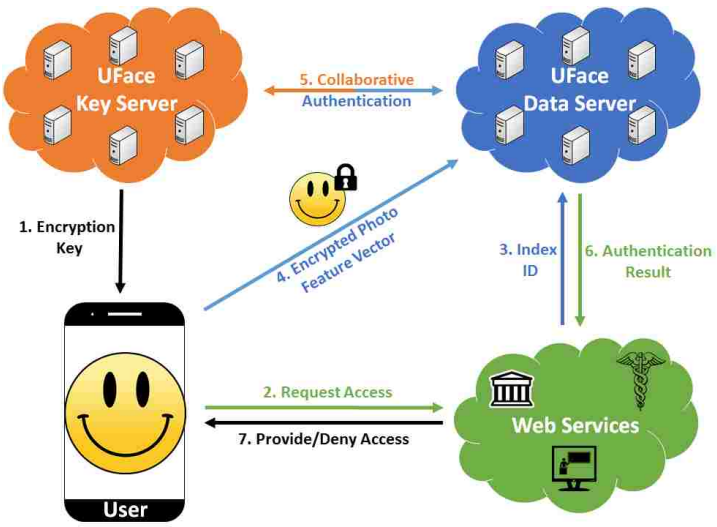


Figure 1.1. UFace system authentication overview

- UFace utilizes an efficient garbled circuit based authentication protocol which allows the two UFace servers to collaboratively conduct feature vector comparison on encrypted data. The encrypted data is based on Paillier's cryptosystem which allows for certain homomorphic encryption operations. The protocol required mapping and integrating of various types of encrypted computations to work alongside garble circuit operations. The overall process needed to be highly efficient so the response time to users would be comparable to authenticating using typical password strings.

- UFace has a mobile application that is capable of efficient photo feature extraction, compaction, and encryption while keeping each user's computational burden to minimum. The development of the mobile application involved multiple challenges that dealt with the limited memory/computing power of these mobile devices along with the need to design a new library for facial feature generation tailored to work on mobile device (or any device with limited power capabilities).

- UFace has been evaluated both theoretically and experimentally. The security analysis shows that UFace is robust against various types of attacks. The experimental results demonstrate that UFace not only can correctly authenticate a user, but also can be done within seconds. This is significantly faster than any existing privacy preserving authentication protocol to date.

The rest of the paper is organized as follows. Section 2 discusses the related works on privacy preserving face authentication and face recognition. Section 3 gives an overview into the tools UFace utilizes while Section 4 provides an overview UFace's phases. The threat model and security goals of UFace are analyzed in Section 5. Then, Section 6 presents the UFace Android application at the user side and Section 7 presents the protocols at server side. Section 8 provides a security analysis of the system and Section 9 reports the performance study. Finally, Section 10 concludes the paper.

2. RELATED WORKS

Biometric authentication is very convenient for end users since it reduces the number of passwords to remember to zero. However, it also raises important privacy concerns since users' biometric data may be known by service providers or authentication servers [Bringer et al., 2013]. One of the earliest attempts towards privacy preservation during biometric authentication is by Erkin et al. [Erkin et al., 2009]. In their setup, the server has a set of photos that it does not want the user to see while the user has his/her own photo that needs to remain hidden from the server. They proposed a secure two-party comparison protocol that allows each user to check if his/her photo matches a photo in the server's database using Eigenfaces while keeping both the user's and the server's photos private to themselves. Later, Sadeghi et al. [Sadeghi et al., 2010] improved the efficiency of the above protocol. Following the similar settings, Osadchy et al. [Osadchy et al., 2010] also proposed a privacy-preserving face detection algorithm - SCiFI - that allows a user to check if his/her photo is in the server's database without knowing the server's database. Huang et al. [Evans et al., 2011] proposed a secure protocol for fingerprint matching while Blanton et al. [Blanton and Gasti, 2011] proposed security protocols for both fingerprints and iris. Recently, Sedenka et al. [Sedenka et al., 2015] employed a similar idea and implemented the privacy-preserving face authentication on smart-phones. However, their system needs more than 10 minutes for a single authentication which is not suitable for real-time applications.

Compared with the aforementioned works, UFace has a totally different setting. The above works all assume that the authentication server has non-encrypted information, i.e., knows the unencrypted content of the each user's biometric data. Unlike their works, the authentication servers in UFace only have access to encrypted feature vectors representing

facial images. This setting significantly enhances privacy preservation and also introduces bigger challenges into the system design even though some of the same techniques are being utilized: garbled circuits and Paillier encryption.

Recently, there are several works which have similar security goals by having only encrypted data at the server side. One is by Blanton and Aliasgari who proposed both a single-server and a multi-server secure protocol to outsource computations of matching iris biometric data records. However, their single-server protocol uses predicate encryption scheme [Katz et al., 2008, Shen et al., 2009] which is not as secure as the additive homomorphic encryption scheme adopted into UFace. Their multi-server protocol leverages a secret sharing scheme [Shamir, 1979] and requires at least three independent servers, whereas UFace only needs two independent servers and is much more efficient. In [Pal et al., 2015], Pal et al. proposed to watermark each user's facial image with fingerprints and then encrypt the watermarked biometric data to protect its privacy from adversaries. Their security protocol is conducted directly by the user and a single server, and hence the user bears a heavy computation workload. In UFace, the computation at the user side is lightweight, which helps conserve smart phone batteries. Another recent related work is by Chun et al. [Chun et al., 2014] who developed a secure protocol that allows an organization to outsource encrypted users' biometric data to the cloud and let the cloud conduct authentication process on fully encrypted data. However, they mainly focus on fingerprint matching, the computation of which is much simpler than that for the face recognition on encrypted data within UFace. Also, their algorithm takes over an hour to authenticate a user, which is not practical in a real world application.

In summary, there have been very limited efforts on privacy preserving face authentication and none of these existing work achieves the same security goal and efficiency as the proposed UFace system.

3. AUTHENTICATION TOOLS

To accomplish authentication between the UFace servers, a few different types of tools are used: facial recognition and secure multiparty computations (SMCs). This section gives an overview of each of these UFace operations to better understand the implementation detailed in Section 7.

3.1. FACE RECOGNITION

Research on facial representation and recognition has been ongoing for numerous years. Two of the earlier methods for representing a person's face were Eigenfaces [Turk and Pentland, 1991] and Fisherfaces [Belhumeur et al., 1997]. Recognition using eigenfaces uses principal component analysis which essential takes a database of images and generates something akin to factors of each face (known as eigenvalues). By assigning weights to each of these eigenfaces, every user's face in the database can be represented by a vector of these weights combined with the associated eigenface. The vector of weights is known as the feature vector. When a user wants to be recognized, first the new image is broken into eigenfaces already designated by the server. Then a new feature vector is generated based off the weights needed to best represent this image. Finally, the server and the user just need to compare the feature vector of the new image against the feature vectors in the database to find a match. Fisherfaces use a similar technique as eigenfaces, except instead of using principal component analysis to generate factors, linear discriminant analysis is used. The problem with these 2 facial recognition tools is that each is sensitive to change in lighting and if more users are added to the database then the eigenfaces/fisherfaces need to be updated.

Later, a more advanced approach was proposed using local binary patterns (LBP) [Ahonen et al., 2004] to generate a feature vector from a photo. Face recognition algorithms using LBP patterns yield a higher accuracy rate under different environments (e.g., different lighting). Another great aspect of LBP is that each user's generated feature vector is not based on any other user's information in the database. Thus, if the database grows, a single user maintains the same feature vector. In UFace, there can be any number of different users for every registered web service and updating each user's feature vector upon every new registration would be time consuming and inefficient. Therefore, the LBP algorithm is employed in this work as the foundation of the proposed encrypted face recognition.

The original LBP method follows a straightforward algorithm of picking an individual pixel and comparing its intensity against the 8 surrounding pixels' intensity (intensity is used since every image is first converted to gray-scale). If a surrounding pixel's intensity was greater than or equal to the intensity of the center pixel then it would be represented by a 1, otherwise it was given a 0. Thus, each of the 8 surrounding pixels are given a single bit of information so the collection of these pixels is a byte which is called a "label" in LBP terms. This label is generated from starting at the pixel above and to the left of the center pixel and then reading each bit in a counter-clockwise pattern.

An example of the basic LBP operation is shown in Figure 3.1. In this example, the center pixel with an intensity value of 92 is the chosen pixel. This pixel is then compared to the surrounding pixels with the function:

$$f(p_i, p_c) = \begin{cases} 0 & \text{if } p_i < p_c \\ 1 & \text{if } p_i \geq p_c \end{cases} \tag{3.1}$$

For the above equation, p_i is the intensity of one of the eight surrounding pixels and p_c is the intensity of the center pixel. The result of of this operation is the encoding on the right hand side of Figure 3.1. After the function has been performed, the generated label for the pixel is obtained by concatenating the 8 bits around the center pixel in a clockwise order

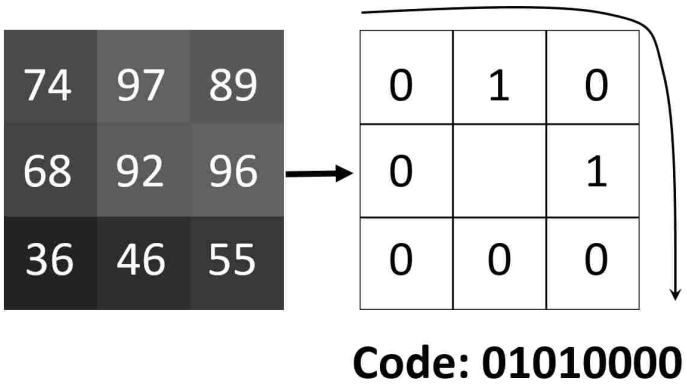


Figure 3.1. An example of computing the LBP for a pixel

starting from the top left pixel. The label encoding for the center pixel with an intensity value of 92 in Figure 3.1 is 01010000. This process is repeated for every pixel in the image. For neighboring pixels that do not exist, such as when the examine pixel is on the edge of the image, bi-linear interpolation can be used to generate values for the imaginary pixels.

Once all the LBP labels have been generated, a histogram is generated using the labels for the individual bins of the histogram. The value for each bin of the histogram is the number occurrences of the specific encoding in the facial image. Since there are 8 bits used to encode a single pixel, there are $2^8 = 256$ possible labels. This means the histogram will be a vector of length 256; however, this can be reduced by using something called uniform labels. A label is considered uniform if there are at most two bit-wise transitions in the encoding (ie. a change from 0 to 1 or vice versa). For an example, the label 01010000 in Figure 3.1 would not be uniform since there are 4 transitions while 00111000 would be uniform since there are only 2 transitions. All non-uniform labels can be placed into a single separate bin. Thus, since there are 58 uniform values between 0 and 2^8 and 1 bin for all non-uniform values. The length of the histogram is then reduced to only needing $n = 59$ bins.

The reason for uniform labels is that some labels occur more frequently than others in an image [Ojala et al., 2002]. For a basic LBP pattern scheme, these uniform labels account for slightly less than 90% of all labels in the image. This means that most pictures can be described by using just these uniform patterns. This is especially true for facial images because there is a gradual change between 2 close pixels; it's much more probable that if 1 pixel's intensity is larger than the center pixel, then the other 2 surrounding pixels will also be larger. In other words, the label for Figure 3.1 is unlikely to occur for pictures of faces.

Now that it's simple enough to create feature vectors from facial images, the next step is to compare them for authentication purposes. To compare two feature vectors of faces, standard histogram comparison techniques can be used such as Histogram Intersection. Given two histograms A and B with n bins, the intersection is defined as:

$$\sum_{i=1}^{n} \min(A_i, B_i) \tag{3.2}$$

The problem is that this formula doesn't provide any percentage on how closely the 2 histograms match. However, this formula can be normalized to by dividing this solution by the number bins in each histogram:

$$H(A, B) = \frac{\sum_{i=1}^{n} \min(A_i, B_i)}{\sum_{i=1}^{n} B_i} \tag{3.3}$$

Now $H(A, B)$ is a percentage showing the closeness of the histograms. This value can then be compared to a threshold value to check if the 2 histograms are related enough to be considered a match.

However, the current LBP scheme doesn't maintain spatial relation of each image. This can be seen from the fact that the label of a pixel at the top left corner of one image can be equivalent to the label at the bottom right corner of a second image. This basic algorithm can be easily expanded upon to maintain spatial information about the image -

divide the image into separate regions and calculate the histogram for each section. This allows for more efficient label comparison since a label found in the top left corner of the image will have a smaller subset of pixels it can possibly match with.

The closeness of 2 histograms can easily be converted to be used with multiple sections with the following modification:

$$H(A, B) = \frac{\sum_{j=1}^{k} \sum_{i=1}^{n} \min(A_{ji}, B_{ji})}{\sum_{j=1}^{k} \sum_{i=1}^{n} B_{ji}} \tag{3.4}$$

In the above equation, j is the region index for the section of the grid and i is the bin index. It should be noted that if histograms are concatenated together, they behave like a single giant histogram for the purposes of histogram intersection. A complete example follows which will be used throughout the length of the paper.

If an image is $N = 256 \times 256 = 65,536$ pixels is separated into $k = 16$ sections (a grid of size 2×2), then each section of the image will contain $128 \times 128 = 16,384$ pixels (since each grid section of the grid is 64×64 pixels). LBP is then done in each of the 4 sections to obtain 4 different histograms. These histograms are then concatenated together in the form $\{H_1 H_2 \ldots H_k\}$ to form the feature vector of the face. Since the number of bins - n - is 59, the large histogram will contain $k \times n = 4 \times 59 = 236$ bins. It should also be noted that the max value in any bin is equal to the number of pixels in a section. The equation below shows how to calculate the number of bits needed per bin:

$$bits_{bin} = log_2 \frac{N}{k} + 1 \tag{3.5}$$

In the previous equation, $bits_{bin}$ is the number of bits needed per bin, N is the total number of pixels in the image and k is the number of sections the image was broken into. Since there are 4,096 pixels in each section, the max value in any bin is 4,096 which means

the number of bits needed to represent each bin value is (at most) $log_2(16,384) + 1 = 15$. Thus, the entire feature vector can be represented by just $236 \times 15 = 3,540$ bits or 0.4321 kB of information.

3.2. SECURE MULTIPARTY COMPUTATION TOOLS

In this paper, privacy/security is closely related to the amount of information disclosed during the execution of a protocol. There are many ways to define information disclosure. To maximize privacy or minimize information disclosure, this paper adopts the security definitions in the literature of SMC first introduced by Yao's millionaire problem for which a provably secure solution was developed [Yao, 1982, 1986]. This was extended to multiparty computations by Goldreich et al. [Goldreich et al., 1987]. It was proved in [Goldreich et al., 1987] that any computation which can be done in polynomial time by a single party can also be done securely by multiple parties. Since then much work has been published for the multiparty case [Canetti, 2000, Ben-David et al., 2008, Katz and Lindell, 2007, Goldreich, 2004]. However, in this paper, security is restricted to the two-party case.

There are two common adversarial models under SMC: semi-honest and malicious. An adversarial model generally specifies what an adversary or attacker is allowed to do during an execution for a security protocol. In the semi-honest model, an attacker (i.e., one of the participating parties) is expected to follow the prescribed steps of a protocol. However, the attacker can compute any additional information based on his or her private input, output and messages received during an execution of a secure protocol. As a result, whatever can be inferred from the private input and output of an attacker is not considered as a privacy violation. An adversary in the semi-honest model can be treated as a passive attacker; on the other hand, an adversary in the malicious model can be treated as an active attacker who can arbitrarily diverge from the normal execution of a protocol.

UFace's proposed protocols are secure under the SMC definition, and construction of these protocols are based on two secure primitives: garbled circuit and additive homomorphic encryption. Garbled circuit is a generic tool to securely implement any two-party polynomially-bounded distributed functionality [Huang et al., 2011] based on the fact that the functionality can be represented by a Boolean circuit which can be evaluated securely. In addition, UFace adopt the Paillier cryptosystem, an additive homomorphic and probabilistic asymmetric encryption scheme [Paillier, 1999]. The encryption scheme is semantically secure [Goldwasser and Micali, 1984], i.e., given a set of ciphertexts, an adversary cannot deduce any information about the plaintext. The following sections will go over each tool in more detail.

3.2.1. Garbled Circuits. The goal of garbled circuits is to provide a secure computation for multiple parties to compute a function in which no party learns the inputs of any single party. For a two party computation, the party that generates the circuits encodes their own inputs into the function that needs to be computed. This creates a function that instead of needing 2 inputs from each party, only needs the input from the other party. The next step is to generate a circuit based off this new function.

A circuit can be considered a sequence of boolean gates which are able to compute the function that needs to be computed. Once the circuit is generated, the next step is to obfuscate the inputs with 2 random keys for each wire of the circuit. These 2 keys (K_i^0 and K_i^1) correspond to wire i being assigned either a 0 or 1. This "garbled circuit" is then sent to the other party where the remaining calculations are computed.

At this point, the second party has the garbled circuit, but no way of computing the function since his/her own inputs are missing. These inputs are obtained through oblivious transfer from the first party. The circuit is then evaluated securely. Since each wire is obfuscated with 2 random keys, the party calculating the garbled circuit cannot determine the inputs of the other party based on the output.

At no point are either party's inputs leaked to any other party. Obtained from the garbled circuit is the proper solution to the function that is evaluated. For more thorough details on garbled circuits, see [Yao, 1986].

3.2.2. Paillier Cryptosystem. This type of cryptosystem is known as an additive homomorphic public-key encryption scheme. In public-key cryptosystems, a public key is used to encrypt a piece of information; however, to decrypt the ciphertext, a second (private) key is needed. In this setting, an authenticator generates both keys and distributes the public key while keeping the private key secure. Then, when a message needs to be sent to the authenticator, it's first encrypted using the public key and then decrypted once it reaches the destination. The only person capable of decrypting the ciphertext is the authenticator with the private key. A great aspect about public-key schemes is that if an attacker obtains the encrypted version of the message, then he/she will not be able to deduce anything about the actual message since the attacker doesn't have a way of decrypting the information. Another great aspect is that this allows one authenticator to talk with many different parties without generating multiple keys between each party - the one decryption key can decrypt any message sent from any party.

However, public-key cryptosystems are not perfect and can be subject to attack. One example is that if an attacker can obtain a ciphertext, then he/she can determine the plaintext information by taking all possible encryptions of every message in the domain space until an encryption matches the ciphertext. If the domain space is small enough, then this could take a short amount of time. So a way of fixing that is to ensure that the ciphertext is semantically secure, which is something Paillier's cryptosystem does. Being semantically secure means that any ciphertext will reveal nothing about the original message. To ensure that the system is semantically secure, each encryption needs to map to a separate ciphertext (even 2 encryptions of the same message). In Paillier's cryptosystem, this is done by introducing a random value during the encryption of a message.

The second aspect of this cryptosystem is that it is additively homorphic. This means that it's possible to compute the encrypted sum of encrypted messages ($E(m_1) \cdot E(m_2) \equiv E(m_1 + m_2)$) and the encrypted multiplication of encrypted messages ($E(m)^k \equiv E(k \times m)$). Essentially, it allows for operations to be done on a message without needing to decrypt a message first, which is utilized in the operation of UFace's garbled circuit protocol. For more thorough details on Paillier's encryption scheme, see [Paillier, 1999].

4. SYSTEM OVERVIEW

UFace is designed as a privacy preserving face authentication framework to prevent web service providers from gaining access to user's facial images or their respective feature vectors. To accomplish this, UFace serves as the middle man between multiple web service providers and users. As shown in Figure 1.1, there are 4 entities involved in the system: (1) end users, (2) web service providers, (3) UFace data servers, and (4) UFace key servers. The UFace data servers store all users' encrypted feature vectors while the UFace key servers manages the key capable of decrypting this information. However, the 2 server clouds never collude about their contents and execute a secure multiparty computation to authenticate users. This design follows the spirit of "separation of duty" to achieve privacy preservation. For the remainder of the paper, each UFace server cloud will be considered to be 1 single server for easy illustration of the main ideas. UFace is comprised of 2 main phases of operation: (1) Registration and (2) Authentication.

Before going any further, it should be stressed that all data stored on the UFace data server is encrypted using Paillier's cryptosystem. This data is the encryption of a feature vector representation of the user's face (not the encrypted version of an image). The encryption is done before being transmitted on a secure line. The data is therefore as secure as Paillier's cryptosystem. Secondly, the security of each of the UFace servers (data and key server) should be using state of the art security measures so attackers cannot easily break into the systems. This paper's focus is not on the security of the servers, but instead focuses on the security of the protocol to compare encrypted feature vectors.

4.1. REGISTRATION

To register with a web service, a user just needs to install the UFace mobile client. The user will select a web service that is already registered with UFace and create a unique $UserID$ for the web service. To finish registration, the user only needs to take a close-up photo of their face. After the user takes a photo, the app executes the LBP algorithm to generate a feature vector on the photo and then encrypts the feature vector before sending this information off to the UFace data server for authentication. This is all done in the background and happens immediately after the user takes a photo. The UFace data server receives the encrypted data and stores the information at a specific location - $IndexID$ - for that user (which is shared with the web service provider and the user). At this point, the user is registered with the web service and the data server contains an encrypted feature vector for the user to compare against for authentication.

4.2. AUTHENTICATION

To begin authentication for a web service, the user only needs to select the registered web service and take a close-up image of the user's face. While this is occurring, the app will send the $UserID$ to the web service so the service knows a specific user is attempting to log into their system. The web service will then forward the associated $IndexID$ to the UFace data server so the data server knows that an authentication attempt will begin for the specified user. Meanwhile, the mobile app has executed the LBP algorithm, generated a feature vector from the taken photo and encrypted this feature vector (same as in the registration phase). Finally, once a message is received from the web service stating that authentication may begin, the app sends the encrypted feature vector and its $IndexID$ to the UFace data server. The application does not send the $UserID$ so that the UFace data server can't map any encrypted feature vectors directly to a user. Instead, there is a layer of obscurity to who the user is since it's just a number which only the user and web service

can reference. Also, the $IndexID$ the user contains is first verified with the $IndexID$ the web service contains, if these IDs are different then the UFace data server won't execute the authentication protocol since it would be waiting on a different $IndexID$. Once the data is successfully sent to the UFace data server, both the data and key servers collaboratively conduct a secure protocol to determine the comparison result between the sent encrypted feature vector and the one stored on the UFace data server at the location determined by the $IndexID$. The secure protocol ensures that each server's information remains confidential to each server, so even though the UFace key server has the key to decrypt all messages, it never obtains information about the user's biometric data nor does the UFace data server ever get the decryption key. The result is then sent to the web service which then forwards the response to the user. Figure 1.1 provides an outline for how authentication works within the UFace system (numbers show order of information travel).

5. THREAT MODEL AND SECURITY GOALS

In UFace system, the commonly used semi-honest security model is adopted which assumes that each participating party will follow the protocol but may try to learn additional information by exploring the information available to them [Goldreich, 2004]. In general, secure protocols under the semi-honest model are more efficient than those under the malicious adversary model, and almost all practical SMC protocols proposed in the literature [Canetti, 2000, Ben-David et al., 2008, Katz and Lindell, 2007, Goldreich, 2004] are secure under the semi-honest model. In this model, the participating parties will not collude, these parties are the UFace data and key servers. This can be guaranteed by deploying the two servers in two different clouds such as Amazon and Microsoft whereby the two big cloud service providers have no incentive to collude.

The security goal of UFace is to keep users' authentication information fully private, which includes the following aspects:

- Users do not need to reveal the actual content of their facial images or plaintext feature vectors to any party during the authentication process.
- Web service providers can safely outsource the authentication process to UFace without violating users' privacy concerns regarding their biometric data that have been used for authentication.
- UFace authentication servers perform authentication on encrypted randomized data.
- UFace authentication servers can not connect any encrypted information back to any specific user.

In the following sections, the UFace application at the client side and the privacy preserving protocols at the server side will be presented, respectively. Afterwards, specific security issues vulnerabilities will be examined in Section 8.

6. UFACE MOBILE APPLICATION

The UFace Android application consists of two modules: Web Service Access and $UFacePass$ Generation. The first refers to the 2 phases of UFace: registration and authentication. The second module explains how the encrypted feature vector is generated. Each will be discussed in the following sections.

6.1. WEB SERVICE ACCESS

To start logging into a web service securely, users only need to register with a web service provider. Upon start-up, the app retrieves a public key (PK) used for encryption and displays a blank screen with a "+" icon as shown in Figure 6.1a. The reason PK is downloaded every time the app starts is for 2 reasons: (1) so the user always has the correct encryption key in case the cryptographic scheme changes and (2) as a check to make sure the user can communicate with the UFace system.

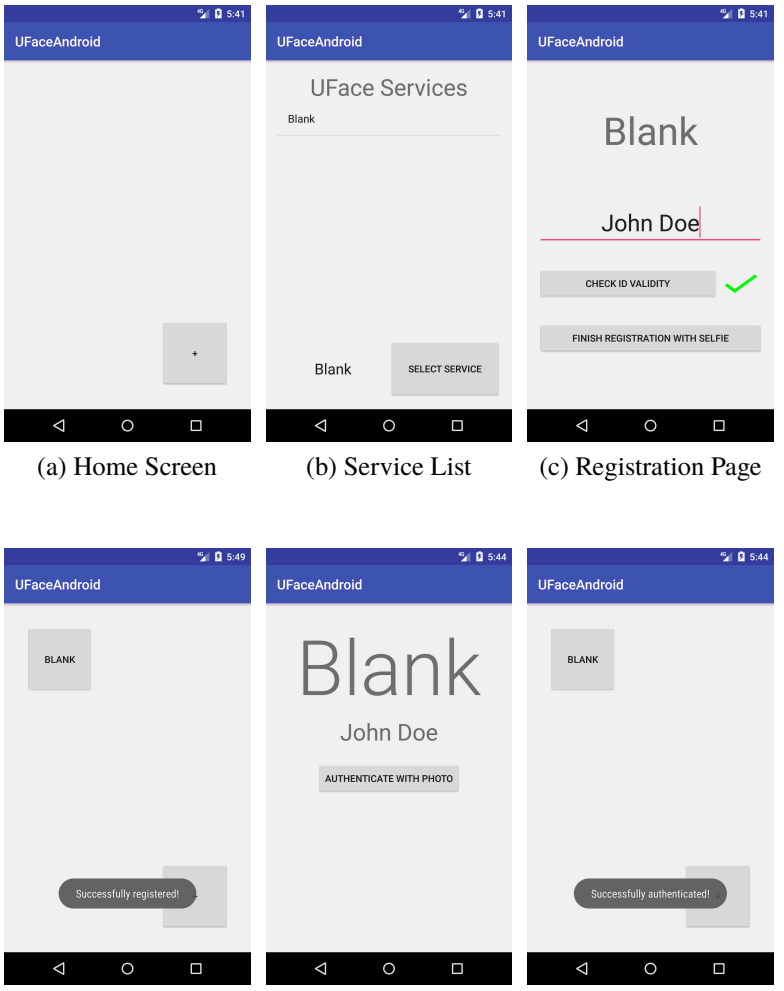
By clicking on the "+" icon, a new window will display a list of web services which use the UFace system for authentication (obtained from the UFace data server automatically upon screen initialization) - seen in Figure 6.1b. Once the user selects a web service, the user will see a registration page to create a unique "$UserID$" for the web service. The $UserID$ will be sent to the web service provider to verify the uniqueness (as shown in Figure 6.1c). If the $UserID$ is unique, the web service provider will return an "$IndexID$" to the user. This $IndexID$ was generated from the UFace data server to indicate the location where the user's encrypted feature vector ("$UFacePass$") will be stored. The $IndexID$ also prevents the data server from knowing the user's actual $UserID$ which prevents the UFace system from knowing exactly whose each $UFacePass$ belongs to. By creating different $IndexID$s, it would be easy to extend the current system to accommodate multiple user devices registered for the same web service.

Finally, the user just needs to take a close-up facial photo to finish registration. This photo needs to be taken extremely close up so that the entire face takes up most of the space in the photo. The further the photo is taken from the user's face the greater risk the user is to specific attacks (see 8.2 under "impersonation attack" for more information). The photo is used to generate the $UFacePass$ (the algorithm is presented in Section 6.2) and it's sent to the UFace data server along with the $IndexID$. The $UserID$, $IndexID$, and web service information are stored in the app if registration is successful.

After successful registration, the main page of the app will now show a list of icons representing registered web services as shown in Figure 6.1d. After selecting a web service, the user will be required to take a close-up facial photo (Figure 6.1e) to generate a new $UFacePass$. The app will send the $UserID$ to the web service provider to begin authentication. After the web service receives notification from the UFace data server, it will then respond to the user by saying a new authentication attempt may begin. The app will then send the $UFacePass$ and the $IndexID$ to the UFace data server for authentication. After UFace executes the authentication protocol, the result will then be send to the web service and forwarded to the client (there is no response from the UFace system directly). If access is granted, the user will be directed to the account for that web service (which Figure 6.1f shows a successful authentication attempt).

6.2. UFACEPASS GENERATION

The most important feature of the UFace Android app is the $UFacePass$ generator. This converts the user's close-up photo into an encrypted feature vector efficiently. The overall process can be seen in Figure 6.2. However, there are 3 main steps to creating the $UFacePass$: (1) feature vector generation, (2) feature vector manipulation and (2) feature vector encryption.



(a) Home Screen (b) Service List (c) Registration Page

(d) Registration Result (e) Authentication Page (f) Authentication Result

Figure 6.1. Snapshots of the UFace mobile application

6.2.1. Feature Vector Generation. When the user is asked to take a facial image, the image needs to be a close-up photo that fills the whole screen of the smartphone. Every phone takes different quality pictures, thus the image needs to go through a small pre-processing phase to create images that can be used in the algorithm. The only requirement is that every image needs to contain the same number of pixels, so the image will be shrunk down if the picture is too large and it will fail if the image is too small. The size of the image was determined in the experimental section of this paper (Section 9) based on the speed

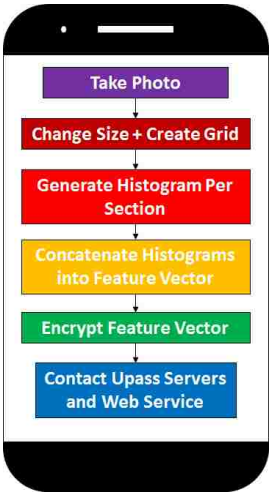


Figure 6.2. UPass generation

needed to accomplish authentication in real-time and for making sure the authentication was accurate in its results. The best pixel count was determined to be $N = 65,536$ which refers to a square image of size 256×256 pixels. It should be noted that camera's on today's cellphones are measured in Megapixels; thus, it'll be incredibly difficult to find a camera today unable to take a picture containing at least 65,536 pixels.

Once the image is captured and properly sized, the LBP algorithm is executed to create a feature vector. As introduced in Section 3.1, a LBP photo feature vector can be represented in the form given by Definition 1.

Definition 1 (Feature Vector) *Let p be a photo. Its feature vector F_p is represented as $F_p = \langle \overrightarrow{H_1}, \ldots, \overrightarrow{H_k} \rangle$, where $\overrightarrow{H_i} = \langle b_1, \ldots, b_{59} \rangle$ $(1 \leq i \leq k)$.*

First, the image is divided into equal sections. The number of sections was determined to be $k = 16$ in the experimental section of this paper to optimize speed and correctness (Section 9). In each of these k sections, the LBP algorithm is executed to obtain a histogram - H_k. This histogram is the count of 59 labels: 58 uniform labels and 1 label for all other values (see Section 3.1).

The full feature vector is simply the concatenation of these k histograms. In the end the feature vector F_p is $\langle \overrightarrow{H_1}, \ldots, \overrightarrow{H_k} \rangle$. Now that the feature vector is created, it needs to be encrypted in such a way that the time taken does not negatively impact the ability for this application to run in real-time.

6.2.2. Feature Vector Manipulation. To speed up encryption time, the feature vector needs to be manipulated. Let's first examine the theoretical case that was first presented in Section 3.1. It should be noted that all the values presented are the values used in the final application; the reasons for each of these values are determined in the experimental section (Section 9).

For an image of size $N = 256 \times 256 = 65,536$ pixels broken into $k = 4$ sections, the number of bins that need to be encrypted is $b = 4 \times 59 = 236$. Another fact is that the max pixel count in each section would be $\lceil \frac{65,536}{4} \rceil = 16,384$ pixels. Thus, the max number of bits needed to represent any individual bin in the feature vector (and more specifically each section) is $b = \log_2 16,384 + 1 = 15$ bits. The final piece of information is the number of bits needed to do a single encryption. Paillier encryption in UFace uses 1,024 bits to encrypt each bin, which means that if each bin is encrypted separately, there would be $1,024 - 15 = 1,009$ bits of wasted information with every encryption. The total size of the feature vector would be $236 \times 1,024 = 241,664$ bits or 29.5 kB with $236 \times 1,009 = 238,124$ wasted bits which is roughly 29.0679 kB.

To improve this, sequential bin values will be concatenated together into 1 single value which is then encrypted. This would mean that $\lfloor \frac{1,024}{15} \rfloor = 68$ bins can be used in 1 encryption with just 1024 modulo 15 = 4 extra bits instead of 1,009 bits for every 68 bins. Instead of 236 encryptions with 29.5 KB of data being sent, there is only $\lceil \frac{236}{78} \rceil = 4$ encryptions with $4 \times 1,024 = 4,816$ bits or 0.5879 KB. The amount of wasted space is just $4 \times 4 = 16$ bits or 0.0020 kB. This new size is roughly 2% of the original size.

UFace's process of reducing the size is done through 2 different algorithms. The first converts the integer matrix of size 4×59, which was generated by LBP, into an integer matrix of size 4×68, which is the 4 encryptions of the 68 sequential integer values. This algorithm can be seen in Algorithm 1.

Algorithm 1 Split Feature Vector for Encryption

Require: $numEnc$ be the number of encryptions needed
Require: $numBins$ be the number of bins that can be concatenated together per encryption
Require: n be the number of bins per section of grid
Require: k be the number of sections in the grid
Require: $hist[k][n]$ be the input histogram
Require: $splitFV$ be the returned array

```
splitFV ← int[numEnc][numBins]
in ← 0
out ← 0
for i ∈ {1, ..., numEnc} do
    for k ∈ {1, ..., numBins} do
        splitFV[i][k] ← hist[out][in]
        in ← in + 1
        if in = n then
            out ← out + 1
            in ← 0
            if out = k then
                return splitFV
            end if
        end if
    end for
end for
```

The second algorithm takes this intermediate matrix and converts it to a byte matrix of size 4×128. The 128 is in reference to the number of bytes that can be encrypted at once (ie. $128 \times 8 = 1024$ - which is the size used in this Paillier cryptosystem). The process can be seen in Algorithm 2, but overall it takes the 68 sequential integer values and breaks them into their corresponding bit value. These bits are then concatenated into sequences of 8 to be placed into the matrix of bytes. The first 4 bits of each of the 4 byte arrays are set to 0 since they are wasted space for each encryption. Figure 6.3 visually shows how each array of integers becomes an array of bytes.

Algorithm 2 Create Byte Feature Vector

Require: $numBytes$ be the number of bytes used per encryption
Require: $numWaste$ be the number of wasted bits per encryption
Require: $numBits$ be the number of bits used to represent each value in any bin
Require: $numEnc$ be the number of encryptions needed
Require: $numBins$ be the number of bins that can be concatenated together per encryption
Require: n be the number of bins per section of grid
Require: k be the number of sections in the grid
Require: $splitFV[numEnc][numBins]$ be the input histogram
Require: $byteFV$ be the returned array

$byteFV \leftarrow byte[numEnc][numBytes]$
for $i \in \{1, \ldots, numEnc\}$ **do**
  $intArray \leftarrow splitFV[i]$
  $leftZeroBits \leftarrow numWaste$
  $next \leftarrow 0x00$
  $index \leftarrow 0$
  $bitsUsed \leftarrow 0$
  for $k \in \{1, \ldots, length(intArray)\}$ **do**
    $bitsNeeded \leftarrow numBits$
    $curVal \leftarrow intArray[k]$
    while $bitsNeeded > 0$ **do**
      if $leftZeroBits >= 8$ **then**
        $byteFV[i][index++] \leftarrow next$
        $next \leftarrow 0x00$
        $leftZeroBits \leftarrow leftZeroBits - 8$
      else
        if $bitsNeeded >= (8 - leftZeroBits - bitsUsed)$ **then**
          $shiftRight \leftarrow bitsNeeded + leftZeroBits + bitsUsed - 8$
          $next \leftarrow ((curVal >>> shiftRight) \& 0xFF) | next$
          $bitsNeeded \leftarrow bitsNeeded - (8 - leftZeroBits - bitsUsed)$
          $bitsUsed \leftarrow 0$
          $byteFV[i][index++] \leftarrow next$
          $next \leftarrow 0x00$
          $leftZeroBytes \leftarrow 0$
        else
          $shiftLeft \leftarrow 8 - bitsNeeded$
          $next \leftarrow ((curVal << shiftLeft) \& 0xFF) | next$
          $bitsUsed \leftarrow bitsNeeded$
          $bitsNeeded \leftarrow 0$
        end if
      end if
    end while
  end for
end for
return $byteFV$

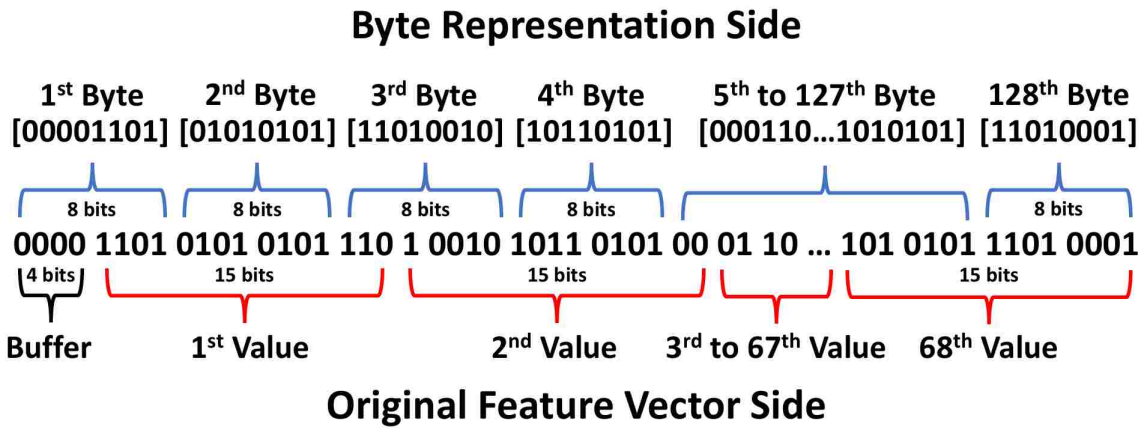


Figure 6.3. Feature vector compaction

Finally, these 4 byte arrays are then converted to a single array of 4 BigInteger values. These BigInteger values are what is encrypted with Paillier's cryptosystem to become the $UFacePass$.

6.2.3. Feature Vector Encryption. At this point, a reduced feature vector is obtained which just needs to be encrypted. The encryption algorithm simply iterates through each of the 4 BigInteger values and encrypts each one using the public key obtained from the UFace key server for Paillier's cryptosystem (see Section 3.2.2). Once the feature vector is fully encrypted, it and the $IndexID$ are sent off to the UFace data server for either registration or authentication. From this point on, the client is done with any calculation and is now just awaiting a response from the web service.

7. UFACE SERVER

UFace system utilizes a data server and a key server for authentication, which are located in two different clouds to avoid potential collusion. The data server is used for storing the encrypted $UFacePass$ for each user, i.e., the encrypted feature vector. Each $UFacePass$ is stored at a specific index which can be referenced by each user's $IndexID$; however, the data server does not know anything about a specific user nor any user's $UserID$. The key server is used for maintaining the public key (PK) that can decrypt a user's $UFacePass$. In what follows, the registration and authentication phases, respectively, are presented.

7.1. REGISTRATION

The registration phase is fast since there is no need for computation on the server side. Figure 7.1 illustrates the main communication between all parties during registration - note that all communications are through secure channels. Before registration begins, the UFace key server transfers PK to the user (this is done every single time the application starts). Then the user (i.e., the UFace mobile application) sends a new $UserID$ to the web service provider. Once the web service provider verifies the uniqueness of the $UserID$ it informs the UFace data server to prepare an $IndexID$ for a new user. The UFace data server will send the $IndexID$ back to the web service provider which will forward it to the user. While this is happening, the user is taking a close up facial photo. The mobile application then generates a feature vector from the LBP algorithm, manipulates the data for better encryption, and then encrypts the feature vector using PK to obtain $UFacePass$. Upon receiving the $IndexID$ (and after generating $UFacePass$), the user will then send the $IndexID$ and $UFacePass$ to the data server. The mobile application will also store the $IndexID$ and $UserID$ for the specific web service so the user does not need to remember

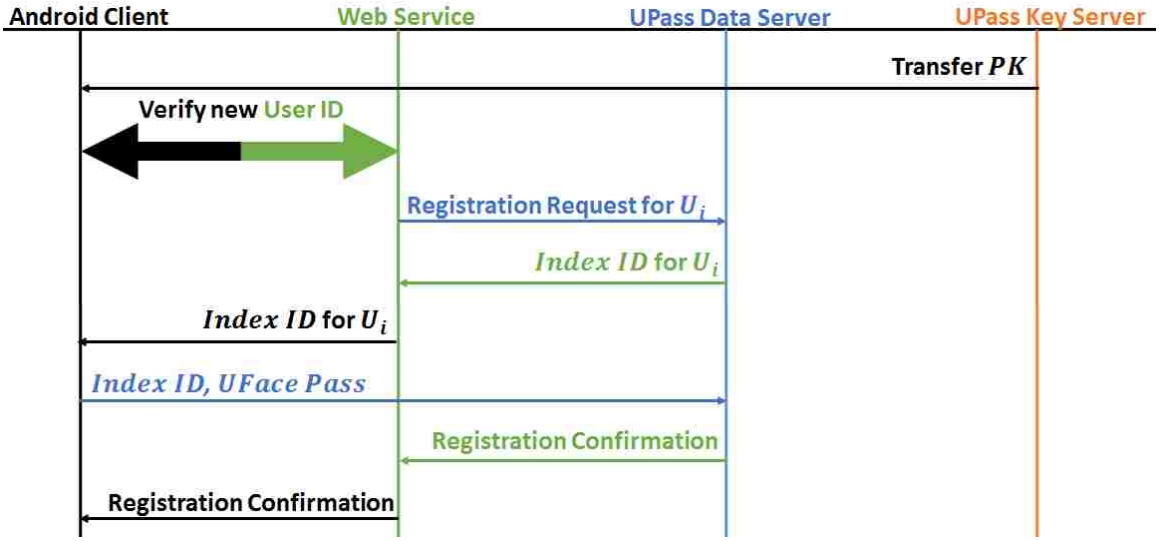


Figure 7.1. Registration protocol

anything. The data server will store the received user information at the location provided from the $IndexID$ and inform the web service of a successful registration which then informs the user. The data server never sees the user's real $UserID$ and the $UFacePass$ is encrypted with the key stored on the UFace key server so it cannot decrypt the information.

7.2. PRIVACY-PRESERVING AUTHENTICATION

After registration, an user can log into the web service by simply selecting the web service on the mobile app and taking a close-up facial image; this offers a similar user experience to typical web service authentication pages - except with no need to remember a password. Again, all communication is conducted through secure channels. The authentication protocol is outlined in Figure 7.2.

First, the user (i.e., UFace app) obtains PK from the UFace key server. Then the user sends the $UserID$ and $IndexID$ to the web service provider who will locate the $IndexID$ of this user compare the sent ID with the stored one. If the IDs match, then the web service will forward it to the UFace data server to establish an authentication request.

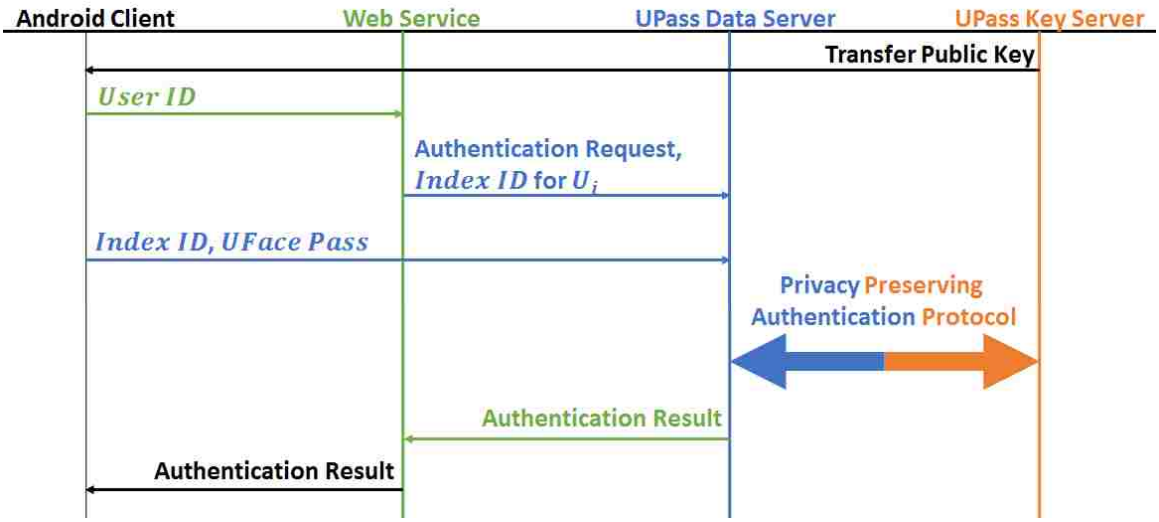


Figure 7.2. Authentication protocol

Then, the user will send his $IndexID$ and a newly generated $UFacePass$ to the data server. Upon receiving the user's authentication information, the data server will initiate a privacy-preserving authentication protocol with the UFace key server to jointly compare the received $UFacePass$ with the user's registered $UFacePass$. The proposed privacy preserving authentication protocol is built with garbled circuits, and ensures that neither the data server nor the key server will see the user's biometric information. The details of the privacy-preserving authentication protocol is presented in the next section (Section 7.2).

At the end of the privacy-preserving authentication protocol, the data server will return the result to the web service provider. If the result is a match, then the user's $UFacePass$ matches the registered stored information over a specific threshold. The threshold value was determined experimentally to be 92.78% similar; the reasoning can be seen in Section 9. Based on the result, the web service provider will grant/deny access to the user accordingly.

Now it's time to describe the privacy-preserving authentication protocol between the data server and the key server. This protocol leverages the garbled circuit techniques [Huang et al., 2011] because garbled circuits have been proven to be efficient for small

functionality represented by a boolean circuit and efficiency is a key requirement to achieve real-time authentication. In the following discussion, the two encrypted feature vectors are denoted as $E(F_1)$ and $E(F_2)$, whereby $E(F_1)$ refers to the feature vector that has been stored with the data server at the registration phase and $E(F_2)$ refers to the feature vector received with the authentication request. F_1 refers to the unencrypted version of the feature vector stored on the data server while F_2 refers to the enencrypted version of the feature vector send to the data server for authentication. It should be noted that neither F_1 nor F_2 are ever able to be viewed by any party participating in the authentication protocol (UFace data server or key server). Table 7.1 summarizes the notations used in the discussion.

Table 7.1. Notations

F_1	Feature vector 1
F_2	Feature vector 2
$E(F_1)$	Encrypted feature vector 1
$E(F_2)$	Encrypted feature vector 2
R_1	Value used to randomize F_1
R_2	Value used to randomize F_2
F_{1R}	Randomized F_1
F_{2R}	Randomized F_2

The design challenge is that garbled circuits can only handle plain text efficiently, but the feature vectors are all encrypted. In order to preserve efficiency, decrypted data needs to be fed into the garbled circuits. If the encrypted feature vector are sent directly to the key server for decryption, the key server will then know the user's feature vector information and hence violate the privacy preservation goal. To prevent this, the data server add random values R_1 and R_2 to feature vectors $E(F_1)$ and $E(F_2)$ respectively using the Paillier cryptosystem's additive homomorphic property, and then send the randomized feature vectors to the key server. Now the key server can decrypt the randomized feature vectors without learning about the user's information. These decrypted randomized feature vectors are the main input to the garbled circuit. The comparison results of the pair of randomized feature vectors would be the same as the original pair since the random values

$R_!$ and R_2 will be removed from F_1 and F_2 within the garbled circuit. Thus, the garbled circuit will end up just comparing F_1 against F_2 and this will still be able to tell whether F_1 matches F_2.

Specifically, the data server sends the following information to the garbled circuit: R_1, R_2, R_{bit} and Th, whereby R_{bit} is a single bit used to hide the circuits outcome from the key server, and Th is an adjustable threshold value for face recognition accuracy. Then, the key server feeds the decrypted randomized feature vectors F_{1R} and F_{2R} to the garbled circuits. It is worth noting that at a high level view there are only these five inputs total, but in practice, there are multiple. Since each input is limited to the same bit size as the encryption key, multiple inputs are needed to represent each feature vector. For ease of understanding, each feature vector will be considered as one input.

The main steps of using a garbled circuit to compare two encrypted feature vectors are outlined in Algorithm 3. At steps 1 and 2, the random values are subtracted from the randomized feature vectors. To speed up the process, the random values are inverted when provided to the garbled circuit; this allows adding to be done instead of subtraction. The result will overflow the value to have the effect of modular division since the overflow bit is lost. This functions identically to subtraction, but faster. For clarity however, the random values are stated as being subtracted.

Algorithm 3 GCParser Circuit Code

Require: Data_Server: R_1, R_2, R_{bit}, and Th; Key_Server: F_{1R} and F_{2R}

1: Subtract R_1 from F_{1R}
2: Subtract R_2 from F_{2R}
3: Now F_1 and F_2 are in the circuit
4: Each bin b_{1i} of F_1 is isolated
5: Each bin b_{2i} of F_2 is isolated
6: **for** $i \leftarrow 1$ to $k \times n$ **do**
7: $\quad min_x = \min(b_{1i}, b_{2i})$
8: **end for**
9: $intersection = \sum_{x=1}^{k \times n} min_x$
10: $pass = intersection \geq Th$
11: $result = pass \ \oplus R_{bit}$

As a result of the first two steps, the circuit will have the original non-randomized, decrypted feature vectors F_1 and F_2. Conceptually each feature vector is a matrix. For the garbled circuit, each feature vector is the individual bins $b_{j,i}$ (where $j \in \{1, 2\}$ and $i \in \{1, 2, \ldots, n \times k\}$) from each $\overrightarrow{v}$ concatenated end to end. The value j represents 1 specific feature vector, k represents the number of sections an image is separated into while n is the number of bins for the histogram created in each section, and $\overrightarrow{v}$ is the histogram generated for each section. Thus, each feature vector has the internal appearance of $b_{j,1}b_{j,2}b_{j,3}\ldots b_{j,k\times n}$. To further process these individual bins, the bins have to be separated, which is the main purpose of this step. Since each bin has a known bit size, the bins can be separated by linearly traversing the feature vector and isolating every block of the bit size. Once each bin is isolated, the intersection calculations begin. As shown in step 3 of the algorithm, by linearly walking through all the bins, the minimum between the corresponding bins of each feature vector is calculated.

In the next step, the sum of the minimums from each $\overrightarrow{v}$ are calculated which is done in parallel to improve efficiency. These minimums are added together to obtain the final sum. Based on the final sum of the two feature vectors F_1 and F_2, the 2 feature vectors can now be evaluated for similarity. To determine similarity, the final sum needs to be compared against a threshold. For UFace, the threshold value is set to 0.9278 based on the results obtained in Section 9, which means the 2 feature vectors need to be at least 92.78% similar.

The following explains the threshold checking. When calculating the local binary patterns, a binary pattern (label) is generated for each pixel in the image. Thus, when histogram intersection is performed between identical feature vectors, the resulting value will simply be the total number of pixels in the image. If the requirement is a 92.78% match for authentication, the threshold value Th can be set to $0.9278 \times N$ where N is the number of pixels. Inside the circuit resulting intersection can be compared to Th and if the intersection is greater than or equal a 1 is returned, otherwise a 0 is returned. In this way, the expensive secure division operation do not need to be performed.

Finally, to prevent the key server from knowing the authentication result, the result is XOR'ed with R_{bit}. What the key server will see is a single bit that has a 50% chance of indicating "match" or "unmatch". The data server can perform the XOR operation on the result to receive the actual result.

An overview of the protocol is given in Algorithm 4. When the protocol begins execution on the server side, it is assumed that both servers have a copy of the garbled circuit. The operations of the circuit do not change with each execution, so the circuit only needs to be constructed upon initial server setup. However, when GCParser runs the circuit file, the circuit will be uniquely garbled. Therefore, with each execution of the protocol, a different garbled circuit is produced. Should either server attempt to change the circuit file, GCParser will abort operations due to these differences.

It is also assumed that a threshold value Th has been set during the setup, and the user has registered a feature vector on the system. After the client has submitted a feature vector for comparison, the authentication protocol can begin. The following analysis is a complete walk-through of Algorithm 4.

At step 1, the data server generates the random values R_1 and R_2 that are used to randomized the feature vectors. Because the feature vectors will be represented by x number of bins, R_1 will also be represented by x number of random values; one random value for each bin. For clarity, each feature vector and random number is referred to as a single entity. To randomize the feature vectors, the random values must also be encrypted as seen in the sub-steps. The encryption is performed using the public key generated by the key server, which is the same key used to encrypt the feature vectors.

Once the random values are encrypted, the homomorphic additive property of Paillier's encryption scheme is used to add a feature vector and a random number by multiplying the two encrypted values together. In preparation for the execution of the

garbled circuit, the data server also generates the random bit R_{bit} at step 1(c) and proceeds to construct its input file. At the end of step 1, the feature vectors can safely be sent to the key server.

At step 2, the key server's operations begin upon receiving $E(F_{1R})$ and $E(F_{2R})$. At step 2(a), these values are decrypted using the key server's private key. Because the values have been randomized, the key server never actually sees the original plaintext values. These two values will be the only inputs that the key server provides to the garbled circuit and as such directly writes the values to its circuit input file. At the end of step 2, the key server starts the server component of GCParser using the circuit file and its circuit input file. What is not shown in Algorithm 4 is that the key server will send a message to the data server telling it that GCParser has been started. This just ensures the proper ordering of steps as displayed in the algorithm.

When the data server begins execution of the garbled circuit at step 3, it executes GCParser as a client that connects the server instance running on the key server. It must provide as input the IP address of the key server, its input file, and a copy of the circuit. As mentioned earlier if the circuit file is different from the one used by the key server, the operation will fail.

At step 4, both servers collaboratively run the GCParser which handles the computations obliviously or securely, so even though the key server is running the garbled circuit, it never sees the plaintext version of the data server's input. After the secure evaluation of the circuit, GCParser will write the circuit output to a file on both servers. These files contain a result that is a randomized bit. For a more in depth view of the circuit, refer to Section 7.2.

With the final result returned from the garbled circuit, the data server performs an XOR operation between the circuit result and R_{bit}. This will undo the XOR operation that was performed within the garbled circuit and provide the result of the threshold comparison.

If this value is 1, then the intersection was greater than the threshold, and is zero otherwise. At this point, authentication has been performed, and the data server can communicate with the user about the results.

Algorithm 4 Overall Protocol Between Authentication Servers

Require: Data_Server: $E(F_1)$, $E(F_2)$ and Th

1: Data_Server:

   (a) Randomly generate R_1 and R_2, and encrypt them to produce $E(R_1)$ and $E(R_2)$

   (b) Calculate $E(F_{1R}) = E(F_1 + R_1) = E(F_1) * E(R_1)$ and $E(F_{2R}) = E(F_2 + R_2) = E(F_2) * E(R_2)$

   (c) Generate a random bit R_{bit} and produce a garbled circuit input file using R_1, R_2, Th, and R_{bit}

   (d) Send $E(F_{1R})$ and $E(F_{2R})$ to Key_Server

2: Key_Server:

   (a) Decrypt $E(F_{1R})$ and $E(F_{1R})$ and write the values to a garbled circuit input file

   (b) Start GCParser as server using its input file and the circuit

3: Data_Server:

   (a) Use GCParser to connect to the circuit running on Key_Server as a client using its input file

4: Data_Server and Key_Server:

   (a) Using GCParser, collaboratively evaluate the garbled circuit, and the evaluation result returns to both parties

5: Data_Server:

   (a) Perform XOR operation with the result and R_{bit}

   (b) Inform the authentication result to the web service: If the XOR result is a 1, authentication passed, else it failed

8. SECURITY ANALYSIS

UFace does not leak any user's biometric information to the data server, the key server or the web service provider. This is because our approach follows the security definitions in the literature of SMC. As a result, our proposed protocol can be easily proved to be secure under the semi-honest model of SMC by using the simulation argument [Goldreich, 2004].

8.1. SECURITY PROOF

In our work, privacy/security is determined by the amount of information disclosed during the execution of the authentication protocol. Our approach follows the security definitions in the literature of SMC [Yao, 1982, 1986, Goldreich et al., 1987, Canetti, 2000, Ben-David et al., 2008, Katz and Lindell, 2007, Goldreich, 2004]. As a result, our proposed protocol can be easily proved to be secure under the semi-honest model of SMC by using the simulation argument [Goldreich, 2004] as follows.

Definition 2 *Let T_i be the input of party i, $\prod_i(\pi)$ be i's execution image of the protocol π and s be the result computed from π. π is secure if $\prod_i(\pi)$ can be simulated from $\langle T_i, s\rangle$ and distribution of the simulated image is computationally indistinguishable from $\prod_i(\pi)$.*

In the above definition, an execution image generally includes the input, the output and the messages communicated during an execution of the protocol. By showing that the execution image of a protocol does not leak any information regarding the private inputs of participating parties, the protocol can be proved to be secure [Goldreich, 2004]. In our case, the execution image for the data server mainly includes the two encrypted feature vectors. As mentioned earlier, since the data server does not have the private/decryption key and the encryption scheme is semantically secure, the image is computationally indistinguishable

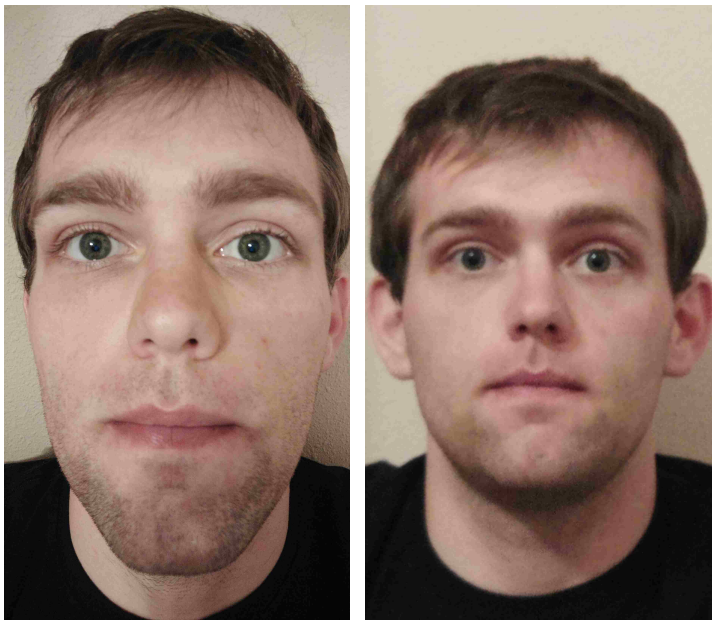
from a random sequence. Therefore, no information regarding the user's private data is leaked to the data server before executing the garbled circuit. Similar argument applies to the key server because the information that it received is randomized. In addition, all the intermediate results are either encrypted or randomized, and the garbled circuit approach is secure under the semi-honest model. As a result, based on the composition theorem [Goldreich, 2004], the overall protocol (Algorithm 4) is secure under the semi-honest model, i.e., any information regarding any users' private data is never leaked during the execution of our proposed protocol.

8.2. ATTACK ANALYSIS

8.2.1. Impersonation Attack. This is the most concerning attack in face authentication whereby the attacker tries to use the user's photo to gain access to the user's web accounts [Duc and Minh, 2009]. To perform such attacks on our UFace system, the attacker needs to have the targeted user's $UserID$ and the user's close up facial image. The user and the web service are the only 2 parties that know the user's $UserID$. UFace assumes that the web service provider is responsible for its own security since if the attacker compromises the web service provider, the attacker directly gains all control of the user's account without the need for authentication. Even so, it is worth noting that the attacker still would not have the users' feature vectors to masquerade as the user in other web services. Thus, the attacker would need to compromise each web service individually to obtain all of the user's accounts. UFace also assumes that the user's phone has an up-to-date operating system and anti-virus software so that the phone cannot be directly attacked and compromised. Moreover, to further prevent the attacker from collecting authentication information on the user's phone, all the user-end authentication can be performed in the secure zone on the phone (this varies from manufacturer to manufacturer). Also, the Android app deletes the photo used to generate the $UFacePass$ after each authentication attempt. This means that every attempt at authentication is done on a new image.

The next discussion will cover the scenarios when the attacker breaks into either the UFace data server or the key server since these two servers are located in a cloud and may be less protected. The data server possesses only an $IndexID$ corresponding to the user's $UFacePass$ and the key server has nothing with respect to the user. By compromising both of these authentication servers, the attacker still would not be able to guess the user's $UserID$ from the $IndexID$ since the $IndexID$ is basically a memory address in the data server. However, the attacker would be able to decrypt each user's $UFacePass$ to obtain the original feature vector. Since the feature vector is just a histogram of different labels in each of the sections, the attacker would not be able to reconstruct the original image of the user's face easily. So if both UFace servers are compromised, the attacker would still not be able to gain access to a specific user's account since it would not have the $UserID$ nor the image used to generate the $UFacePass$.

Considering a more advanced attack whereby the attacker obtains the $UserID$ as well, our UFace system still prevents attackers to access a user's web service account. This security is based on how the Android application itself works. The application stores the $UserID$ and the $IndexID$ for each web service the user registers with and doesn't allow the user to re-input once the web service has been registered. Thus, to be able to impersonate a specific user, the attacker would need to rewrite the Android application to allow for specific values to be sent for $UserID$ and $IndexID$. All the while, the application needs to maintain the same signature to fool the UFace servers that the Android application has not been tampered with. Finally, if the attacker manages all of this, then he/she would still need to convert the feature vector into the image it came from which is a computationally hard problem. If the attacker would try and crop an image of the user's face stored online or take a photo of the user's face from a distance, then the focal point will be at a different location which will change the feature vector generated by LBP. The cause for this change is that when the camera's focal point is closer to the face, then the picture that is taken will make the face appear more narrow. Meanwhile, if the picture is taken farther away, the user's



(a) Close-up Photo (b) Zoomed-in Photo

Figure 8.1. Difference between close-up photos vs. zoomed-in photos

face will appear more round. These different facial appearances will cause the respective feature vectors to be different as well (as long as the grid size is large enough to account for spatial differences). An example of this difference can be seen in Figure 8.1a and Figure 8.1b which compares the images of 2 different photos of the same face taken with different focal points making sure the entire face is filling most of the image. Our experiments with 20 users have proved that the LBP algorithm is capable of distinguishing these 2 types of images (see Section 9 for a detailed walk-through of the comparisons. If the user does not use a close-up facial photo for registering, then it will be easier for an attacker to crop out the user's face from previously stored photos on social media or take a new picture by zooming in. The further back the original photo was taken, the more easily the attacker can create a similar image. This is the reason why each photo should be taken extremely close up so that the user's face is narrow and fills the entire image.

8.2.2. Man-in-the-Middle Attack. This is an attack where the attacker acts in-between the user and the authentication servers trying to fool each party into thinking they are directly communicating with each other. Many existing techniques, such as Public Key Infrastructures, can be adopted to help users verify the genuine authentication servers when establishing the secure communication channel. For example, the user encrypts the session key using the server's public key. Then, only the genuine server would be able to decrypt it and obtain the session key which will be used for the subsequent communication between the user and the server. Thus, the man-in-the-middle attack can be prevented.

8.2.3. Malleability Attack. An encryption algorithm is malleable if it is possible for an adversary to transform a ciphertext into another ciphertext. Our protocol is robust against this because it adopts the secure communication channel established using AES encryption which has been proven to not be malleable.

9. EXPERIMENTAL STUDY

In this section, the experimental settings are first introduced followed by the evaluation metrics. Followed this is the experimental setup in which the speed and accuracy of the mobile application was analyzed.

The UFace system consists of an mobile application for the client side and the security protocols at the server side. The UFace app was tested on an Android Nexus 5 device which uses the Snapdragon 800 processor (4 cores at 2.3 GHz) and contains 2 GB RAM. The web services, data server, and key server were all run on the same virtual machine that used an Intel Xeon processor (6 cores at 3.5 GHz) and had 8.5GB of RAM. All photos taken of users were taken using their own mobile phones and the quality of the camera varied from each phone. To keep tests comparable, each photo was then tested using the Android One Plus Three as the client.

The performance of the UFace system is evaluated using two metrics: (i) *accuracy*, and (ii) *response time*. Accuracy refers to the rate at which each user is able to correctly authenticate into the system using only a close-up picture of their own face and not a zoomed-in/cropped picture of the user's face nor using a different person's face. Accuracy can thus be broken into the number of false negatives (true user's close-up image not working) and false positives (incorrect user/zoomed-in image of the user's face able to be used for authentication). Response time refers to getting the user the authentication results back in a timely manner; for this system, the time should be within a few seconds so that UFace can be used in real time. The analysis of [Card et al., 1991, Miller, 1968] show that users maintain attention when response time is less than 10 seconds, which is something UFace can guarantee. Table 9.1 summarizes the parameters tested with the highlighted metrics being the values determined to be best. In what follows, the experiments are reported on what was done along with the results.

Table 9.1. Experimental settings

Parameters	**Values**
Number of users	20
Number of comparisons per test	800
Number of bits needed per bin	11, 13, **15**, 17, 19, 21
Encryption key size	**1,024 bits**
LBP threshold	**92.78%**
Grid size	1, **4**, 16
Photo Size	128×128, **256**×**256**, 512×512, 1,024×1,024

9.1. EXPERIMENT SETUP

The purpose of this experiment was to analyze the accuracy and speed of the UFace system. The experiment was done using 20 different users all between the age of 22 and 25. Each user provided 3 images for testing using their own mobile devices: 2 different close-up images and one zoomed-in image. These images were stored on the same Nexus 5 before being tested on that device. All tests were done on the same mobile Android device even though the photos were taken from different many different devices. One randomly chosen close-up photo was used to generate the $UFacePass$ to store on the UFace data server. The other close-up photo and the zoomed-in photo were converted to a $UFacePass$ and tested against every user's stored $UFacePass$. This meant that with 20 users, each close up photo was tested against $20 \times 2 = 40$ different images; thus, the total number of tests for each experiment came out to be $20 \times 40 = 800$.

These tests were analyzed from varying 2 variables: grid size and photo size. The grid size varied from having 1 section, to 4 sections, and finally 16 sections. The photo size varied from having a width/height of 128 pixels each up to 1,024 in powers of 2. Thus, there were 12 different experiments run each with 800 tests each. Once again, the number of bits used to represent each bin varied based off the grid and picture size by the following

equation: $log_2\frac{N}{k} + 1$ where N is the total number of pixels and k is the grid size. For each test, the bit size was also changed to be the most optimal value (ie. it means that the entire bin value can be saved and no bits are missing).

During each of the 800 tests, the accuracy was analyzed by calculating the similarity of the 2 feature vectors (the close-up image stored on the UFace data server and the test image). The garbled circuit outputs the histogram intersection of the 2 feature vectors which is the number of pixels per section of the images that matched. This value was then divided by the number of pixels in the image to obtain the percentage of similarity between the 2 feature vectors. Finally, for each of the 20 users, the 40 accuracy values were ordered from greatest to least. By ordering each user's tests, it was easy to find the number of true positives/false positives/false negatives by just comparing which image had the highest comparison value and checking the rank of the image that should correctly match.

The time was analyzed from each different component of the mobile application and from the time taken to run the garbled circuit. The time taken for message transmission is omitted since this would vary based on the user's phone service or home internet service provider - factors outside of the system's control. The time for the mobile application was divided into 3 sections: (1) feature vector generation, (2) data manipulation, and (3) encryption. The time for the authentication was simply the time taken for the garbled circuit to run. These 4 components determined the amount of time taken to complete authentication.

9.2. ACCURACY ANALYSIS

The first step was aimed to identify the picture size, grid size and threshold value that would achieve the highest accuracy rate. This meant finding the values which would match the most users correctly while not matching an incorrect user. It also meant that zoomed-in photos should never be able to be matched with to preserve security of using online photos.

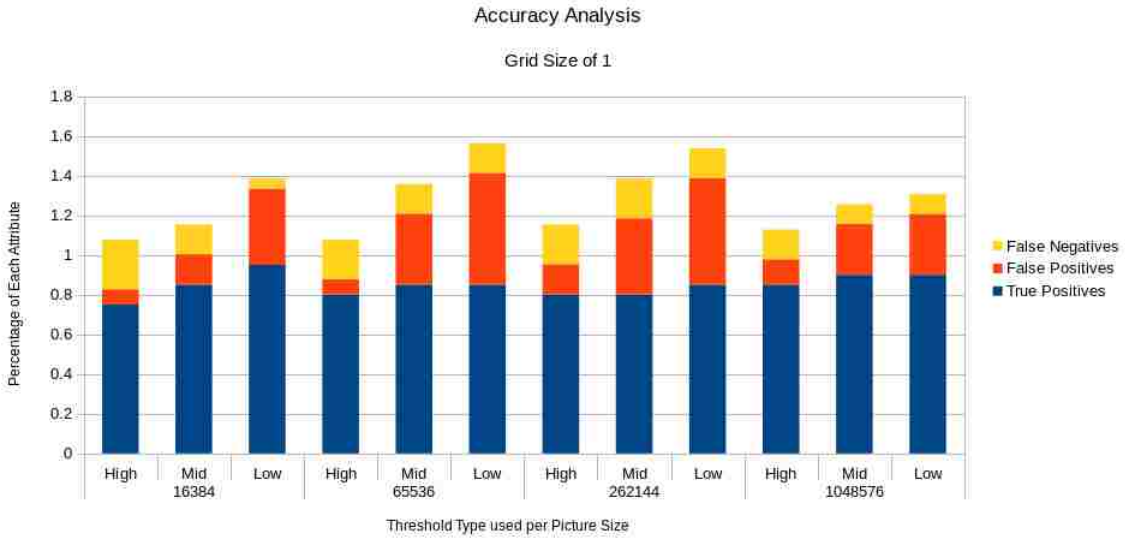


Figure 9.1. Graph showing accuracy for a grid size of 1 while picture size varies

For this analysis, the percentage of correctly matched users was compared against matched incorrect users and unmatched correct users. This test was also done against 3 different threshold values which were calculated from the best matching accuracy average - Avg_{Best} - and its associated standard deviation - $STDEV$. The threshold values are considered: High, Mid, and Low. High is associated with the threshold value of the $Avg_{Best} - STDEV$, Mid is calculated similarly with $Avg_{Best} - 1.5 \times STDEV$, and finally Low is derived from $Avg_{Best} - 2 \times STDEV$.

9.2.1. Varied Picture Size. The first analysis allowed the picture size to vary while the grid size was kept constant. The result can be seen in Figures 9.1, 9.2, and 9.3. Each graph shows the correctly matched users (true positives), incorrectly matched users (false positives), and incorrectly unmatched users (false negatives). Finally, to track general trends across different threshold values, the correlation tables and the associated average values from the graphs can be seen in Tables 9.2, 9.3, and 9.4.

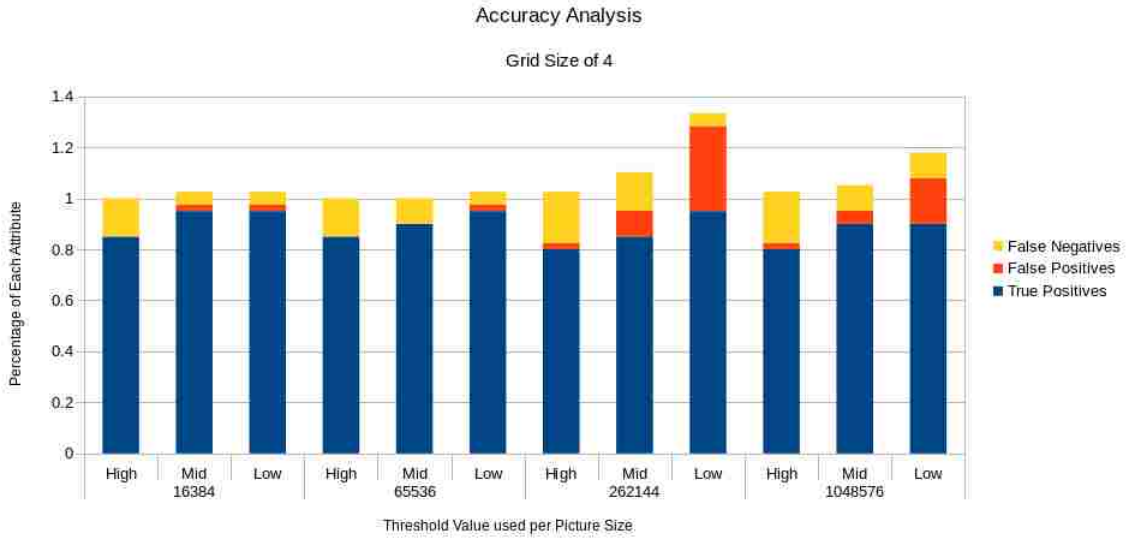


Figure 9.2. Graph showing accuracy for a grid size of 4 while picture size varies

Table 9.2. Table showing true positive percentages with correlation for varying picture size

	Grid Size of 1			Grid Size of 4			Grid Size of 16		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low
16,384	0.7500	0.8500	0.9500	0.8500	0.9500	0.9500	0.8500	0.9000	1.0000
65,536	0.8000	0.8500	0.8500	0.8500	0.9000	0.9500	0.8000	0.9000	0.9500
262,144	0.8000	0.8000	0.8500	0.8000	0.8500	0.9500	0.8500	0.9000	1.000
1,048,576	0.8500	0.9000	0.9000	0.8000	0.9000	0.9000	0.8000	0.9000	0.9500
Correlation	0.9487	0.3162	-0.4045	-0.8944	-0.6325	-0.7746	-0.4472	N/A	-0.4472

The results show that across the same grid size and same threshold percentages, there are a few general trends in the data. The first general trend is that as the picture size increased, the percentage of correct users decreased. Secondly, as the picture size increased, the percentage of false positives increased. Finally, as the picture size increased, the percentage of false negatives decreased. The reason for these results comes from how the histogram intersection works: summing the minimum value of each bin in the 2 $UFacePass$ and then dividing by the number of pixels in the image. If there are more

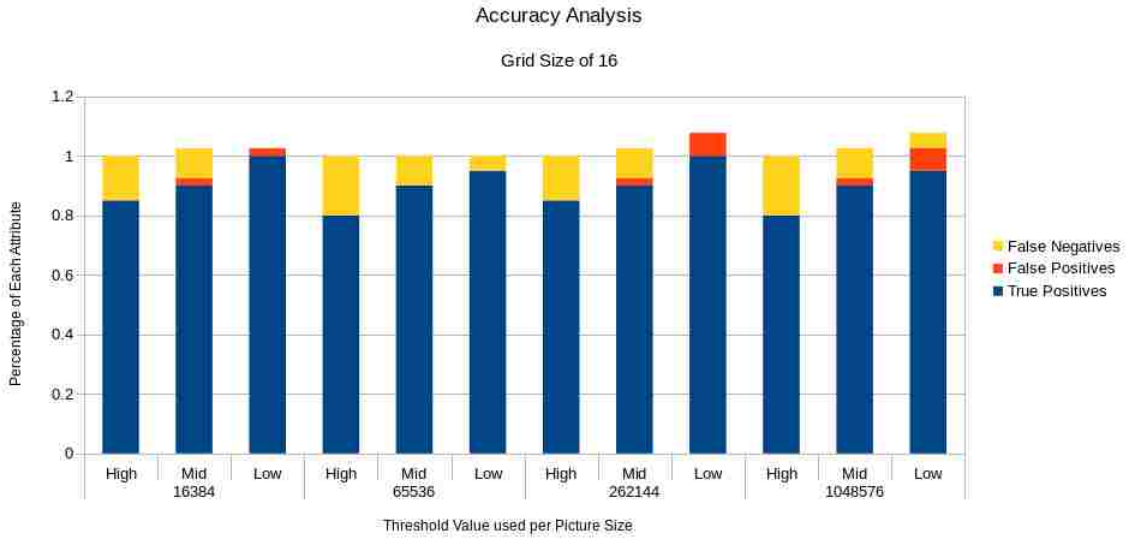


Figure 9.3. Graph showing accuracy for a grid size of 16 while picture size varies

Table 9.3. Table showing false positive percentages with correlation for varying picture size

	Grid Size of 1			Grid Size of 4			Grid Size of 16		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low
16,384	0.0769	0.1538	0.3846	0.0000	0.0256	0.0256	0.0000	0.0256	0.0256
65,536	0.0769	0.3590	0.5641	0.0000	0.0000	0.0256	0.0000	0.0000	0.0000
262,144	0.1538	0.3846	0.5385	0.0256	0.1026	0.3333	0.0000	0.0256	0.0769
1,048,576	0.1282	0.2564	0.3077	0.0256	0.0513	0.1795	0.0000	0.0256	0.0769
Correlation	0.7746	0.4080	-0.2692	0.8944	0.5292	0.6742	N/A	0.2582	0.7746

pixels in the image, then each bin should theoretically be larger as well since there are just more labels per section of the images. Thus, the minimum values are larger with a larger picture size and so the sum of the minimums is larger, too.

So the reason the percentage of true positives decreased while the percentage of false positives increased is because with a larger histogram, more labels appear. Thus, if a test $UFacePass$ is supposed to be a correct match, then there is a higher chance that the labels appearing in each $UFacePass$ will be different as the picture size increases. On the other side of this, if a test $UFacePass$ is supposed to fail, then there is a higher chance

Table 9.4. Table showing false negative percentages with correlation for varying picture size

	Grid Size of 1			Grid Size of 4			Grid Size of 16		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low
16,384	0.2500	0.1500	0.0500	0.1500	0.0500	0.0500	0.1500	0.1000	0.0000
65,536	0.2000	0.1500	0.1500	0.1500	0.1000	0.0500	0.2000	0.1000	0.0500
262,144	0.2000	0.2000	0.1500	0.2000	0.1500	0.0500	0.1500	0.1000	0.0000
1,048,576	0.1500	0.1000	0.1000	0.2000	0.1000	0.1000	0.2000	0.1000	0.0500
Correlation	-0.9487	-0.3162	0.4045	0.8944	0.6325	0.7746	0.4472	N/A	0.4472

that more of the similar labels will appear in each $UFacePass$ as the picture size increases. If the 2 images are a perfect match, however, then there will be no change in accuracy if the picture size increases. Another reason these change could appear is because for every different picture size, the threshold value is adjusted to still be related to the Avg_{Best} and the $STDEV$. If the threshold remained constant across every test, then the percentage of correct users would increase. Therefore, the reason the percentage of false positives generally increases is because the sum of minimums is larger for a comparison between 2 $UFacePass$es. Thus, for 2 images of different users or if one is zoomed-in of the user, they would have a higher probability of matching if there are more pixels in the image. Finally, the reason the percentage of false negatives increases is because it's the complement of true positives. As the number of true positives decrease, then the number of false negatives increases by the same amount.

9.2.2. Varied Grid Size. The second analysis of the data varied the grid size while the picture size was kept constant. The result can be seen in the 4 graphs: Figure 9.4, 9.5, 9.6, and 9.7. Again, the tables for each graph can be seen in Tables 9.5, 9.6, and 9.7 which also show the correlation values for each comparison.

Generally, as the grid size increases, the percentage of true positives increases. Also, as the grid size increases, the percentage of false positives and false negatives decreases. The reason for these trends is that as the grid size increases, spatial information plays more

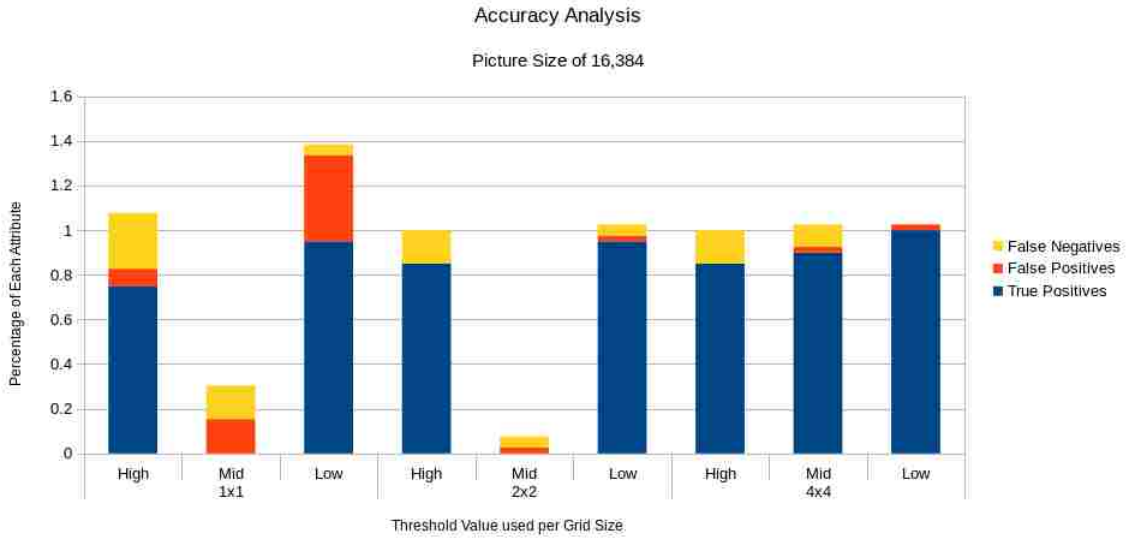


Figure 9.4. Graph showing accuracy for a picture size of 16,384 while grid size varies

Table 9.5. Table showing true positive percentages with correlation for varying grid size

	Picture Size of 16,384			Picture Size of 65,536			Picture Size of 262,144			Picture Size of 1,048,576		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low	High	Mid	Low
1	0.7500	0.8500	0.9500	0.8000	0.8500	0.8500	0.8000	0.8000	0.8500	0.8500	0.9000	0.9000
4	0.8500	0.9500	0.9500	0.8500	0.9000	0.9500	0.8000	0.8500	0.9500	0.8000	0.9000	0.9000
16	0.8500	0.9000	1.0000	0.8000	0.9000	0.9500	0.8500	0.9000	1.0000	0.8000	0.9000	0.9500
Correlation	0.8660	0.5000	0.8660	0.0000	0.8660	0.8660	0.8660	1.0000	0.9820	-0.8660	N/A	0.8660

of a role in comparing 2 images. In smaller grid sizes, a specific pixel near one side of the section may have a label that matches to the same label as a pixel on the opposite side of the same section in the other photo. However, in larger grids, the number of pixels per section decreases and the pixel with the same label might be in a completely different section. This would mean those 2 labels would not be accounted for in same portion of the histogram and thus, the difference between the 2 images would be greater than if the pixels were located in the same section.

Since the spatial information for 2 close-up images of the same user should be similar, the percentage of true positives should increase. This also means that false positives should decrease because if the 2 feature vectors don't come from close-up photos of the same user,

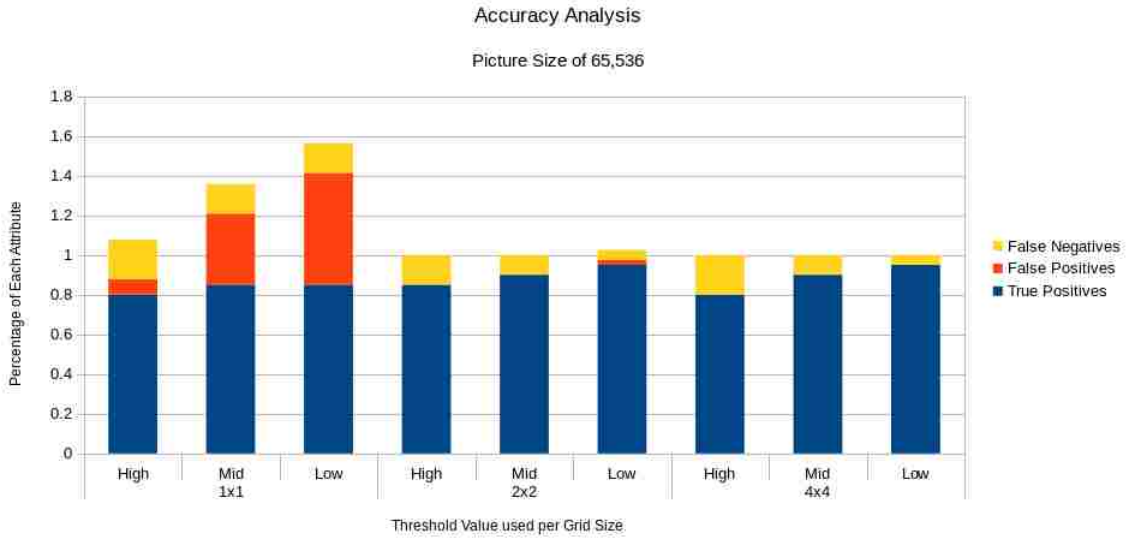


Figure 9.5. Graph showing accuracy for a picture size of 65,536 while grid size varies

Table 9.6. Table showing false positive percentages with correlation for varying grid size

	Picture Size of 16,384			Picture Size of 65,536			Picture Size of 262,144			Picture Size of 1,048,576		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low	High	Mid	Low
1	0.0769	0.1538	0.3846	0.0769	0.3590	0.5641	0.1538	0.3846	0.5385	0.1282	0.2564	0.3077
4	0.0000	0.0256	0.0256	0.0000	0.0000	0.0256	0.0256	0.1026	0.3333	0.0256	0.0513	0.1795
16	0.0000	0.0256	0.0256	0.0000	0.0000	0.0000	0.0000	0.0256	0.0769	0.0000	0.0256	0.0769
Correlation	-0.8660	-0.8660	-0.8660	-0.8660	-0.8660	-0.8854	-0.9333	-0.9497	-0.9979	-0.9449	-0.9122	-0.9979

then the spatial information should be different resulting in different histograms. The reason the false negatives decrease is once again inversely related to the number of true positives and their increase as grid size increases.

Another reason for the increase in true positives from increasing the grid size is because the number of pixels in each section decreases. A general trend in the previous trend was that as picture size increases, the percentage of true positives decreases. The inverse of this can be true: as picture size decreases the percentage of true positives increases. By increasing the grid size, the picture size is essentially decreasing for each section of the $UFacePass$. Thus, since the number of pixels decreased per each section, the percentage of matching to only correct $UFacePass$ increased, too.

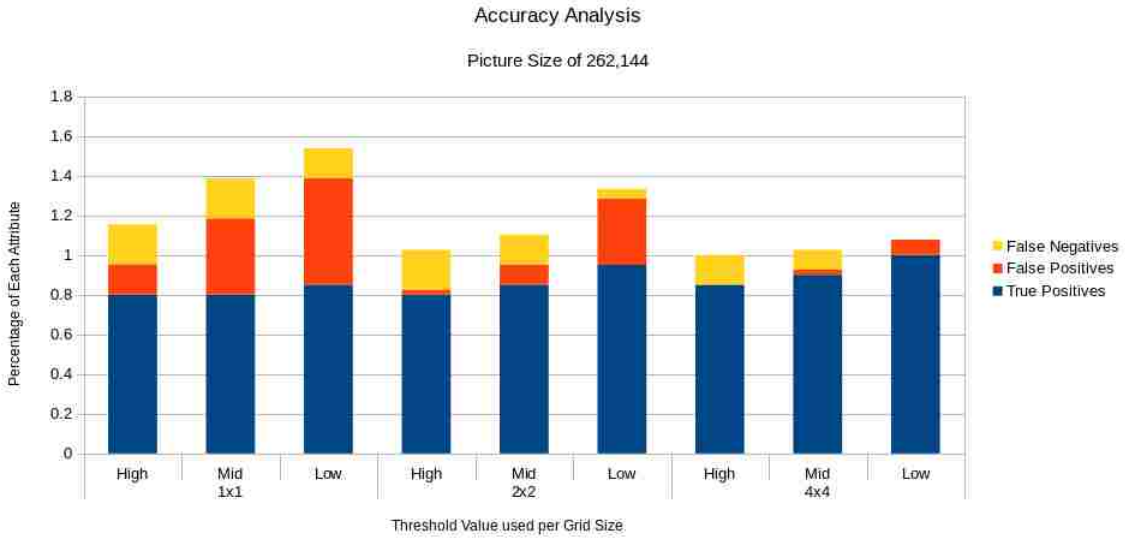


Figure 9.6. Graph showing accuracy for a picture size of 262,144 while grid size varies

Table 9.7. Table showing false negative percentages with correlation for varying grid size

	Picture Size of 16,384			Picture Size of 65,536			Picture Size of 262,144			Picture Size of 1,048,576		
Picture Size	High	Mid	Low	High	Mid	Low	High	Mid	Low	High	Mid	Low
1	0.2500	0.1500	0.0500	0.2000	0.1500	0.1500	0.2000	0.2000	0.1500	0.1500	0.1000	0.1000
4	0.1500	0.0500	0.0500	0.1500	0.1000	0.0500	0.2000	0.1500	0.0500	0.2000	0.1000	0.1000
16	0.1500	0.1000	0.0000	0.2000	0.1000	0.0500	0.1500	0.1000	0.0000	0.2000	0.1000	0.0500
Correlation	-0.8660	-0.5000	-0.8660	0.0000	-0.8660	-0.8660	-0.8660	-1.0000	-0.9820	0.8660	N/A	-0.8660

9.2.3. Best Accuracy Comparison. For determining the best accuracy, false positives should be 0 while true positives should be high (or false negatives being low). False positives mean that an attacker would be able to access a user's account. If this value is anything besides a 0, then the system would allow at least 1 attacker access to a user's account in just the experiments already completed; this should never happen. False negatives, on the other hand, should be low, but don't necessarily have to be 0. False negatives mean that an attempt at logging in failed for a correct user. So, if this occurs then the user would just need to attempt to log in once again. However, the higher the false negatives the more likely a correct user will not be able to access his/her account unless the test image used is nearly

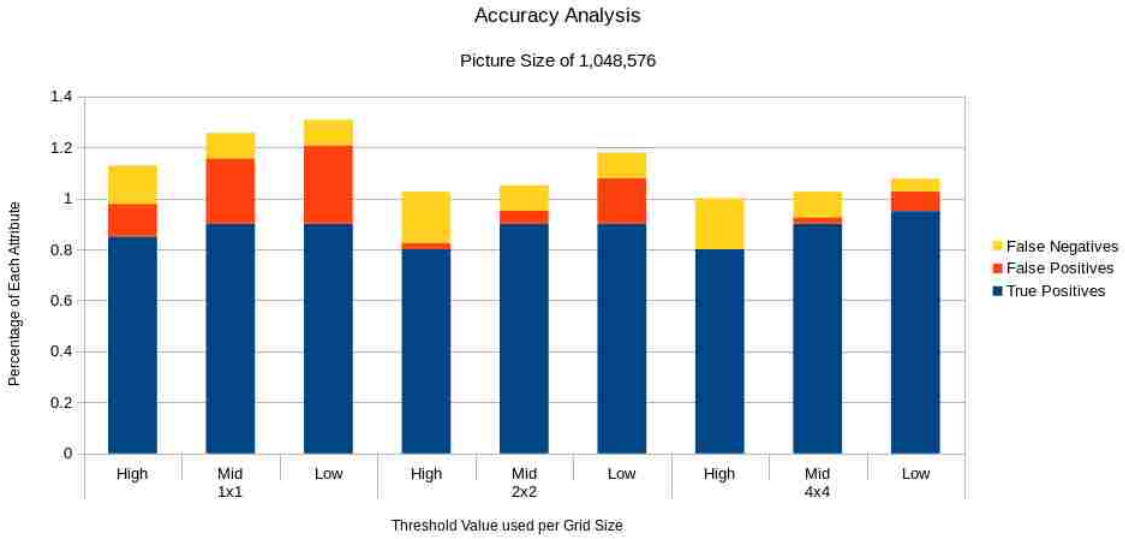


Figure 9.7. Graph showing accuracy for a picture size of 1,048,576 while grid size varies

identical to the image used to generate the first $UFacePass$. The final piece of information in deciding on an ideal picture and grid size is the difference between the threshold value used and the next best matching incorrect photo.

There needs to be a large enough difference between the threshold value decided upon and the next best matching incorrect image. If the difference is negative, then there will be at least 1 false positive, which as already stated is not allowed in this system. If the difference is positive and close to 0, then it would be more likely that a false positive can occur with more users when the system is implemented in the real world. Thus, having a large positive difference between the threshold and the max next best matching incorrect image is ideal.

Therefore, for determining the best accuracy there were 3 nearly ideal cases each with 0 false positives: grid size of 2×2 with a picture size of 256×256 with a Mid threshold value or a grid size of 4×4 with a picture size of 256×256 with a Mid or Low threshold. For the case with a grid size of 2×2 and picture size 256×256, the threshold value was 92.78%. This setting provided 90% correct match rate by matching 18 users with his/her

second close-up image correctly. This meant that there were 2 out of 18 users that were not able to authenticate properly (or 10%) on the first try. These users would need to attempt authentication again to successfully authenticate. It should be noted that difference between the threshold and the next best matching accuracy value was 0.0071.

The next 2 cases came from using a grid size of 4×4 and a picture size of 256×256. The 2 thresholds were 88.65% and 87.61% for the Mid and $High$ case respectively. The Mid case had provided 90% correct match rate with 10% false negatives. The $High$ case provided better accuracy with 95% correct match rate and only 5% false negatives. However, the main difference between these is the difference between the threshold value and the next best accuracy value: 0.0111 (Mid) and 0.0006 ($High$). For this case, even though the $High$ threshold value allowed 1 more person to authenticate on the first try, it's more likely that an attacker will be able to gain access compared to using the Mid threshold. Therefore, between these 2 threshold values, the Mid with a threshold of 88.65% was chosen to be a better.

There were a few more cases were 0 false positives occurred; however, they resulted in 3 or more false negatives. For this system, 90% was chosen to be the minimum rate at which correct users could authenticate on the first try. If the value was any lower, then the system could be seen to be unreliable.

9.3. TIME ANALYSIS

Now that accuracy has been analyzed and 2 cases stand out, the time needs to be analyzed to see if authentication can be done in real-time. This section analyses the time taken to authenticate a user based off both the time taken to run the garbled circuit and the time taken for the mobile device to create the encrypted feature vector.

9.3.1. Mobile Time Analysis. The amount of time UFace is operating on the mobile device can be seen in Figure 9.8.

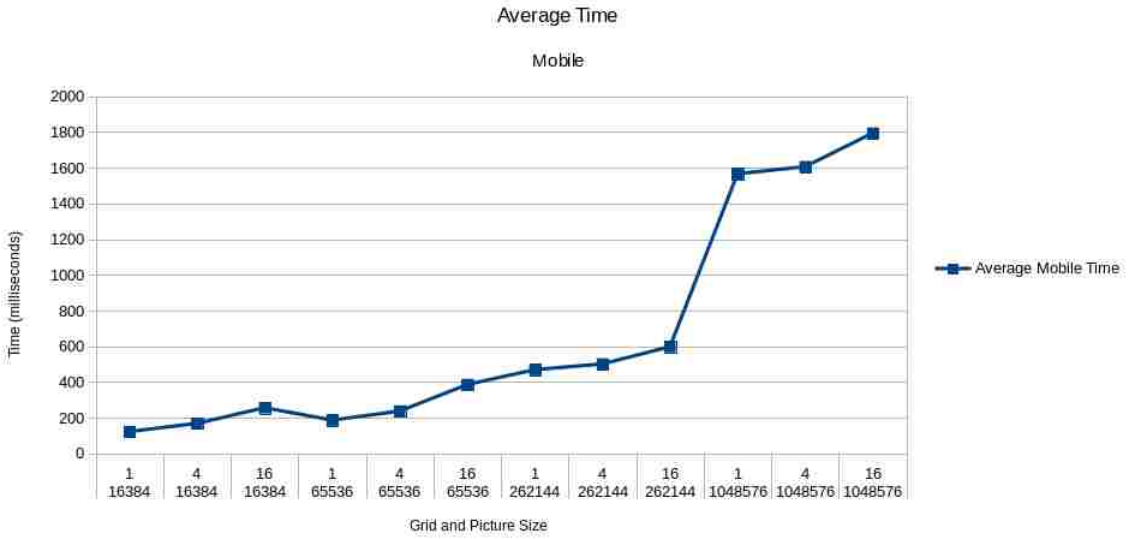


Figure 9.8. Graph showing the average total mobile time for each experiment

The general trend is that by increasing either the grid size or the picture size, the amount of time will increase for the mobile device. The reason for this increase is because as the picture size increases, the LBP protocol needs to be executed on more pixels which takes more time - this can be seen in Figure 9.9.

Also, if the grid size increases, there will be more bins that need to be concatenated together and then encrypted before being sent to the UFace data server. The way grid size affects the time for encryption can be seen in Figure 9.10.

Finally, Figure 9.11 shows the entire amount of time needed for the UFace application to run on a mobile device. In Figure 9.11, there are 2 extra segments in the column - average manipulation time and average resize time. The average manipulation time generally increases as grid size increases; however, the amount of time taken is tiny in comparison to the time needed for LBP and encryption. Resize time is the time needed to change an image into one that can used in the UFace system; this time does increase as the picture size increases, but it increases at a slower rate compared to LBP execution. See the Table 9.8 to see how much time is taken for every aspect of the mobile application.

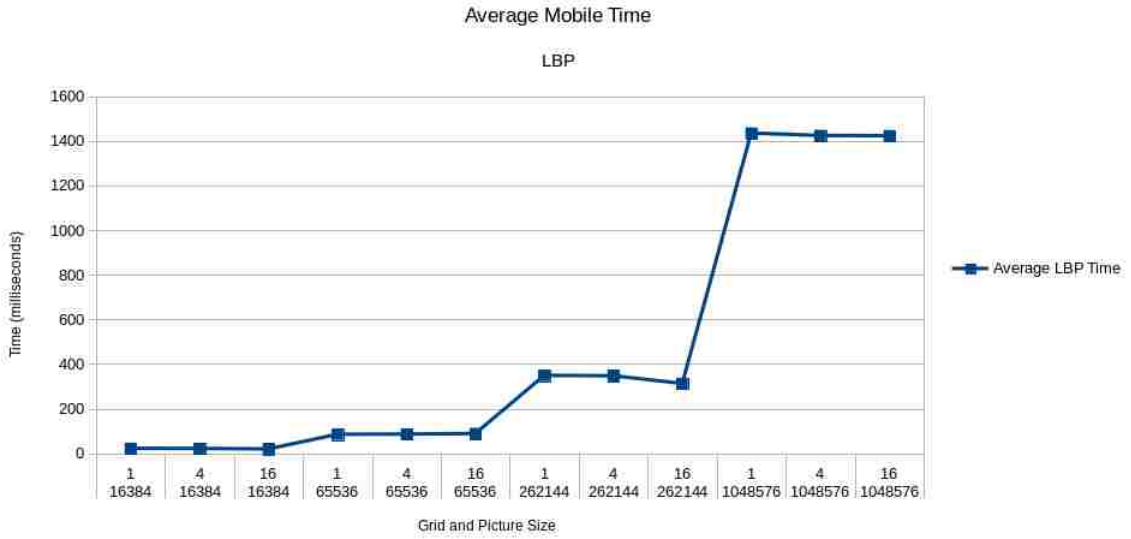


Figure 9.9. Graph showing the average total LBP execution time for each experiment

The time taken for LBP is completely dependent on the number of pixels, and has a linear relationship. The time taken to encrypt the feature vector is also a linear relationship with the number of needed encryptions; however, the number of encryptions increases at a non-linear rate. The number of decryptions is dependent on the number of bits needed to completely encompass all values possible in any bin of the histogram and the number of bins in the histogram. Since the number of bits needed follows the equation $log_2 \frac{N}{k} + 1$ and the number of bins comes from the equation $k \times 59$, the total number of encryptions then becomes:

$$Count_{encryptions} = \lceil \frac{k \times 59}{\lfloor \frac{1024}{log_2 \frac{N}{k} + 1} \rfloor} \rceil \tag{9.1}$$

For the experimental setup, the amount of time to execute LBP on any specific pixel was roughly 0.00132 ms while the time for a single encryption was 16.391 ms. As a simple example, for the case with a picture size of 65,536 and a grid size of 4, the time taken for LBP

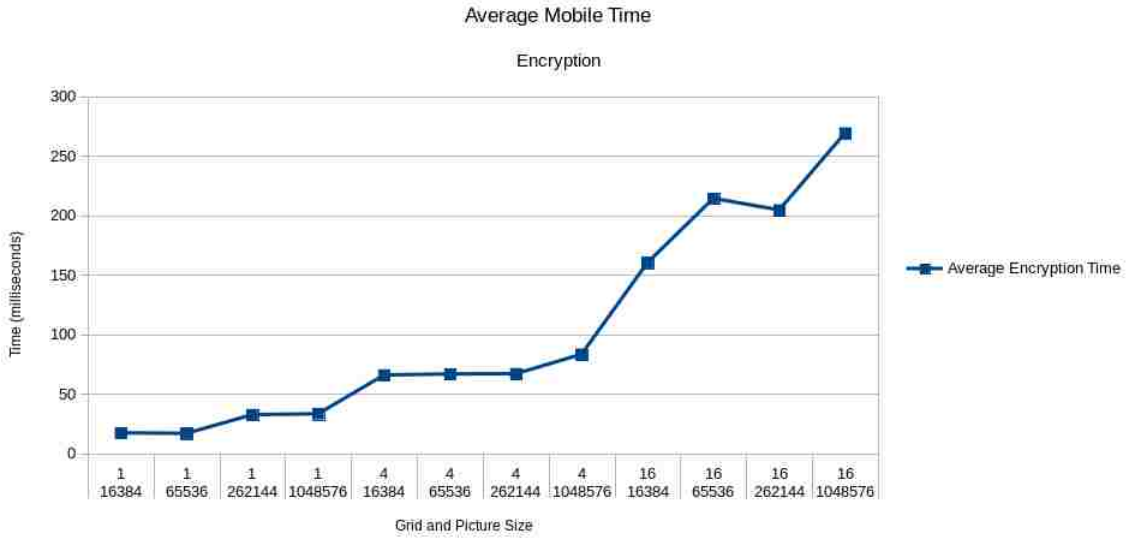


Figure 9.10. Graph showing the average total encryption time for each experiment

would be roughly $0.00132 \times 65,536 = 86.5075$ ms which is similar to the 86.3175 ms as seen in Table 9.8. The number of encryptions would be equal to 4 and $4 \times 16.391 = 65.564$ ms which is similar to the 66.7225 ms as seen in Table 9.8.

9.3.2. Garbled Circuit Time Analysis. The other portion of time spend executing UFace was the time spent executing the authentication protocol through garbled circuits. Figure 9.12 displays the data and Table 9.9 shows the values.

Increasing the grid size increased the amount of time needed to execute the protocol. The reason for this is that as the grid size increased, the number of bins needed increased as well and thus, the garbled circuit needed to be larger to run the protocol over all of the bins. And since the garbled circuit was larger, it took more time to execute. Increasing the picture size played no significant role in changing the amount of time needed for the GC to run for the smaller 2 grid sizes. However, when the grid size was large, then as the picture size increased the number of encryptions needed increased more rapidly. The number of encryptions doesn't necessarily play a role in executing the authentication protocol; however, it does play a role in the time for the proper data to be used as input to

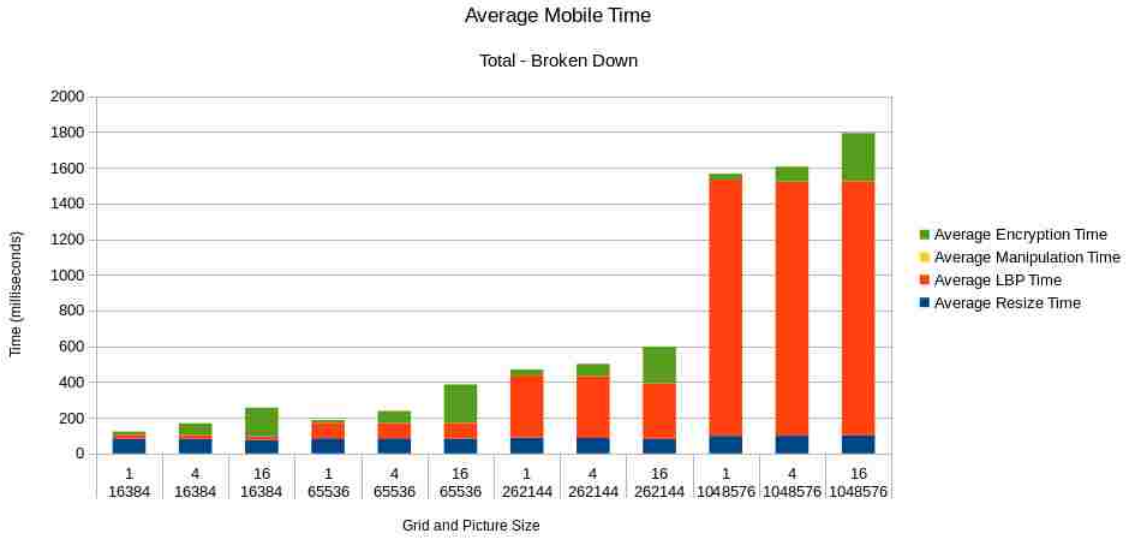


Figure 9.11. Graph showing the average total time expanding each component of mobile time for each experiment

the garbled circuit. The reasoning behind this is that, these encryptions relate to a specific number of bins across 1 grid size (ie. if the grid size remains constant, then the number of bins remains constant too - and thus the number of inputs into the garbled circuit). As the picture size increases, these encrypted values needed to be first decrypted and then the random value provided to it from the UFace data server needs to be removed (see Section 7.2). This happens within the garbled circuit, so the time does get affected by varying the picture size when the grid size is large enough.

9.3.3. Total Time Analysis. The total time taken can be seen in Figure 9.13. In every possible case, the amount of time required to authenticate a user would require less than 7 seconds; in most cases, the time would be less than 5 seconds. However, this time needs to be comparable to the time a typical login would need. As stated in 9, if the time was less than 10 seconds, then users would remain attentive. No experimental setup took

Table 9.8. Table showing the average total time expanding each component of mobile time for each experiment

Picture Size	Grid Size	Resize Time	LBP Time	Manipulation Time	Encryption Time	Total Time
16,384	1	82.5625	22.4875	0.03625	17.24625	122.3325
16,384	4	80.44375	22.11	0.135	65.79875	168.4875
16,384	16	75.37625	19.87	0.37375	160.20625	255.82625
65,536	1	84.0325	85.49125	0.04125	16.81375	186.37875
65,536	4	84.05	86.3175	0.1325	66.7225	237.2225
65,536	16	82.14375	88.9575	0.4525	214.2275	385.78125
262,144	1	87.28875	349.29	0.065	32.54625	469.19
262,144	4	86.17	347.79	0.15125	66.87125	500.9825
262,144	16	79.8775	312.965	0.4	204.50875	597.75125
1,048,576	1	98.08625	1,434.78375	0.05875	33.265	1,566.19375
1,048,576	4	98.575	1,424.19	0.1425	83.2975	1,606.205
1,048,576	16	101.075	1,423.05625	0.455	269.16	1,793.74625

that long and thus, UFace can work in a real setting; however, having a system faster than 7 seconds would be ideal. Thus, the accuracy needs to be analyzed with the time to find the ideal parameters.

9.4. IDEAL PARAMETERS ANALYSIS

Back in Section 9.2, there were 2 cases that provided 90% accuracy for correct users with 0 false positives. The first used a grid size of 2×2 and a picture size of 256×256. The average total amount of time for a single authentication attempt was 2830.48 milliseconds or roughly 2.8 seconds. For the case with a grid size of 4×4 and a picture size of 256×256 the average total time was 4,414.56 milliseconds or about 4.4 seconds. The time with the larger grid size was about 1.57 times larger than the case with the smaller grid size and only provided a difference of nearly 0.004 for the threshold against the next best matching value. With these 2 different cases, it seems that the smaller grid size was more ideal since the average total time was under 3 seconds and they both provided the same accuracy values. The ideal values are highlighted in Table 9.1. Table 9.10 shows the accuracy values and times for each experiment and the highlighted values are the ideal values used.

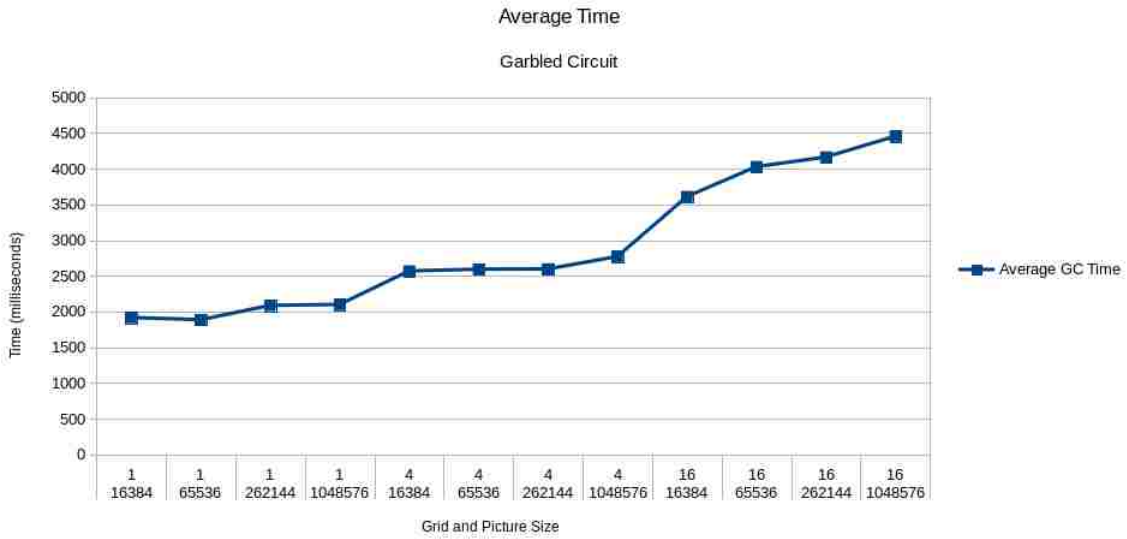


Figure 9.12. Graph showing the average Garbled Circuit time for each experiment

Table 9.9. Table showing the average Garbled Circuit time for each experiment

Picture Size	Grid Size	Average Garbled Circuit Time
16,384	1	1,915.84125
65,536	1	1,885.1175
262,144	1	2,085.71375
1,048,576	1	2,098.87625
16,384	4	2,566.95625
65,536	4	2,593.2575
262,144	4	2,596.1575
1,048,576	4	2,770.56375
16,384	16	3,609.5475
65,536	16	4,028.7825
262,144	16	4,163.6275
1,048,576	16	4,454.64625

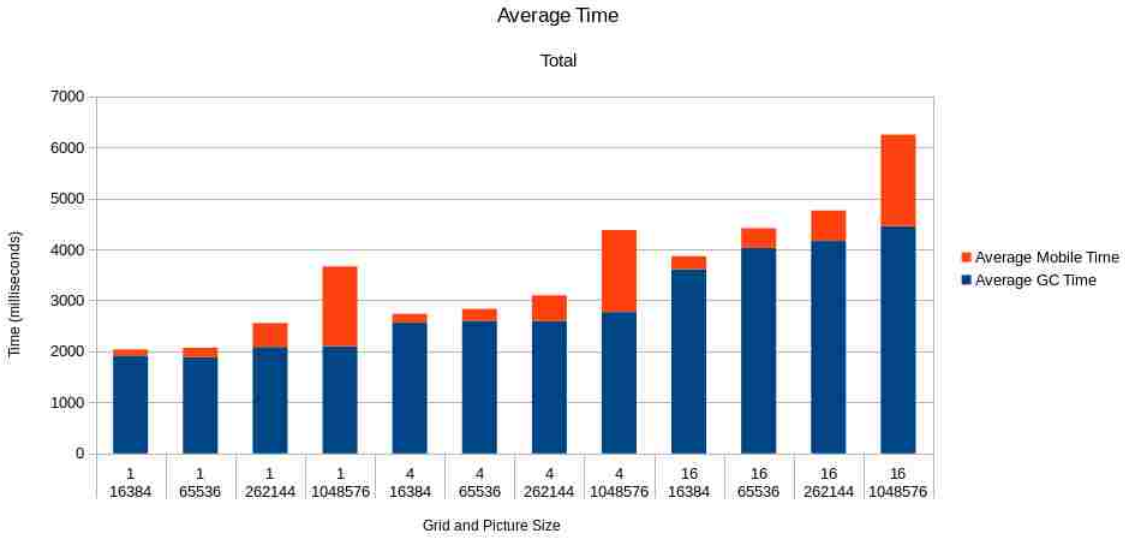


Figure 9.13. Graph showing the average total time slices for each experiment

Table 9.10. Table showing average accuracy and time analysis for all experiments

Grid	Picture	Level	Threshold	T. Pos.	F. Neg.	F. Pos.	Difference	GC Time	Mobile Time	Total Time
1	16,384	High	0.9461	15	5	3	-0.0106	1,915.8413	122.3325	2,038.1738
1	16,384	Mid	0.9401	17	3	6	-0.0166	1,915.8413	122.3325	2,038.1738
1	16,384	Low	0.9341	19	1	15	-0.0226	1,915.8413	122.3325	2,038.1738
1	65,536	High	0.9549	16	4	3	-0.0130	1,885.1175	186.3788	2,071.4963
1	65,536	Mid	0.9481	17	3	14	-0.0198	1,885.1175	186.3788	2,071.4963
1	65,536	Low	0.9412	17	3	22	-0.0267	1,885.1175	186.3788	2,071.4963
1	262,144	High	0.9514	16	4	6	-0.0170	2,085.7138	469.1900	2,554.9038
1	262,144	Mid	0.9424	16	4	15	-0.0260	2,085.7138	469.1900	2,554.9038
1	262,144	Low	0.9334	17	3	21	-0.0350	2,085.7138	469.1900	2,554.9038
1	1,048,576	High	0.9541	17	3	5	-0.0143	2,098.8763	1,566.1938	3,665.0700
1	1,048,576	Mid	0.9450	18	2	10	-0.0234	2,098.8763	1,566.1938	3,665.0700
1	1,048,576	Low	0.9359	18	2	12	-0.0325	2,098.8763	1,566.1938	3,665.0700
4	16,384	High	0.9120	17	3	0	0.0066	2,566.9563	168.4875	2,735.4438
4	16,384	Mid	0.9041	19	1	1	-0.0013	2,566.9563	168.4875	2,735.4438
4	16,384	Low	0.8962	19	1	1	-0.0092	2,566.9563	168.4875	2,735.4438
4	65,536	High	0.9354	17	3	0	0.0148	2,593.2575	237.2225	2,830.4800
4	**65,536**	**Mid**	**0.9278**	**18**	**2**	**0**	**0.0071**	**2,593.2575**	**237.2225**	**2,830.4800**
4	65,536	Low	0.9202	19	1	1	-0.0005	2,593.2575	237.2225	2,830.4800
4	262,144	High	0.9325	16	4	1	-0.0039	2,596.1575	500.9825	3,097.1400
4	262,144	Mid	0.9208	17	3	4	-0.0156	2,596.1575	500.9825	3,097.1400
4	262,144	Low	0.9091	19	1	13	-0.0273	2,596.1575	500.9825	3,097.1400
4	1,048,576	High	0.9398	16	4	1	0.0000	2,770.5638	1,606.2050	4,376.7688
4	1,048,576	Mid	0.9283	18	2	2	-0.0116	2,770.5638	1,606.2050	4,376.7688
4	1,048,576	Low	0.9168	18	2	7	-0.0231	2,770.5638	1,606.2050	4,376.7688
16	16,384	High	0.8496	17	3	0	0.0092	3,609.5475	255.8263	3,865.3738
16	16,384	Mid	0.8370	18	2	1	-0.0033	3,609.5475	255.8263	3,865.3738
16	16,384	Low	0.8244	20	0	1	-0.0159	3,609.5475	255.8263	3,865.3738
16	65,536	High	0.8970	16	4	0	0.0215	4,028.7825	385.7813	4,414.5638
16	65,536	Mid	0.8865	18	2	0	0.0111	4,028.7825	385.7813	4,414.5638
16	65,536	Low	0.8761	19	1	0	0.0006	4,028.7825	385.7813	4,414.5638
16	262,144	High	0.9096	17	3	0	0.0068	4,163.6275	597.7513	4,761.3788
16	262,144	Mid	0.8967	18	2	1	-0.0061	4,163.6275	597.7513	4,761.3788
16	262,144	Low	0.8838	20	0	3	-0.0190	4,163.6275	597.7513	4,761.3788
16	1,048,576	High	0.9197	16	4	0	0.0079	4,454.6463	1,793.7463	6,248.3925
16	1,048,576	Mid	0.9063	18	2	1	-0.0055	4,454.6463	1,793.7463	6,248.3925
16	1,048,576	Low	0.8929	19	1	3	-0.0190	4,454.6463	1,793.7463	6,248.3925

10. CONCLUSION

In this paper, a novel privacy-preserving face authentication system was presented, called UFace, for authenticating web services on a mobile device. UFace is unique in that it allows a web service to offload its authentication computation while still securely authenticating any of its users. UFace is also able to authenticate users without the need to store facial image in its database. Instead, UFace maintains a database of encrypted facial image feature vectors and uses a garbled circuit technique to compare these feature vectors without leaking data. UFace is able to correctly authenticate 90% of the users that were tested while preventing 0 masquerade attacks. All the while, UFace can complete authentication in less than 3 seconds which can be used in many systems today. Since UFace is correct, fast, and prevents multiple types of attacks, it has the capability to be the central authentication protocol for any future web service.

REFERENCES

Anurag Tagat. Online fraud: too many accounts, too few passwords, 2012. URL http://www.techradar.com/us/news/internet/online-fraud-too-many-accounts-too-few-passwords-1089283.

Xiaoyang Tan and Bill Triggs. Enhanced local texture feature sets for face recognition under difficult lighting conditions. *Image Processing, IEEE Transactions on*, 19(6):1635–1650, 2010.

Yan Li, Yingjiu Li, Qiang Yan, Hancong Kong, and Robert H. Deng. Seeing your face is not enough: An inertial sensor-based liveness detection for face authentication. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1558–1569, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813612. URL http://doi.acm.org.libproxy.mst.edu/10.1145/2810103.2813612.

Yi Xu, True Price, Jan-Michael Frahm, and Fabian Monrose. Virtual u: Defeating face liveness detection by building virtual models from your public photos. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 497–512, 2016.

Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies*, pages 235–253. Springer, 2009.

J. Bringer, H. Chabanne, and A. Patey. Privacy-preserving biometric identification using secure multiparty computation: An overview and recent trends. *Signal Processing Magazine, IEEE*, 30(2):42–52, March 2013. ISSN 1053-5888. doi: 10.1109/MSP.2012.2230218.

Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *Information, Security and Cryptology–ICISC 2009*, pages 229–244. Springer, 2010.

Margarita Osadchy, Benny Pinkas, Ayman Jarrous, and Boaz Moskovich. Scifi-a system for secure face identification. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 239–254. IEEE, 2010.

David Evans, Yan Huang, Jonathan Katz, and Lior Malka. Efficient privacy-preserving biometric identification. In *Proceedings of the 17th conference Network and Distributed System Security Symposium, NDSS*, 2011.

Marina Blanton and Paolo Gasti. Secure and efficient protocols for iris and fingerprint identification. In *Computer Security–ESORICS 2011*, pages 190–209. Springer, 2011.

Jaroslav Sedenka, Sathya Govindarajan, Paolo Gasti, and Kiran S Balagani. Secure outsourced biometric authentication with performance evaluation on smartphones. *Information Forensics and Security, IEEE Transactions on*, 10(2):384–396, 2015.

Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology–EUROCRYPT 2008*, pages 146–162. Springer, 2008.

Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *Theory of Cryptography*, pages 457–473. Springer, 2009.

Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

Doyel Pal, Praveenkumar Khethavath, Johnson P Thomas, and Tingting Chen. Secure and privacy preserving biometric authentication using watermarking technique. In *Security in Computing and Communications*, pages 146–156. Springer, 2015.

Hu Chun, Yousef Elmehdwi, Feng Li, Prabir Bhattacharya, and Wei Jiang. Outsourceable two-party privacy-preserving biometric authentication. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 401–412. ACM, 2014.

Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

Peter N Belhumeur, João P Hespanha, and David J Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(7):711–720, 1997.

Timo Ahonen, Abdenour Hadid, and Matti Pietikäinen. Face recognition with local binary patterns. In *Computer vision-eccv 2004*, pages 469–481. Springer, 2004.

Timo Ojala, Matti Pietikäinen, and Topi Mäenpää. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):971–987, July 2002. ISSN 0162-8828. doi: 10.1109/TPAMI.2002.1017623. URL http://dx.doi.org/10.1109/TPAMI.2002.1017623.

Andrew C. Yao. Protocols for secure computation. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 160–164. IEEE, 1982.

Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.

Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game - a completeness theorem for protocols with honest majority. In *19th ACM Symposium on the Theory of Computing*, pages 218–229, New York, New York, United States, 1987. URL http://doi.acm.org/10.1145/28395.28420.

R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp - a system for secure multi-party computation. In *Proceedings of the ACM Computer and Communications Security Conference (ACM CCS)*, October 2008.

J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.

Oded Goldreich. *The Foundations of Cryptography*, volume 2, chapter Encryption Schemes. Cambridge University Press, 2004. URL http://www.wisdom.weizmann.ac.il/~oded/PSBookFrag/enc.ps.

Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *The 20th USENIX Security Symposium*, August 2011.

P. Paillier. Public key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - Eurocrypt '99 Proceedings, LNCS 1592*, pages 223–238, Prague, Czech Republic, May 2-6 1999. Springer-Verlag.

S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

NM Duc and BQ Minh. Your face is not your password. In *Black Hat Conference*, volume 1, 2009.

Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM. ISBN 0-89791-383-3. doi: 10.1145/108844.108874. URL http://doi.acm.org/10.1145/108844.108874.

Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL http://doi.acm.org/10.1145/1476589.1476628.

VITA

Nicholas Steven Hilbert was born in 29 Palms California in January 15^{th} 1992. He received his Bachelor Degree in Computer Engineering from Missouri University of Science and Technology in December 2014. He joined the Master's program in Computer Science at the same university in the beginning of 2015. His research adviser was Dr. Dan Lin. Nicholas Hilbert's primary interest were computer security in biometrics and secure multi-party computations. He received the Master of Science in Computer Science from Missouri S&T in July 2017.