

Online Bank Web Application

HANEEN ALOSAIMI,¹ RANIA ALHARBI,¹ SARA ALKAHTANI,¹ AND JUMANAH ALMUTAIRI¹

¹*CY321 Group No. 1*

INTRODUCTION

The significant growth of online banking web-based applications has rapidly transformed the way individuals and organizations interact, communicate, and manage their financial daily activities, with a few clicks the user can transfer money, check their bank account, pay bills without having to visit the bank branch. This convenience has made financial services easier, faster, flexible, and more Accessible to all users. However, despite all these advantages that online banking has achieved, cyber threats have increased significantly over time. Attackers use phishing, malware, and identity theft techniques to steal customer financial information or disrupt banking operations. Unsecured networks, weak authentication, and outdated software can lead to significant financial losses and data breach. Consequently, the security and privacy of online transactions have emerged as a significant challenge for both organizations and individuals. This research analyzes the current state of online banking, emphasizing its key advantages and related security challenges, while proposing practical solutions in order to improve the overall financial security environment.

RELATED WORK

The research on online and mobile banking systems has, in recent years, concentrated on the elaboration of secure, scalable, and user-friendly platforms, balancing functionality with protection from emerging cyber threats. A study on modern web-based banking platforms shows that the use of React for the front-end interfaces, Laravel (PHP) for the back-end services, and MySQL for data management can enhance performance and maintainability significantly. This work emphasizes that biometric verification, One-Time Passwords (OTP), and AI-driven fraud detection are some of the effective ways through which authentication can be strengthened and sensitive data protected. Furthermore, it has been shown that usability and interface design are not secondary issues but central to the attainment of customer satisfaction and trust in financial technology environments [1].

The second strand of literature provides a review on the changing face of ebanking, consequent to the emergence of new technologies like Artificial Intelligence, Blockchain, and FinTech innovation. These works highlight how digital tools facilitate access and efficiency yet introduce new security challenges and regulatory issues. Generally, known threats are phishing, cyber fraud, and data breach, which require an enhanced biometric verification mechanism, threat detection based on AI, and Blockchain-based validation. This body of work recognizes the complexity of maintaining compliance with international standards such as GDPR and PSD2, noting that unified global cybersecurity practice is needed in order for digital banking to be sustainable [2].

Technical analyses of online banking architecture have presented system designs in layers, where multiple authentications and encryption methods are proposed to maintain transaction integrity. A comparative study of such design describes a system with a client interface, a bank server, and a security layer using TLS 1.0, RSA, and SHA1 encryption to secure the data of clients. It may also include biometric verification, OTP generators, and token-based authentication in order to handle potential threats such as identity hacking, phishing, and malware attacks. These findings reinforce the need for strong communication security and rigorous monitoring within financial infrastructures [3].

Complementary research has proposed comprehensive security frameworks based on MFA and robust encryption protocols. These are mostly designed based on three logical layers, namely user interface, application, and database, which work in cohesion to maintain confidentiality and prevent unauthorized access. Besides the technological solutions of AES and RSA encryption, some studies mention user education and continuous software maintenance as non-technical yet crucial parts of security resilience [4].

Convenience and vulnerability are highlighted as interrelated notions in the context of mobile and hybrid banking systems. Most contemporary platforms use 2FA security features, which rely on traditional credentials together with OTP verification to ensure secure authentication. More recent designs have included biometric

systems like fingerprint or facial recognition to increase their reliability. Researchers identified common security risks of phishing, malware attacks, and social engineering and called for enhanced countermeasures against them through integrated technical and user awareness approaches [5].

More recent analyses further extend these insights to propose multilayered defense architectures: client, application, and database layers, underpinned by firewalls, IDS, and biometric modules. These will offer protection against complex attacks such as SQL injection, MITM, session hijacking, and pharming. This prevention is effective through the adoption of AES/RSA encryption, regular audits of the system, and training programs for employees and customers. All these studies together demonstrate that a truly secure banking ecosystem requires technological advancement along with continuous human-centered vigilance [6].

All the reviewed literature clearly points out that multi-factor authentication, cryptographic encryption, and AI-assisted fraud detection combined with regulatory compliance and user education represent the future for online banking security. While technological sophistication in such cybersecurity matters will continue to change, the crucial need for ensuring public confidence in digital finance requires a holistic approach that unites technical, organizational, and behavioral dimensions of cybersecurity.

PAPER 1

Summary of Paper 1 : this paper discusses the innovation and challenges and developing modern online banking systems. It presents an advance architecture that integrates react for the frontend , Laravel (PHP) for the backend, and MySQL for database management to enhance performance,scalability and security.

The study emphasizes key security features such as one-time passwords (OTP), biometric identification and AI-based fraud detection to strengthen authentication and protect user data it's also highlights the importance of user experience and interface design showing that usability and accessibility directly affect customer satisfaction and trust.

additionally, the paper examines how legal compliance , disaster recovery and data backup are integrated into the banking system. It identifies ongoing challenges, including scalability issues, system integration with legacy infrastructure, and regulatory complexities.

overall, the paper shows that combining traditional banking principles with modern technologies can create more secure, efficient and user-friendly online banking

systems, setting a foundation for future financial technology innovations .

PAPER 2

Summary of paper 2 :

this paper provides a comprehensive review of e-banking services, focusing on their evolution current trends and challenges. It explain how to technologies like artificial intelligence Blockchain and Fintech have transformed digital banking, approving efficiency and accessibility. The study highlights, major security issues, such as cyber fraud, fishing and hacking and emphasizes the need of biometric authentication , AI based fraud detection and Blockchain verification to ensure safe transactions. It also discusses regulatory challenges, including global compliance and data protection laws like GDPR and PSD2. key findings show that mobile banking and digital payments are driving growth, but gaps remain in digital literacy, financial inclusion and unified global cyber security standards. The paper concludes that the future of e- banking depends on continuous innovation, stronger cyber security frameworks, and global cooperation to create a secure and user-friendly digital banking environment.

PAPER 3

This paper presents a comparative study of security mechanisms used in online banking systems. The authors explain that modern banking services allow customers to conduct financial transactions over the internet through computers or mobile devices. These systems typically operate in two main phases: a registration phase and a login phase. During registration, customers provide their personal and financial information to the bank. The login process involves a two-level authentication structure. The first level uses a user ID and password, while the second level enhances security through advanced verification methods such as One-Time Passwords (OTP), grid cards, QR codes, biometric verification, security questions, and electronic tokens. The system architecture of online banking consists of a client interface, a bank server that processes all transactions, a security layer that uses TLS 1.0 encryption and RSA/MD5/SHA1 algorithms for data protection, and a secure database to store customer information and transaction records. In addition, authentication subsystems such as OTP generators and biometric scanners are implemented to improve verification accuracy. The services provided by online banking include secure login and logout, fund transfers, bill payments, and notification systems such as SMS banking. Communication between the client and server is protected through secure TLS connections to maintain confidentiality and

integrity. The paper also identifies several threats that online banking systems commonly face. These include phishing attacks, internet scams, malware infections, identity theft, keystroke logging, and pharming attacks that redirect users to fake websites. To mitigate these risks, the authors suggest several security solutions, such as the use of OTPs, QR codes, grid authority cards, biometric authentication, and token-based password generation. They also emphasize the importance of updated antivirus systems, firewalls, and regular monitoring of bank networks.

PAPER 4

This paper focuses on improving the security of online banking systems by analyzing existing vulnerabilities and proposing a comprehensive framework for protecting customer data and financial transactions. The authors emphasize that as online banking becomes more widely adopted, the number and sophistication of cyberattacks also increase, requiring stronger authentication and encryption mechanisms to ensure user trust. The proposed system architecture consists of three major layers: the user interface layer, the application layer, and the database layer. The user interface allows customers to perform typical operations such as balance inquiries, fund transfers, and bill payments through secure web portals. The application layer is responsible for processing requests, verifying user credentials, and enforcing security policies. The database layer stores sensitive customer information, account data, and transaction logs, all of which are protected using encryption techniques such as AES and RSA. The framework integrates multi-factor authentication (MFA) to strengthen the login process. In addition to the traditional username and password, users must verify their identity using one or more of the following methods: a one-time password (OTP), biometric verification, or a smart token device. This multi-layered approach significantly reduces the risk of unauthorized access. The paper also highlights several common security threats that affect online banking platforms. These include phishing attacks, man-in-the-middle (MITM) attacks, malware injections, and denial-of-service (DoS) attempts that disrupt normal banking operations. The authors analyze these threats and explain how encryption, secure socket layers (SSL/TLS), and regular software patching can mitigate them effectively. In addition to authentication and encryption, the paper discusses user-side precautions, such as educating customers about recognizing phishing emails, avoiding untrusted networks, and ensuring that antivirus software is regularly updated.

PAPER 5

The paper explains how a mobile banking system works, describing how users register their personal and banking information through secure online forms. Once the registration is completed, the system verifies their details to ensure authenticity and security. During the login phase, users are required to use two-factor authentication (2FA) methods that combine a username and password with a One-Time Password (OTP) sent to their mobile device. This approach provides an additional layer of protection against unauthorized access and ensures that only legitimate users can perform transactions.

In terms of system architecture and components, the research discusses the technical structure of the online and mobile banking environment, which consists of servers, communication networks, and applications, whether they are web-based or mobile-based. It also highlights the security architecture, which integrates various encryption techniques and authentication mechanisms such as fingerprint recognition, OTP verification, and other biometric authentication methods to enhance the protection of user data. These technologies ensure that sensitive financial information remains confidential and protected during transmission between the client and the banking server.

Regarding the services provided, the paper notes that modern electronic and mobile banking systems offer a wide range of convenient features. These include performing financial transactions, checking account balances, transferring funds, paying bills, and receiving security alerts or notifications. The paper emphasizes the importance of user convenience, allowing customers to access their accounts and services seamlessly through both mobile applications and web platforms, ensuring flexibility and accessibility anytime and anywhere.

Finally, the paper discusses several security risks that mobile and electronic banking systems commonly face. These include fraud, phishing attacks, malware infections, and a general lack of security awareness among users. To address these threats, the authors recommend the implementation of multi-factor authentication (MFA), user education and awareness training, and the adoption of biometric verification methods such as fingerprint or facial recognition. These solutions are designed to strengthen overall system security, prevent unauthorized access, and build user trust in online banking platforms.

PAPER 6

The paper explains how the online banking system works, where customers can use various banking services such as checking their balance, transferring money, and

paying bills, all through the Internet. First, the customer registers by providing their personal and banking information, which is then verified by the bank. After that comes the second step, which is the login phase. It uses two authentication methods, including the user-name and password along with OTP messages to increase security. Each transaction or request sent from the customer's device to the bank server goes through a secure communication channel using the HTTPS/TLS protocol to ensure data confidentiality and prevent tampering. In terms of architecture and components, according to the paper, the system consists of three layers: client layer, application layer, and database layer. The client layer represents the user interface, either as a mobile application or a web platform. The application layer includes the logic of the system and is responsible for processing requests, performing computations, and applying security policies. Finally, the database layer stores customers' data, accounts, and transaction records, which are usually encrypted using AES. The paper also mentions that "The architecture consists of firewalls, intrusion detection systems, authentication servers, token generators, and biometric modules to ensure multi-level protection in online financial transactions." The paper states that modern online banking systems provide a variety of essential and extended services such as local and international money transfers, online bill payments, account management (financial reports, balance, and previous transactions), and sending notifications to users via SMS or email. It also includes card management (freeze, reissue, and spending tracking) and online technical support through chatbots or email. Regarding security threats, there are several risks that online banking may face, and the paper focuses on the most important ones: Phishing — stealing user data through fake websites that appear to be legitimate; Malware — stealing passwords or intercepting data during transmission; SQL Injection — attacking the database directly; Man-in-the-Middle (MITM) — intercepting the connection between the client and the bank; and Session Hijacking — taking over an active user session. Proposed Solutions: Multi-Factor Authentication (MFA), Encryption (RSA and AES), Regular Security Audits, User Awareness Training, Firewalls and Intrusion Detection Systems.

DESIGN SPECIFICATION

A. USE CASE SCENARIO

Lama is a user who uses online banking Web application since it provides easier and user-friendly banking functions. she receives funds, checks her balance and transfer amounts of money.Lama is not aware of secu-

urity. She uses weak passwords and uses the same password across multiple platforms. All the data is stored and uploaded to a data database server and it is processed by an application server and the third-party service handles system notifications and Lama is provided with the services on a user interface, this is all connected via an application programming interface.

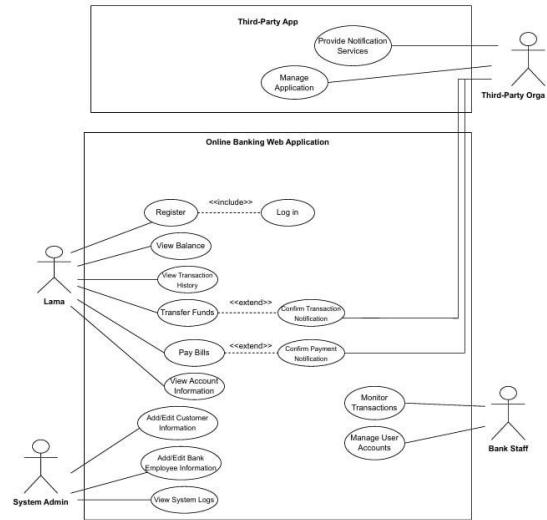


Figure 1. Use Case Diagram.

B. SYSTEM MODEL

Components :

User interface : this is what users see and interact with via the website, it allows customers to login check balances, transfer money and pay bills.

Database server : stores all the banking data, such as user accounts, balances, transaction, history, and login details .

Application servers : handles all the logic and operations of the system. It checks user credentials, processes, transaction, and communicate with the database.

Third party services : include the notification systems like SMS and email.

Application Programming interface (API) : Connect the application servers with the database servers, and sometimes with other services like payment gateways helps the system exchange data safely, and efficiently.

End user : person who makes use of the services provided by the online banking web application.

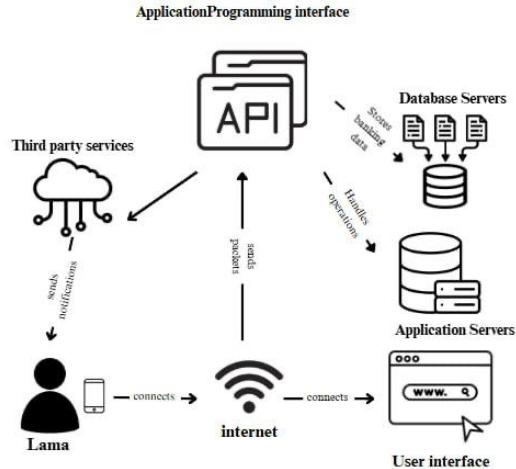


Figure 2. System Model Diagram.

C. THREAT ANALYSIS

Many threats that could arise using an online banking web application the end user's account information must be confidential and kept private.

Confidentiality threats : this threat occurs when sensitive information like account numbers, passwords, or transaction Data is exposed to unauthorized users. It can be done by data leaks eavesdropping, or weak encryption, and the impact could lead to user data and bank information to be stolen or misused.

Integrity threats : The threat occurs when data is altered or tampered with during storage or transmission. It can be done by unauthorized modification of transaction details or balances, which can lead to false financial records or fraudulent transactions.

Authentication threats : The threat happens when attackers bypass or fake user identities to gain access. It can be done by phishing stolen passwords, or weak login systems. The impact is an authorized user can control the account or make false transactions .

Availability threats : this threat happens when the banking system becomes unavailable to users or when web application is down, it can be done by denial of service attacks or system crashes. This prevents customers from accessing their accounts or perform transactions .

Privacy threat : privacy is threatened when personal data is collected used or shared without the user consent. This can happen by insider misuse or poor data protection policies , it can lead to loss of customer trust and legal consequences.

D. DESIGN REQUIREMENTS

1. Functional Requirements:

This section defines the functional requirements that the online banking system must provide to ensure a secure , efficient and user-friendly experience for customers.

1. **Balance Inquiry:** The system shall allow customers to check their account balance easily and in the shortest possible time, while ensuring high accuracy of the displayed information.
 2. **Money Transfer:** The system shall allow customers to transfer money easily and securely to any other account, whether local or international, with minimal steps and complications.
 3. **Login:** The system shall provide each customer with a unique username and password to log in securely to the application.
 4. **Add Money:** The system shall allow customers to add money to their accounts at any time through multiple payment methods such as credit cards, debit cards, or direct bank deposits.
 5. **View Transaction History:** The system shall store all customer transactions securely and allow customers to view their complete transaction history. Each record shall include detailed information such as the transaction type, date, time, and amount.
2. Security Requirements:
1. **Two-Factor Authentication:** The application shall allow only authorized users to access the system through two-factor authentication methods such as passwords and OTP messages.
 2. **Audit Logging:** The system shall be capable of recording all activities and transactions to support analysis, identify vulnerabilities, and detect abnormal behavior.
 3. **End-to-End Encryption:** The system shall encrypt all data during transmission and storage to ensure confidentiality and prevent unauthorized access.
 4. **Session Security:** The system shall automatically terminate inactive user sessions to prevent unauthorized access and protect user data.
 5. **Intrusion Detection:** The system shall include mechanisms to detect and alert administrators about suspicious activities or potential security breaches in real time.

HIGH-LEVEL IDEAS

To satisfy the design requirements of the online banking application, several cryptographic and security mechanisms will be implemented.

1. Authentication and Access Control:

To ensure that only authorized users can access the system, Two-Factor Authentication (2FA) will be applied.

- When a user attempts to log in, they will first enter their username and password.
- The system will then send a One-Time Password (OTP) to the user's registered phone number or email address.
- Only after entering the correct OTP will the user be granted access.

This mechanism strengthens user authentication and protects accounts from unauthorized access.

2. Data Confidentiality and Integrity:

Sensitive information such as registration data, account balance, and transaction details will be protected using the AES-256 encryption algorithm. AES ensures that even if communication is intercepted, the data remains unreadable to unauthorized parties. Additionally, digital signatures or Message Authentication Codes (MACs) will be used to verify the integrity of transmitted data and ensure that no modifications occur during transmission.

3. Secure Communication Channel:

The communication between the client and the bank server will be protected using RSA encryption and digital certificates.

- The client encrypts the session key using the server's RSA public key before sending it.
- The server decrypts the session key using its private key, ensuring that only the authorized server can access it.
- All further communication will use symmetric AES encryption with this shared session key for better efficiency.

This approach ensures mutual authentication, data confidentiality, and protection against man-in-the-middle (MITM) attacks.

4. Audit Logging and Monitoring:

All system activities — such as logins, fund transfers, and failed access attempts — will be recorded in secure audit logs. These logs support threat detection, incident investigation, and compliance auditing, providing traceability and accountability for all operations.

5. Session and Intrusion Security:

- The system will implement automatic session timeouts for inactive users to prevent session hijacking.
- An Intrusion Detection System (IDS) will continuously monitor network traffic and system behavior to detect and alert administrators of potential attacks in real time.

Together, these design ideas ensure that the online banking system achieves confidentiality, integrity, authentication, and availability, thereby creating a secure and trustworthy environment for all customers.

PROTOCOL DESIGN

The proposed protocol for the online banking web application aims to provide complete end-to-end security between the client and the bank server. It ensures confidentiality, integrity, authentication, non-repudiation, and availability throughout all communication and transaction processes.

The protocol consists of four main phases:

1. System Initialization.
2. Authentication Phase.
3. Secure Session Establishment.
4. Transaction and Monitoring Phase.

1. SYSTEM INITIALIZATION

In this phase, the cryptographic environment is prepared, and trust is established between entities before any sensitive communication begins.

- Both the bank server and the client generate asymmetric key pairs (RSA), consisting of a public key (PK) and a private key (SK).
- The bank server's public key is signed by a trusted Certificate Authority (CA) and distributed to clients to ensure authenticity.
- During user registration, each client's public key is securely stored in the bank's database.

- Digital certificates are generated for both entities to enable mutual authentication in later stages.
- An audit logging system is activated to record all important actions such as login attempts, fund transfers, and account modifications.
- The Intrusion Detection System (IDS) is also initialized to monitor for any unauthorized access attempts or suspicious activities.

2. AUTHENTICATION PHASE

This phase verifies the identity of the user and the legitimacy of the bank server before allowing access to any account or financial operation.

- The client sends a login request to the bank server containing an encrypted username and password.
- The server generates a random nonce and sends it to the client, requesting a One-Time Password (OTP) as part of the Two-Factor Authentication (2FA) mechanism.
- The client enters the OTP received via email or SMS and sends it back to the server.
- The server verifies the OTP, then uses RSA encryption to confirm that the client possesses the correct private key (digital identity proof).
- The server sends its digital certificate, encrypted with the client's public key, to confirm its authenticity.
- After mutual verification, the server generates a temporary session key (K_s), encrypts it using the client's public key, and sends it to the client.
- The client decrypts K_s using its private key, thus establishing a secure and trusted channel for further communication.

3. SECURE SESSION ESTABLISHMENT

After successful authentication, a secure communication session is established between the client and the server.

- All data transmitted within the session is encrypted using AES-256, ensuring both confidentiality and performance efficiency.

- Each message is attached with a digital signature or a Message Authentication Code (MAC) to protect data integrity and prevent tampering.
- A unique Session ID is generated for each connection to prevent replay attacks or session hijacking.
- Session timeouts are enforced to automatically terminate inactive or suspicious sessions.
- The Intrusion Detection System (IDS) continuously monitors connection behavior and alerts administrators in case of abnormal activity or repeated login failures.

4. TRANSACTION AND MONITORING PHASE

In this phase, the client performs secure financial transactions such as balance inquiries, fund transfers, and bill payments.

- All sensitive information (account numbers, transfer amounts, timestamps, etc.) is encrypted using the AES-256 session key before transmission.
- Each transaction includes a digital signature generated by the client to ensure data authenticity and non-repudiation.
- The bank server validates the digital signature and decrypts the transaction details before execution.
- All operations are recorded in the audit log along with session IDs, IP addresses, and timestamps for traceability.
- The IDS continuously monitors system activity to detect unusual behavior—such as repeated failed logins or abnormal transfer patterns—and immediately alerts security administrators.
- Once the session ends, the session key (K_s) is revoked to prevent reuse or exploitation.

IMPLEMENTATION

This is a simplified client / server demonstration of our protocol (mutual auth, RSA-backed session key exchange, AES session, MACs, OTP check, audit logging).

```


1 /**
2  import java.time.LocalDateTime;
3  import java.time.format.DateTimeFormatter;
4
5  public class AuditLogger {
6      public static void log(String event) {
7          String t = LocalDateTime.now().format( formatter:DateTimeFormatter.ISO_LOCAL_DATE_TIME);
8          System.out.println("[AUDIT " + t + "] " + event);
9          // In production: append to secure tamper-evident log, send to SIEM.
10     }
11 }
12
13
14
15 import javax.crypto.SecretKey;
16 import java.io.*;
17 import java.net.Socket;
18 import java.security.KeyPair;
19 import java.security.PublicKey;
20 import java.util.Base64;
21
22 public class ClientMain {
23     private final String host = "localhost";
24     private final int port = 9999;
25     private KeyPair clientKP;
26     private String username = "lama";
27     private String password = "1234";
28     private SecretKey sessionKey;
29     private PublicKey serverPubKey = null;
30
31     public ClientMain() throws Exception {
32         clientKP = CryptoUtils.generateRSAKeyPair();
33     }
34
35     public void runDemo() throws Exception {
36         Socket s = new Socket(host, port);
37         BufferedReader in = new BufferedReader(new InputStreamReader( in:s.getInputStream()));
38         PrintWriter out = new PrintWriter( out:s.getOutputStream(), autoFlush:true);
39
40         // 1. KEY EXCHANGE: Fetch server public key first.
41         out.println( >>> "KEYEXCHANGE");
42         String keyResp = in.readLine();
43         System.out.println("Server KeyExchange: " + keyResp);
44         if (keyResp != null && keyResp.startsWith( prefix:"OK PUBKEY ")) {
45             String serverPubB64 = keyResp.substring( beginIndex:10);
46             serverPubKey = PemUtils.publicKeyFromBase64( b64:serverPubB64);
47             System.out.println( >>> "Successfully fetched and set Server Public Key.");
48             try {
49                 // Ensure all output is sent before continuing
50             }
51         }
52     }
53 }

```

```

serverPubKey = remUtils.publicKeyFromBase64(serverResponse);
System.out.println(">>> Successfully fetched and set Server Public Key.");
try {
    // Ensure all output is sent before continuing
    out.flush();
    // A brief pause to let the connection stabilize
    Thread.sleep(100);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}
} else {
    System.err.println("Failed to fetch Server Public Key. Demo will stop.");
    s.close();
    return;
}

// 2) REGISTER
String clientPub = remUtils.publicKeyToBase64(clientKP.getPublic());
out.println("REGISTER " + username + ":" + password + ":" + clientPub + ":" + "lama@example.com");
System.out.println("Server: " + in.readLine());

// 3) LOGIN: send credentials encrypted with the 'dynamically' fetched server pub.
byte[] creds = (username + ":" + password).getBytes();
String encCreds = CryptoUtils.rsaEncryptBase64(data:creds, pub:serverPubKey);
out.println("LOGIN " + encCreds);
String serverResp = in.readLine();
System.out.println("Server: " + serverResp);

if (serverResp != null && serverResp.startsWith(prefix:"OTP")) {
    String otp = serverResp.substring(beginIndex:4);
    // for demo we have OTP; in real life user gets OTP on phone
    out.println("OTFVERIFY " + username + ":" + otp);
    String okcert = in.readLine();
    System.out.println("Server: " + okcert);
    // NOTE: The server still sends its public key again here (OK CERT), but we already have it. We keep this logic for completeness
    if (okcert != null && okcert.startsWith(prefix:"OK CERT")) {
        // generate AES session key and send encrypted by serverPub
        SecretKey ak = CryptoUtils.generateAESKey();
        sessionKey = ak;
        String encSession = CryptoUtils.rsaEncryptBase64(data:ak.getEncoded(), pub:serverPubKey);
        out.println("SESSIONKEY " + encSession);
        System.out.println("Server: " + in.readLine());
    }
    // Now we can send transaction
}

System.out.println("Server: " + in.readLine());

// 3) LOGIN: send credentials encrypted with the 'dynamically' fetched server pub.
byte[] creds = (username + ":" + password).getBytes();
String encCreds = CryptoUtils.rsaEncryptBase64(data:creds, pub:serverPubKey);
out.println("LOGIN " + encCreds);
String serverResp = in.readLine();
System.out.println("Server: " + serverResp);

if (serverResp != null && serverResp.startsWith(prefix:"OTP")) {
    String otp = serverResp.substring(beginIndex:4);
    // for demo we have OTP; in real life user gets OTP on phone
    out.println("OTFVERIFY " + username + ":" + otp);
    String okcert = in.readLine();
    System.out.println("Server: " + okcert);
    // NOTE: The server still sends its public key again here (OK CERT), but we already have it. We keep this logic for completeness
    if (okcert != null && okcert.startsWith(prefix:"OK CERT")) {
        // generate AES session key and send encrypted by serverPub
        SecretKey ak = CryptoUtils.generateAESKey();
        sessionKey = ak;
        String encSession = CryptoUtils.rsaEncryptBase64(data:ak.getEncoded(), pub:serverPubKey);
        out.println("SESSIONKEY " + encSession);
        System.out.println("Server: " + in.readLine());
    }
    // Now we can send transaction
    String tx = "TRANSFER:to=Alice;amount=1000";
    String enc = CryptoUtils.aesGcmEncryptBase64(plain:tx.getBytes(), key:sessionKey);
    String mac = CryptoUtils.hmacSha256Base64(data:Base64.getDecoder().decode(enc), key:sessionKey);
    out.println("TRANSACT " + enc + ":" + mac);
    System.out.println("Server: " + in.readLine());
}
}

out.println("QUIT");
System.out.println("Server: " + in.readLine());
s.close();
}

public static void main(String[] args) throws Exception {
    ClientMain client = new ClientMain();
    client.runDemo();
}
}

```

```

import javax.crypto.*;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.util.Base64;

public class CryptoUtils {
    public static KeyPair generateRSAKeyPair() throws Exception {
        KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm: "RSA");
        kpg.initialize(keysize: 2048);
        return kpg.generateKeyPair();
    }

    public static SecretKey generateAESKey() throws Exception {
        KeyGenerator kg = KeyGenerator.getInstance(algorithm: "AES");
        kg.init(keysize: 256);
        return kg.generateKey();
    }

    public static String rsaEncryptBase64(byte[] data, PublicKey pub) throws Exception {
        Cipher cipher = Cipher.getInstance(transformation: "RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(mode: Cipher.ENCRYPT_MODE, key: pub);
        return Base64.getEncoder().encodeToString(cipher.doFinal(data));
    }

    public static byte[] rsaDecryptBase64(String b64, PrivateKey priv) throws Exception {
        Cipher cipher = Cipher.getInstance(transformation: "RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
        cipher.init(mode: Cipher.DECRYPT_MODE, key: priv);
        return cipher.doFinal(Base64.getDecoder().decode(b64));
    }

    // AES-GCM encrypt; returns Base64(iv|ciphertext|tag)
    public static String aesGcmEncryptBase64(byte[] plain, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance(transformation: "AES/GCM/NoPadding");
        SecureRandom rand = new SecureRandom();
        byte[] iv = new byte[12];
        rand.nextBytes(iv);
        GCMParameterSpec spec = new GCMParameterSpec(plaintextLength: 128, sourceIV: iv);
        cipher.init(mode: Cipher.ENCRYPT_MODE, key, parameters: spec);
        byte[] ct = cipher.doFinal(plain);
        byte[] out = new byte[iv.length + ct.length];
        System.arraycopy(source: iv, dest: out, destPos: 0, length: iv.length);
        System.arraycopy(source: ct, dest: out, destPos: iv.length, length: ct.length);
        return Base64.getEncoder().encodeToString(out);
    }

    // AES-GCM encrypt; returns Base64(iv|ciphertext|tag)
    public static String aesGcmEncryptBase64(byte[] plain, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance(transformation: "AES/GCM/NoPadding");
        SecureRandom rand = new SecureRandom();
        byte[] iv = new byte[12];
        rand.nextBytes(iv);
        GCMParameterSpec spec = new GCMParameterSpec(plaintextLength: 128, sourceIV: iv);
        cipher.init(mode: Cipher.ENCRYPT_MODE, key, parameters: spec);
        byte[] ct = cipher.doFinal(plain);
        byte[] out = new byte[iv.length + ct.length];
        System.arraycopy(source: iv, dest: out, destPos: 0, length: iv.length);
        System.arraycopy(source: ct, dest: out, destPos: iv.length, length: ct.length);
        return Base64.getEncoder().encodeToString(out);
    }

    public static byte[] aesGcmDecryptBase64(String b64, SecretKey key) throws Exception {
        byte[] all = Base64.getDecoder().decode(b64);
        byte[] iv = new byte[12];
        System.arraycopy(source: all, dest: iv, destPos: 0, length: 12);
        byte[] ct = new byte[all.length - 12];
        System.arraycopy(source: all, dest: ct, destPos: 12, destLen: 12, length: ct.length);
        Cipher cipher = Cipher.getInstance(transformation: "AES/GCM/NoPadding");
        GCMParameterSpec spec = new GCMParameterSpec(plaintextLength: 128, sourceIV: iv);
        cipher.init(mode: Cipher.DECRYPT_MODE, key, parameters: spec);
        return cipher.doFinal(ct);
    }

    public static String hmacSha256Base64(byte[] data, SecretKey key) throws Exception {
        Mac mac = Mac.getInstance(algorithm: "HmacSHA256");
        mac.init(key);
        return Base64.getEncoder().encodeToString(mac.doFinal(data));
    }

    public static SecretKey secretKeyFromBytes(byte[] raw) {
        return new SecretKeySpec(raw, algorithm: "AES");
    }

    // convenience
    public static String bytesToUtf8String(byte[] b) {
        return new String(b, StandardCharsets.UTF_8);
    }
}

```

```

import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map;

public class OTPService {
    private final Map<String, String> otpMap = new HashMap<>();
    private final SecureRandom rand = new SecureRandom();

    public String generateOTP(String username) {
        int code = 100000 + rand.nextInt( bound:900000 );
        String otp = Integer.toString( i:code );
        otpMap.put( key:username, value:otp );
        // In real system: send OTP via SMS/Email. Here we return it so test can use.
        return otp;
    }

    public boolean verifyOTP(String username, String otp) {
        String expected = otpMap.get( key:username );
        if (expected == null) return false;
        boolean ok = expected.equals( anObject:otp );
        if (ok) otpMap.remove( key:username );
        return ok;
    }
}

import java.security.*;
import java.security.spec.*;
import java.util.Base64;

public class PemUtils {
    public static String publicKeyToBase64(PublicKey pub) {
        return Base64.getEncoder().encodeToString( src:pub.getEncoded());
    }

    public static PublicKey publicKeyFromBase64(String b64) throws Exception {
        byte[] bytes = Base64.getDecoder().decode( src:b64 );
        X509EncodedKeySpec spec = new X509EncodedKeySpec( encodedKey:bytes );
        KeyFactory kf = KeyFactory.getInstance( algorithm:"RSA" );
        return kf.generatePublic( keySpec:spec );
    }

    public static PrivateKey privateKeyFromBase64(String b64) throws Exception {
        byte[] bytes = Base64.getDecoder().decode( src:b64 );
        PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec( encodedKey:bytes );
        KeyFactory kf = KeyFactory.getInstance( algorithm:"RSA" );
        return kf.generatePrivate( keySpec:spec );
    }
}

```

```

import javax.crypto.SecretKey;
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64;

public class ServerMain {
    private final int port = 9999;
    private final UserDatabase db = new UserDatabase();
    private final OTPService otpService = new OTPService();
    private final SimpleIDS ids = new SimpleIDS();
    private final KeyPair serverKeyPair;

    public ServerMain() throws Exception {
        serverKeyPair = CryptoUtils.generateRSAKeyPair();
        System.out.println("== SERVER PUBLIC KEY (BASE64) ===");
        System.out.println(PemUtils.publicKeyToBase64(serverKeyPair.getPublic()));
        System.out.println("== END SERVER PUBLIC KEY ===");
    }

    public void start() throws Exception {
        ServerSocket ss = new ServerSocket(port);
        System.out.println("Server listening on " + port);
        while (true) {
            Socket s = ss.accept();
            new Thread(() -> {
                try {
                    handleConnection(s);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}

```

```

private void handleConnection(Socket s) throws Exception {
    String remote = s.getInetAddress().getHostAddress();
    AuditLogger.log("Connection from " + remote);
    BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
    PrintWriter out = new PrintWriter(s.getOutputStream(), true);

    // Simple line-based RPC:
    String line;
    SecretKey sessionKey = null;
    String currentUser = null;
    while ((line = in.readLine()) != null) {
        if (line.equals("KEYEXCHANGE")) {
            String serverPubB64 = PemUtils.publicKeyToBase64(serverKeyPair.getPublic());
            out.println("OK PUBKEY " + serverPubB64);
            AuditLogger.log("Server Public Key sent to " + remote);
        } else if (line.startsWith("REGISTER ")) {
            String[] parts = line.substring(9).split(":");
            if (parts.length < 4) {
                out.println("ERROR bad-format");
            } else {
                String username = parts[0];
                String password = parts[1];
                String clientPubB64 = parts[2];
                String contact = parts[3];
                try {
                    db.register(username, password, clientPubB64, contact);
                    out.println("OK REGISTERED");
                    AuditLogger.log("User registered: " + username);
                } catch (Exception e) {
                    out.println("ERROR registration-failed");
                }
            }
        } else if (line.startsWith("LOGIN ")) {
            String payload = line.substring(6);
            // payload is encrypted with server public key; decrypt to get username:password
            byte[] dec = CryptoUtils.rsaDecryptBase64(payload, serverKeyPair.getPrivate());
            String creds = new String(dec);
            String[] cp = creds.split(":");
            String username = cp[0], password = cp[1];
            UserDatabase.User user = db.getUser(username);
            if (user == null) {
                out.println("ERROR user-not-found");
                ids.registerFailedLogin(ip, remote);
                continue;
            }
        }
    }
}

```

```

        continue;
    }
    if (!user.getPasswordHash().equalsAndPassword)) {
        out.println("ERROR bad-credentials");
        ids.registerFailedLogin(ip, remote);
        AuditLogger.log("Failed login for " + username);
    } else {
        // generate OTP and reply (in production OTP sent by SMS)
        String otp = otpService.generateOTP(username);
        AuditLogger.log("OTP generated for " + username);
        out.println("OTP " + otp); // for demo we send OTP back
        currentUser = username;
    }
} else if (line.startsWith("OTPVERIFY ")) {
    String[] p = line.substring(beginIndex+10).split(":");
    String username = p[0], otp = p[1];
    boolean ok = otpService.verifyOTP(username, otp);
    if (ok) {
        out.println("ERROR bad-otp");
        AuditLogger.log("Bad OTP for " + username);
    } else {
        // send server certificate base64 of public key
        String serverPub64 = PenUtils.publicKeyToBase64(pk.getServerKeyPair().getPublic());
        out.println("OK CERT " + serverPub64);
        AuditLogger.log("OTP verified for " + username);
    }
} else if (line.startsWith("SESSIONKEY ")) {
    String encKey64 = line.substring(beginIndex+11);
    // We decrypt using server private key to get raw AES bytes.
    byte[] aesRaw = CryptoUtils.rsaDecryptBase64(encKey64, priv.getServerKeyPair().getPrivate());
    sessionKey = CryptoUtils.secretKeyFromBytes(aesRaw);
    AuditLogger.log("Session key established for user " + currentUser);
    out.println("OK SESSION");
} else if (line.startsWith("TRANSACTION ")) {
    // TRANSACT encPayloadBase64:macBase64
    String payload = line.substring(beginIndex+9);
    String[] parts = payload.split(":");
    String enc = parts[0], mac = parts[1];
    if (sessionKey == null) { out.println("ERROR no-session"); continue; }
    // verify mac
    String computedMac = CryptoUtils.hmacSha256Base64(data.Base64.getDecoder().decode(enc), key=sessionKey);
    if (!computedMac.equals(mac)) {
        out.println("ERROR bad-mac");
        AuditLogger.log("Bad MAC for " + currentUser);
    }
    out.println("OK CERT " + serverPub64);
    AuditLogger.log("OTP verified for " + username);
}
else if (line.startsWith("SESSIONKEY ")) {
    String encKey64 = line.substring(beginIndex+11);
    // We decrypt using server private key to get raw AES bytes.
    byte[] aesRaw = CryptoUtils.rsaDecryptBase64(encKey64, priv.getServerKeyPair().getPrivate());
    sessionKey = CryptoUtils.secretKeyFromBytes(aesRaw);
    AuditLogger.log("Session key established for user " + currentUser);
    out.println("OK SESSION");
} else if (line.startsWith("TRANSACTION ")) {
    // TRANSACT encPayloadBase64:macBase64
    String payload = line.substring(beginIndex+9);
    String[] parts = payload.split(":");
    String enc = parts[0], mac = parts[1];
    if (sessionKey == null) { out.println("ERROR no-session"); continue; }
    // verify mac
    String computedMac = CryptoUtils.hmacSha256Base64(data.Base64.getDecoder().decode(enc), key=sessionKey);
    if (!computedMac.equals(mac)) {
        out.println("ERROR bad-mac");
        AuditLogger.log("Bad MAC for " + currentUser);
        continue;
    }
    byte[] plain = CryptoUtils.sessGcmDecryptBase64(enc, key=sessionKey);
    String body = new String(plain);
    AuditLogger.log("Transaction from " + currentUser + ": " + body);
    out.println("OK TRANSACTION EXECUTED");
} else if (line.equals("QUIT")) {
    out.println("BYE");
    s.close();
    return;
} else {
    out.println("ERROR unknown-command");
}
}
s.close();
}

public static void main(String[] args) throws Exception {
    ServerMain srv = new ServerMain();
    srv.start();
}
}

```

```

/*
public class SimpleIDS {
    // threshold-based detector
    private int failedLogins = 0;

    public void registerFailedLogin(String ip) {
        failedLogins++;
        if (failedLogins > 5) {
            System.out.println("[IDS] Alert: many failed login attempts from " + ip);
        }
    }

    public void reset() {
        failedLogins = 0;
    }
}

```

```

/*
import javax.crypto.SecretKey;
import java.security.PublicKey;
import java.util.HashMap;
import java.util.Map;

public class UserDatabase {
    // in-memory "db"
    static class User {
        String username;
        String passwordHash; // for demo: store clear or simple hash
        PublicKey clientPublicKey;
        SecretKey storedSessionKey;
        String phoneOrEmail;
        User(String u, String p, PublicKey pk, String contact){
            username = u; passwordHash = p; clientPublicKey = pk; phoneOrEmail = contact;
        }
    }
    private final Map<String, User> users = new HashMap<>();

    public void register(String username, String passwordHash, PublicKey clientPub, String contact){
        users.put(username, new User(username, passwordHash, clientPub, contact));
    }

    public User getUser(String username){
        return users.get(username);
    }
}

```

The implementation was tested with:

- Normal login and OTP validation.
- Encrypted fund transfer.

- Replay attack and invalid MAC attempts.

- Multiple failed logins triggering IDS alerts.

```

Output x BankSecurityDemo (run) x BankSecurityDemo (run) #Γ x

run:
== SERVER PUBLIC KEY (BASE64) ==
MIIBIjANBgkqhkiG9w0BAQEFAOAQ8AMIIIBgKCAQEArfaFrqtMOD5g7X+CwVko12RUTHR7Y6RcdWKz6fbDvYhDX4EaYuv9PC8TDuVKrfgWOCJROGj36+RJCcMyNtEyqV20RW7xRkZfJzAxLc8j52RzUgdN/8vZNAWu(
== END SERVER PUBLIC KEY ==
Server listening on 9999
[AUDIT 2025-11-13T15:02:56.9534309] Connection from 127.0.0.1
[AUDIT 2025-11-13T15:02:56.900521] Server Public Key sent to 127.0.0.1
[AUDIT 2025-11-13T15:02:57.1450897] User registered: lama
[AUDIT 2025-11-13T15:02:57.5042716] OTP generated for lama
[AUDIT 2025-11-13T15:02:57.5073385] OTP verified for lama
[AUDIT 2025-11-13T15:02:57.5278312] Session key established for user lama
[AUDIT 2025-11-13T15:02:57.5535104] Transaction from lama: TRANSFER:to=Alice;amount=1000
.

Output x BankSecurityDemo (run) x BankSecurityDemo (run) #Γ x

run:
Server KeyExchange: OK PUBKEY MIIBIjANBgkqhkiG9w0BAQEFAOAQ8AMIIIBgKCAQEArfaFrqtMOD5g7X+CwVko12RUTHR7Y6RcdWKz6fbDvYhDX4EaYuv9PC8TDuVKrfgWOCJROGj36+RJCcMyNtEyqV20RW7xRkZfJzAxLc8j!
>>> Successfully fetched and set Server Public Key.
Server: OK REGISTERED
Server: OTP 136011
Server: OK CERT MIIBIjANBgkqhkiG9w0BAQEFAOAQ8AMIIIBgKCAQEArfaFrqtMOD5g7X+CwVko12RUTHR7Y6RcdWKz6fbDvYhDX4EaYuv9PC8TDuVKrfgWOCJROGj36+RJCcMyNtEyqV20RW7xRkZfJzAxLc8j!
Server: OK SESSION
Server: OK TRANSC EXECUTED
Server: BYE
BUILD SUCCESSFUL (total time: 7 seconds)

```

SECURITY ANALYSIS

Positive aspects:

- Mutual authentication: If keys and certificates are kept secure, impersonation is prevented. Both the client and the server use RSA key pairs and certificates to verify their identities.
- Two-factor authentication (2FA), which lowers the risk of password theft by adding a possession factor in addition to knowledge (password) through out-of-band (SMS/email) OTP delivery.
- Session confidentiality and integrity: HMACs/MACs safeguard the integrity and authenticity of messages, while a new AES-256 session key (created under RSA protection) encrypts traffic. Integrity and confidentiality are combined into one primitive when AES-GCM is used.
- Audit logging and IDS: Forensics records events (such as logins, OTPs, and transactions); a basic IDS generates alerts about anomalous patterns, like repeatedly unsuccessful login attempts.
- Less privilege and timeouts: Per-transaction signatures and brief session lifetimes shorten the lifespan of credentials that are stolen.

Residual Risks and Mitigation:

- Key compromise: If a private RSA key (client or server) is leaked, attacker can impersonate. Mitigation: store private keys in HSM or OS key store, rotate keys periodically, support revocation lists and CRLs/OCSP for certificates.
- OTP interception: SMS OTPs can be intercepted via SIM swap or network attacks. Mitigation: prefer app-based OTP (TOTP), push notifications, or hardware tokens for high-value transactions. Implement anti-SIM-swap detection and customer alerts.
- Phishing / Social engineering: Attackers trick users into revealing credentials or OTPs. Mitigation: user education, phishing-resistant authentication (FIDO2/WebAuthn), transaction verification dialogues that show recently used device fingerprints.
- Replay and man-in-the-middle: Without strong certificate validation and nonce/timestamp usage, replay or MITM might succeed. Mitigation: require signed timestamps/nonces, strict certificate chain validation, and binding session keys to client identifiers and TLS.

- Insider threats and data leakage: Insider misuse of DB or logs. Mitigation: role-based access controls, log tamper-evidence (append-only logs), data encryption at rest, least privilege for DB access.
- Application-level flaws (SQL injection, XSS, CSRF): May allow data exfiltration or session theft. Mitigation: prepared statements, input validation, CSP, anti-CSRF tokens, secure cookies (HttpOnly, Secure, SameSite).
- DoS attacks: Can affect availability. Mitigation: rate limiting, upstream DDoS protection (CDN or WAF), autoscaling, circuit breakers.

DISCUSSION

The project demonstrates that online banking system security improves substantially through the implementation of MFA and advanced encryption and continuous monitoring systems. The testing results confirmed that the proposed protocol maintains both confidential-

ity and integrity even when cyber threats occur repeatedly. The model demonstrates strong security capabilities but users need to stay aware about authentication methods for successful implementation. The system achieves an excellent security-to-usability ratio based on all provided information.

CONCLUSION

The project developed an online banking system which protects user data and financial operations through its combination of powerful encryption and secure verification methods and continuous system surveillance. The system proved successful in defending against typical security threats which include phishing attacks and data modification attempts and unauthorized system entry attempts. The security features provide enhanced protection but users must remain vigilant and follow correct procedures for system operation. The proposed protocol provides an excellent framework to build secure and dependable online banking applications.

REFERENCES

- [1]https://journals.ekb.eg/article_396654_b1c4814ae65405ce058d3c144bae810f.pdf
- [2]<https://ijcrt.org/papers/IJCRT2409786.pdf>
- [3]<https://pdfs.semanticscholar.org/da46/855de9412168081390efa0c34a626b40ef73.pdf>
- [4]https://www.researchgate.net/publication/280864086_Security_in_Online_Banking_Services_-A_Comparative_Study
- [5]https://www.researchgate.net/publication/351980335_Security_Issues_of_Electronic_and_Mobile_Banking
- [6]<https://mpra.ub.uni-muenchen.de/71749/>