Implementing Zero-knowledge Proof Authentication in Android

Jagadeesh Duggirala

Software Engineer,
Mercari, Japan
jag4364u@gmail.com

Abstract
Zero-knowledge proofs (ZKPs) allow one party to prove to another that they know a secret without revealing the secret itself. In the context of authentication, ZKPs can enhance security by enabling passwordless and secure identity verification. This paper discusses the theory behind ZKPs, their application in Android authentication, and practical implementation details, highlighting the advantages and challenges of using ZKPs in mobile applications.

Keywords: Zero-Knowledge Proof, Android, Authentication, Cryptography, Passwordless Login, Security, Privacy

1. Introduction

- **Overview**: Zero-knowledge proofs (ZKP) are cryptographic techniques that allow one party (the prover) to prove to another party (the verifier) that they possess certain knowledge without revealing the information itself. ZKPs are increasingly being used in various fields, including blockchain, privacy-preserving authentication, and identity verification.
- **Motivation for ZKPs in Authentication**: Traditional authentication methods, like passwords, are vulnerable to various attacks, such as phishing, brute force, and data breaches. ZKPs provide a more secure and privacy-preserving alternative.
- **Purpose of the Paper**: This paper explores how ZKPs can be implemented in Android applications for secure authentication, focusing on their practical usage, implementation steps, and real-world applications.
- **Structure of the Paper**: This paper is divided into sections that explain the fundamentals of ZKPs, explore authentication mechanisms, and provide an in-depth guide on implementing ZKPs in Android applications.

2. Fundamentals of Zero-knowledge Proofs

- **Definition of Zero-knowledge Proofs**: A Zero-knowledge proof is a cryptographic protocol through which one party can prove to another that they know a piece of information without revealing it.
- **Key Properties of ZKPs**:
  - *Completeness*: If the statement is true, an honest prover can convince the verifier.
  - *Soundness*: If the statement is false, no prover can convince the verifier.
  - *Zero-knowledge*: The verifier learns nothing about the secret except that the statement is true.
- **Types of Zero-knowledge Proofs**:
  - Interactive ZKPs: Require multiple rounds of communication between prover and verifier.

    - Non-interactive ZKPs: Require only a single message from the prover to the verifier, making them more suitable for real-world applications like authentication.
- **Use Cases for ZKPs in Authentication**:
    - Verifying a user's identity without transmitting sensitive information.
    - Passwordless authentication.
    - Blockchain-based authentication systems.

3. Cryptographic Foundations of Zero-knowledge Proofs

- **Mathematical Foundations**: ZKPs rely heavily on advanced cryptographic techniques such as elliptic curve cryptography, hash functions, and commitment schemes.
- **Commitment Schemes**: A commitment scheme allows a user to commit to a value without revealing it, ensuring that they cannot change it later. This is crucial in ZKP to ensure that the prover cannot alter the proof after it is made.
- **Hash Functions and ZKPs**: Cryptographic hash functions are used in ZKPs to ensure data integrity and make it computationally infeasible to reverse or forge the proof.
- **Elliptic Curve Cryptography (ECC)**: ECC is a key component in many ZKP systems, such as zk-SNARKs, which offer efficient proofs for complex statements, and can be used in mobile devices for authentication.

4. Zero-knowledge Proof Authentication Mechanisms

- **Traditional Authentication Methods**: Usernames and passwords, multi-factor authentication (MFA), and biometric-based authentication.
- **Limitations of Traditional Authentication**:
    - Vulnerability to password breaches.
    - High reliance on centralized systems.
    - Privacy concerns with biometric data storage.
- **ZKP-based Authentication**:
    - Passwordless authentication using ZKPs.
    - Secure challenge-response mechanisms: The server sends a challenge to the client, which proves the knowledge of the secret without revealing it.
    - Examples of ZKP authentication protocols: zk-SNARKs, zk-STARKs, and Bulletproofs.
    - *Authentication Flow*: The prover (user) generates a proof that they know a secret (such as a password or private key) without transmitting the secret itself to the verifier (server).

5. Implementing ZKP Authentication in Android Applications

Implementing Zero-Knowledge Proof (ZKP) authentication involves a sequence of cryptographic steps and secure communication between the Android client and the server. Below is a comprehensive guide on how to implement ZKP-based authentication in an Android application.

Step 1: Setting Up the Development Environment

Before starting the actual implementation, you’ll need to set up your development environment.

- **Prerequisites**:
    - Install **Android Studio**, which is the official IDE for Android development.
    - Use **Kotlin** or **Java** for development. Kotlin is generally recommended due to its modern syntax and safety features.
    - Set up dependencies for cryptographic operations.
- **Dependencies**:

BouncyCastle: A widely used cryptographic library that provides the necessary cryptographic primitives (e.g., elliptic curve operations, hashing). Add this dependency to your build.gradle file:

```
dependencies {
   implementation 'org.bouncycastle:bcprov-jdk15on:1.68'
   implementation 'org.bouncycastle:bcpg-jdk15on:1.68'
}
```

Libsodium: For secure cryptographic operations such as hashing and encryption (optional, but provides enhanced cryptographic support).

Step 2: Creating the ZKP Prover and Verifier

The core of ZKP authentication is the interaction between the *prover* (the client, i.e., the Android app) and the *verifier* (the server). We need to define both roles:

- **Prover (Android App)**:
    - The prover (Android app) generates a proof to show it knows a secret (e.g., a password or PIN) without revealing it to the server.
    - The app will use cryptographic operations such as hashing and elliptic curve encryption to generate a proof.
- **Verifier (Server)**:
    - The server receives the proof and verifies it without knowing the secret.
    - The server will check the proof using preconfigured cryptographic parameters and return either a success or failure message.

Prover's Side (Android Application)

The prover's role in ZKP is to prove knowledge of a secret (e.g., a PIN or password) using cryptographic operations, typically involving a commitment scheme.

1. **Generating a Commitment**:
   In a basic ZKP setup, the prover commits to a secret (e.g., a password) without revealing it. This is done using a cryptographic commitment scheme such as a hash function.

```
import org.bouncycastle.crypto.digests.SHA256Digest
import org.bouncycastle.crypto.engines.AESFastEngine
import org.bouncycastle.crypto.params.KeyParameter

// Example of committing to a secret
fun generateCommitment(secret: String): String {
    val digest = SHA256Digest()
    val secretBytes = secret.toByteArray(Charsets.UTF_8)
    digest.update(secretBytes, 0, secretBytes.size)
    val commitment = ByteArray(digest.digestSize)
    digest.doFinal(commitment, 0)
    return commitment.joinToString("") { String.format("%02x", it) }  // Return hex string of
commitment
}
```

2. **Generating the ZKP Proof**: Using elliptic curve cryptography or other cryptographic primitives, the prover will generate a proof. For simplicity, let's assume we use a basic ZKP protocol, where the prover generates a challenge-response pair:

    - The prover generates a random challenge value (random number).
    - The prover computes a response by using the challenge, secret, and commitment.

```
fun generateZKPProof(secret: String): Pair<String, String> {
    // Assume we have a precomputed commitment to the secret
    val commitment = generateCommitment(secret)

    // Generate a random challenge (e.g., a nonce)
    val challenge = (1..1000).random().toString()

    // Compute a response using secret and challenge
    val response = (challenge + secret + commitment).hashCode().toString()

    return Pair(challenge, response)  // Return challenge and response as proof
}
```

Verifier's Side (Server)

On the server side, the verification process will involve checking the proof received from the prover (Android app). The server will check that the response corresponds to the commitment and challenge.

1. **Verifying the Proof**: The server compares the provided response with its own computation using the secret and challenge. If they match, the proof is valid.

```
fun verifyZKPProof(secret: String, commitment: String, challenge: String, response: String):
Boolean {
    // Recompute the response on the server side
    val expectedResponse = (challenge + secret + commitment).hashCode().toString()
    return expectedResponse == response  // Check if response matches
}
```

2. **Server-side Verification API**: In a real-world setup, the server would receive the challenge, response, and commitment, and perform the above verification using a secure API.

Step 3: ZKP Protocol Implementation

Once the basic ZKP operations are in place, you'll need to implement the challenge-response protocol between the client and the server.

- **Client-Side (Android)**:
    - When the user attempts to authenticate, the Android app generates a ZKP proof (as described in Step 2) and sends the challenge and response to the server.
    - Securely transmit the proof using HTTPS or TLS to ensure data privacy.
- **Server-Side**:
    - The server receives the challenge and response from the client and verifies the proof using the procedure outlined above.
    - If the proof is valid, the server authenticates the user; otherwise, authentication fails.

Step 4: Integrating with Android UI

1. **User Interface**:
    - Create a simple authentication screen where users can enter their password or PIN. This will be used to generate the proof.
    - Provide appropriate feedback if authentication is successful or failed.

```
// Example of UI interaction in MainActivity
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val passwordInput = findViewById<EditText>(R.id.passwordInput)
        val loginButton = findViewById<Button>(R.id.loginButton)
        loginButton.setOnClickListener {
            val password = passwordInput.text.toString()
            // Generate proof
            val (challenge, response) = generateZKPProof(password)
            // Send proof to server
            sendProofToServer(challenge, response)
        }
    }
    // Method to send proof to server (simulated here)
    private fun sendProofToServer(challenge: String, response: String) {
        // Send the challenge and response to the server for verification
        // Server-side verification logic would be implemented as shown above
    }
}
```

6. Security and Privacy Considerations

- **Data Privacy**: ZKPs enhance privacy by ensuring sensitive information (e.g., passwords) is not transmitted over the network.
- **Server-side Security**: The server must securely store cryptographic keys and handle challenges properly to prevent replay attacks.
- **Mobile Device Constraints**: Mobile devices may have limited computational resources for heavy cryptographic operations, which could impact performance. Efficient cryptographic algorithms and optimization techniques are necessary for real-time applications.

7. Conclusion

Zero-knowledge proofs offer a promising solution for secure, privacy-preserving authentication on Android applications. By implementing a ZKP-based authentication system, developers can reduce the risk of data breaches and provide users with a more secure and seamless login experience. However, challenges related to cryptographic complexity, mobile device limitations, and network security must be addressed to ensure a robust and scalable solution.

References

1. Rivest, R. L., Shamir, A., & Adleman, L. (1978). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. Communications of the ACM.
2. Bellare, M., & Rogaway, P. (1993). *Random oracles are practical: A paradigm for designing efficient protocols*. ACM SIGACT News.

3. Goldwasser, S., Micali, S., & Rackoff, C. (1985). *The Knowledge Complexity of Interactive Proof Systems*. SIAM Journal on Computing.