Università degli studi di Messina
DipartimentodiScienze Matematiche, Informatiche,Scienze Fisiche e Scienze della Terra

Course: Data Analysis

Project on “System Security”

by

SultanovDair(551930)

MESSINA 2026

X3DH & Double Ratchet Implementation in Go

Student: Dair Sultanov
Project Repository:https://github.com/ItzDair/securechat

1. Introduction

Modern messaging security requires more than just a single encrypted tunnel; it requires Forward Secrecy and Post-Quantum Resistance concepts like the Double Ratchet. This project implements a secure asynchronous key-agreement layer and a message-chaining system.

The goal of this project was to:

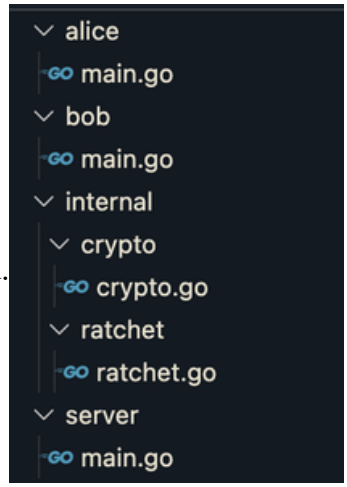
- Implement a Redis-backed relay server for asynchronous messaging.
- Simulate the X3DH Handshake to establish a shared secret between two parties who are not online at the same time.
- Implement a Double Ratchet (DH + Symmetric Ratchet) to ensure that if one message key is compromised, future and past messages remain secure.

2. System Architecture

The system follows a "Client-Server-Client" model where the server acts as an untrusted intermediary.

Relay Server (server.go):

- Powered by Redis for high-performance key-value storage.
- Handles upload_prekey and send_secure endpoints.
- Stores "Mailboxes" (message queues) for offline users.



Cryptographic Engine (internal/crypto/crypto.go)

- X25519: Elliptic Curve Diffie-Hellman (ECDH) for key exchange.
- HKDF-SHA256: For key derivation and "ratcheting" the chain.
- AES-256-GCM: For authenticated encryption of the message payload.

Ratchet Logic (internal/ratchet/ratchet.go)

- Manages the state of the RootKey and ChainKey.
- Updates the keys after every message to ensure Forward Secrecy.

3. Protocol Walkthrough

3.1 Key Generation & Registration (Bob)

- Bob generates an Identity Keypair (IK_b).
- Bob uploads his Public Identity Key to the Redis server via /upload_prekey.
- The server stores this bundle, making Bob "discoverable" by Alice.

```
pass1234@Dair-Sultanovs-MacBook-Pro securechat % go run bob/main.go
[BOB] Generating Bob X25519 identity keypair...
[BOB] Uploading Bob public key to server (stored in Redis as prekey:bob)...
[BOB] Bob ready. Polling mailbox from server (secure_mailbox:bob in Redis)...
```

3.2 The Handshake & First Message (Alice)

- Alice fetches Bob's public key from the server.
- Alice generates an Ephemeral Keypair (EK_a).
- DH Computation: Alice computes $SharedSecret = DH(EK_a, IK_b)$.
- Initial Ratchet: Alice initializes her Ratchet state using the Shared Secret and generates the first MessageKey.
- Encryption: The message is encrypted using AES-GCM and sent to the server.

```
pass1234@Dair-Sultanovs-MacBook-Pro securechat % go run alice/main.go
[ALICE] Type a message to Bob and press Enter:
```

```
pass1234@Dair-Sultanovs-MacBook-Pro securechat % go run alice/main.go
[ALICE] Type a message to Bob and press Enter: Message for report
[ALICE] Fetching Bob prekey from server...
[ALICE] Bob prekey received.
[ALICE] Generating Alice ephemeral X25519 keypair...
[ALICE] Computing shared secret (X25519 DH)...
[ALICE] Initializing ratchet and deriving message key...
[ALICE] Encrypting message with AES-GCM...
[ALICE] Sending encrypted message to server (queued for bob in Redis)...
[ALICE] Done. Message sent.
```

3.3 The Ratchet Step (Bob)

- Bob retrieves the message and Alice's Ephemeral Key.
- Bob reconstructs the Shared Secret: $DH(IK_b, EK_a)$.
- Decryption: Bob derives the same MessageKey to decrypt the text.
- Ratchet Step: Both parties update their internal ChainKey so the next message uses a brand-new, unique key.

```
pass1234@Dair-Sultanovs-MacBook-Pro securechat % go run bob/main.go
[BOB] Generating Bob X25519 identity keypair...
[BOB] Uploading Bob public key to server (stored in Redis as prekey:bob)...
[BOB] Bob ready. Polling mailbox from server (secure_mailbox:bob in Redis)...
[BOB] Received encrypted message. Deriving shared secret (X25519 DH)...
[BOB] Initializing ratchet and deriving message key...
[BOB] Decrypting with AES-GCM...
[BOB] Bob received: Message for report
```

Alice's workflow

```
fmt.Println("[ALICE] Fetching Bob prekey from server...")
resp, err := http.Get("http://localhost:8080/prekey?user=bob")
if err != nil {
    fmt.Println("[ALICE] Error fetching prekey:", err)
    os.Exit(1)
}
defer resp.Body.Close()
if resp.StatusCode != http.StatusOK {
    fmt.Println("[ALICE] Server did not return Bob prekey. Status:", resp.Status)
    fmt.Println("[ALICE] Run `go run ./bob` first to upload Bob's key.")
    os.Exit(1)
}
var bundle PrekeyBundle
if err := json.NewDecoder(resp.Body).Decode(&bundle); err != nil {
    fmt.Println("[ALICE] Error decoding prekey bundle:", err)
    os.Exit(1)
}
if len(bundle.IdentityKey) != 32 {
    fmt.Println("[ALICE] Invalid Bob identity key length:", len(bundle.IdentityKey))
    os.Exit(1)
}
fmt.Println("[ALICE] Bob prekey received.")
```

- Alice calls the server endpoint /prekey?user=bob to retrieve Bob's public key (stored earlier when Bob ran his client).
- It decodes JSON into PrekeyBundle and checks the key length is 32 bytes (X25519 public key size).
- If Bob hasn't uploaded yet, the server returns non-200 and Alice exits.

```
fmt.Println("[ALICE] Generating Alice ephemeral X25519 k
alicePriv, alicePub := crypto.GenerateKeyPair()
```

- Creates a fresh X25519 private/public keypair.
- alicePriv is kept secret; alicePub can be shared.

```
fmt.Println("[ALICE] Computing shared secret (X25519 DH)...")
shared := crypto.DH(alicePriv, bundle.IdentityKey)
fmt.Println("[ALICE] Initializing ratchet and deriving message key...")
r := ratchet.NewRatchet(shared, bundle.IdentityKey)
msgKey := r.NextMessageKey()
```

- crypto.DH(alicePriv, bobPub) computes a shared secret that only Alice and Bob can derive.
- ratchet.NewRatchet(...) uses HKDF to turn that shared secret into "root/chain" keys.
- NextMessageKey() advances the chain and produces a one-time AES key (msgKey) for this message.

```
msg := SecureMessage{
    FromIdentity: alicePub,
    EphemeralKey: alicePub,
    Nonce:        nonce,
    Ciphertext:   ciphertext,
}
```

This packages what Bob needs to decrypt:

- EphemeralKey (Alice public key used in DH),
- Nonce and Ciphertext.
- Note: here FromIdentity and EphemeralKey are the same value (alicePub) in this code.

```
fmt.Println("[ALICE] Sending encrypted message to server (queued for bob in Redis)...")
data, err := json.Marshal(msg)
if err != nil {
    fmt.Println("[ALICE] Error encoding secure message:", err)
    os.Exit(1)
}
sendResp, err := http.Post("http://localhost:8080/send_secure?to=bob",
    "application/json", bytes.NewBuffer(data))
if err != nil {
    fmt.Println("[ALICE] Error sending secure message:", err)
    os.Exit(1)
}
defer sendResp.Body.Close()
if sendResp.StatusCode != http.StatusOK {
    body, _ := io.ReadAll(sendResp.Body)
    fmt.Println("[ALICE] Server rejected message. Status:", sendResp.Status, "Body:", string(body))
    os.Exit(1)
}
```

- Serializes SecureMessage to JSON and POSTs it to /send_secure?to=bob.
- The server does not decrypt; it just stores the JSON blob in Bob's mailbox (Redis list) for later retrieval.

Bob's workflow

```go
func main() {
    // Bob identity
    fmt.Println("[BOB] Generating Bob X25519 identity keypair...")
    bobPriv, bobPub := crypto.GenerateKeyPair()
```

- Bob creates an X25519 private/public keypair.
- bobPriv stays on Bob's side and is used later to compute the shared secret.
- bobPub is safe to upload so others (Alice) can derive a shared secret with Bob.

```go
fmt.Println("[BOB] Uploading Bob public key to server (stored in Redis as prekey:bob)...")
bundle := PrekeyBundle{IdentityKey: bobPub}
data, err := json.Marshal(bundle)
if err != nil {
    fmt.Println("[BOB] Error encoding prekey bundle:", err)
    os.Exit(1)
}
resp, err := http.Post("http://localhost:8080/upload_prekey?user=bob",
    "application/json", bytes.NewBuffer(data))
if err != nil {
    fmt.Println("[BOB] Error uploading prekey bundle:", err)
    os.Exit(1)
}
defer resp.Body.Close()
if resp.StatusCode != http.StatusOK {
    body, _ := io.ReadAll(resp.Body)
    fmt.Println("[BOB] Server rejected prekey upload. Status:", resp.Status, "Body:", string(body))
    os.Exit(1)
}
```

- Bob wraps his public key in JSON (PrekeyBundle) and POSTs it to /upload_prekey?user=bob.
- The server stores it in Redis under prekey:bob so Alice can fetch it later.

```go
fmt.Println("[BOB] Bob ready. Polling mailbox from server (secure_mailbox:bob in Redis)...")

for {
    resp, err := http.Get("http://localhost:8080/fetch_secure?user=bob")
    if err != nil {
        fmt.Println("[BOB] Error fetching secure message:", err)
        time.Sleep(500 * time.Millisecond)
        continue
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        time.Sleep(500 * time.Millisecond)
        continue
    }

    var msg SecureMessage
    if err := json.NewDecoder(resp.Body).Decode(&msg); err != nil {
        resp.Body.Close()
        time.Sleep(300 * time.Millisecond)
        continue
    }
    resp.Body.Close()
    if len(msg.Ciphertext) == 0 {
        time.Sleep(300 * time.Millisecond)
        continue
    }
```

- Bob repeatedly calls /fetch_secure?user=bob to check his mailbox.
- It decodes the JSON into SecureMessage.
- If the server returns {} (no message), Ciphertext is empty, so Bob sleeps briefly and keeps polling.

```go
fmt.Println("[BOB] Received encrypted message. Deriving shared secret (X25519 DH)...")
shared := crypto.DH(bobPriv, msg.EphemeralKey)
fmt.Println("[BOB] Initializing ratchet and deriving message key...")
r := ratchet.NewRatchet(shared, msg.EphemeralKey)
msgKey := r.NextMessageKey()
```

- Bob computes the same DH shared secret using his private key (bobPriv) and Alice's public key (msg.EphemeralKey).
- He runs the same ratchet derivation so he ends up with the same msgKey Alice used to encrypt.

```go
    fmt.Println("[BOB] Decrypting with AES-GCM...")
    plaintext := crypto.Decrypt(msgKey, msg.Nonce, msg.Ciphertext)
    fmt.Println("[BOB] Bob received:", string(plaintext))
    break
}
```

- Uses the derived msgKey plus the transmitted nonce to decrypt ciphertext.
- Prints the plaintext and exits the loop after successfully receiving one message.

Redis-server workflow

```go
func main() {
    rdb = redis.NewClient(&redis.Options{Addr: "localhost:6379"})
    if err := rdb.Ping(ctx).Err(); err != nil {
        fmt.Println("[SERVER] Redis is not reachable at localhost:6379:", err)
        os.Exit(1)
    }
```

- Creates a Redis client pointing at localhost (line 6379).
- PING is a startup check; if Redis isn't running, the server exits immediately.

```go
http.HandleFunc("/upload_prekey", uploadPrekey)
http.HandleFunc("/prekey", getPrekey)
http.HandleFunc("/send_secure", sendSecure)
http.HandleFunc("/fetch_secure", fetchSecure)
```

- Exposes 4 routes:
- upload a public key (/upload_prekey)
- fetch a public key (/prekey)
- send an encrypted message (/send_secure)
- fetch an encrypted message (/fetch_secure)
- Listens on port 8080.

```go
func uploadPrekey(w http.ResponseWriter, r *http.Request) {
    user := r.URL.Query().Get("user")
    defer r.Body.Close()
    var bundle PrekeyBundle
    if err := json.NewDecoder(r.Body).Decode(&bundle); err != nil {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("invalid JSON"))
        return
    }
    if len(bundle.IdentityKey) != 32 {
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("invalid identity_key length"))
        return
    }
    data, err := json.Marshal(bundle)
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("marshal error"))
        return
    }
    if err := rdb.Set(ctx, "prekey:"+user, data, 0).Err(); err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("redis error"))
        return
    }
    fmt.Println("[SERVER] Stored prekey bundle for", user, "(Redis key: prekey:"+user+")")
    w.Write([]byte("OK"))
}
```

- Reads ?user=bob and JSON body { "identity_key": ... }.
- Validates the key is 32 bytes (X25519 public key size).
- Stores it in Redis under prekey:bob (no expiration because TTL is 0).

```go
func getPrekey(w http.ResponseWriter, r *http.Request) {
    user := r.URL.Query().Get("user")
    val, err := rdb.Get(ctx, "prekey:"+user).Result()
    if err != nil {
        fmt.Println("[SERVER] Prekey not found for", user, "(Redis key: prekey:"+user+")")
        w.WriteHeader(404)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    w.Write([]byte(val))
    fmt.Println("[SERVER] Served prekey bundle for", user)
}
```

- Looks up prekey:<user> in Redis (example: prekey:bob).
- If missing, returns HTTP 404.
- If found, returns the stored JSON as-is.

```go
func fetchSecure(w http.ResponseWriter, r *http.Request) {
    user := r.URL.Query().Get("user")
    val, err := rdb.RPop(ctx, "secure_mailbox:"+user).Result()
    if err == redis.Nil {
        w.Header().Set("Content-Type", "application/json")
        w.Write([]byte("{}"))
        return
    }
    if err != nil {
        w.WriteHeader(http.StatusInternalServerError)
        w.Write([]byte("redis error"))
        return
    }
    w.Header().Set("Content-Type", "application/json")
    w.Write([]byte(val))
    fmt.Println("[SERVER] Delivered 1 secure message to", user)
}
```

- Pops from the right of secure_mailbox:<user> (example: secure_mailbox:bob).
- If the list is empty (redis.Nil), it returns {} (so Bob keeps polling).
- If there's a message, it returns the JSON blob to the client.

Ratchet.go

```go
type Ratchet struct {
    RootKey    []byte
    ChainKey   []byte
    DHPriv     []byte
    DHPub      []byte
    RemotePub  []byte
}
```

- RootKey: “master” key material used to derive/refresh the chain.
- ChainKey: evolving key you advance to derive per-message keys.
- DHPriv/DHPub: this side’s current DH keypair (for doing DH again if you “ratchet step”).
- RemotePub: the other side’s public key used for DH.

```go
func NewRatchet(sharedSecret, remotePub []byte) *Ratchet {
    priv, pub := crypto.GenerateKeyPair()
    root := crypto.HKDF(sharedSecret, "root")
    chain := crypto.HKDF(root, "chain")
    return &Ratchet{
        RootKey:   root,
        ChainKey:  chain,
        DHPriv:    priv,
        DHPub:     pub,
        RemotePub: remotePub,
    }
}
```

- Takes sharedSecret (from X25519 DH) and runs HKDF to derive:
- RootKey = HKDF(sharedSecret, "root")
- ChainKey = HKDF(RootKey, "chain")
- Also generates a fresh DH keypair and stores the other party’s public key.

```go
func (r *Ratchet) RatchetStep() {
    dhOut := crypto.DH(r.DHPriv, r.RemotePub)
    r.RootKey = crypto.HKDF(dhOut, "root")
    r.ChainKey = crypto.HKDF(r.RootKey, "chain")
    r.DHPriv, r.DHPub = crypto.GenerateKeyPair()
}
```

- Computes a new DH output dhOut.
- Re-derives RootKey and ChainKey from that new DH.
- Rotates its DH keypair for the next step.
- In your current Alice/Bob flow, you don’t call RatchetStep()—you only use NextMessageKey() once.

```go
func (r *Ratchet) NextMessageKey() []byte {
    r.ChainKey = crypto.HKDF(r.ChainKey, "chain-step")
    return crypto.HKDF(r.ChainKey, "msg")
}
```

- Advances ChainKey forward (chain-step).
- Derives a 32-byte message key (msg) used for AES-GCM.

Crypto.go

```go
func GenerateKeyPair() (priv, pub []byte) {
    priv = make([]byte, 32)
    rand.Read(priv)
    pub, _ = curve25519.X25519(priv, curve25519.Basepoint)
    return
}

// X25519 DH
func DH(priv, pub []byte) []byte {
    shared, _ := curve25519.X25519(priv, pub)
    return shared
}
```

- GenerateKeyPair() creates a random 32-byte private key and computes the X25519 public key.
- DH() computes the shared secret from (your private key, their public key). Alice and Bob get the same shared if they use matching keys.

```go
func HKDF(secret []byte, info string) []byte {
    h := hkdf.New(sha256.New, secret, nil, []byte(info))
    out := make([]byte, 32)
    io.ReadFull(h, out)
    return out
}
```

- HKDF turns some input secret into a “clean” derived 32-byte key.
- info (“root”, “chain”, “msg”, etc.) separates different derived keys so you don’t reuse the same bytes for different purposes.

```go
func Encrypt(key, plaintext []byte) (nonce, ciphertext []byte) {
    block, _ := aes.NewCipher(key)
    gcm, _ := cipher.NewGCM(block)
    nonce = make([]byte, gcm.NonceSize())
    rand.Read(nonce)
    ciphertext = gcm.Seal(nil, nonce, plaintext, nil)
    return
}

// AES-GCM decrypt
func Decrypt(key, nonce, ciphertext []byte) []byte {
    block, _ := aes.NewCipher(key)
    gcm, _ := cipher.NewGCM(block)
    plaintext, _ := gcm.Open(nil, nonce, ciphertext, nil)
    return plaintext
}
```

- Encrypt() uses AES-GCM with a random nonce to produce ciphertext (includes authentication tag).
- Decrypt() verifies the tag; if verification fails, gcm.Open returns an error (your code currently ignores it and returns nil plaintext in that case).

Conclusion

This project successfully demonstrates a high-performance, asynchronous messaging core tailored for modern decentralized communication. By combining Go’s lightweight concurrency model with Redis’s low-latency data structures and X25519’s elliptic curve security, the system achieves a robust balance between performance and privacy.

- Go & Concurrency: Utilizing Go's net/http and context packages allows the server to handle thousands of concurrent "mailbox" requests with minimal memory overhead, which is critical for scaling a messaging backend.
- Redis as a Message Broker: By offloading the state to Redis, the application server remains stateless. This allows for horizontal scaling and ensures that even if the app server restarts, the "Pre-key" bundles and queued messages remain persistent.
- X25519 Efficiency: The choice of Curve25519 provides 128-bit security with significantly shorter keys and faster computation times compared to traditional RSA or older ECC curves.