IPBeja
INSTITUTO POLITÉCNICO DE BEJA

INSTITUTO POLITÉCNICO DE BEJA
Escola Superior de Tecnologia e Gestão
Licenciatura em Engenharia Informática
Projeto Final de Curso

Desenvolvimento de um sistema de comunicações seguras

Martinho José Novo Caeiro - 23917
Paulo António Tavares Abade - 23919
Rafael Conceição Narciso - 24473

IPBeja Escola Superior de Tecnologia e Gestão

Beja, julho de 2025

INSTITUTO POLITÉCNICO DE BEJA
Escola Superior de Tecnologia e Gestão
Licenciatura em Engenharia Informática
Projeto Final de Curso

Desenvolvimento de um sistema de comunicações seguras

Martinho José Novo Caeiro - 23917
Paulo António Tavares Abade - 23919
Rafael Conceição Narciso - 24473

Orientador: Professor Rui Silva

Beja, julho de 2025

Resumo

Neste relatório será abordado o processo de criação de uma solução de comunicações seguras, que permita a troca de mensagens de texto entre os seus utilizadores. Este relatório foi realizado no âmbito da Unidade Curricular de Estágio ou Projeto (IPBeja, 2025).

Keywords: aplicações, cibersegurança, comunicações, criptografia, c#, python, bash, alma-linux, wireguard, sqlite, kotlin

Abstract

In this report, we will address the creation process of a secure communication solution that allows text message exchange between its users. This report was carried out within the scope of the Curricular Unit of Internship or Project (IPBeja, 2025).

Keywords: applications, cybersecurity, communications, cryptography, c#, python, bash, almalinux, wireguard, sqlite, kotlin

Índice

Índice de Figuras

1 Introdução

Para a realização do projeto é necessário desenvolver um sistema de comunicações seguras, que permita a troca de mensagens de texto entre os seus utilizadores. Para tal, é necessário implementar um sistema de autenticação de utilizadores, que permita a criação de contas de utilizador e a autenticação dos mesmos. O sistema deve ser capaz de garantir a confidencialidade, integridade e autenticidade das mensagens trocadas entre os utilizadores. As tecnologias a utilizadas no desenvolvimento deste projeto são: WireGuard (WireGuard, 2025), SQLite (SQLite, 2025), C# (Microsoft, 2025), Kotlin (JetBrains, 2025), Python (Foundation, 2025), Bash (Bash, 2025), AlmaOS (AlmaOS, 2025). Em relação a criptografia, serão utilizados os algoritmos Rijndael (vencedor do AES) e Serpent (segundo lugar do AES), que são algoritmos de criptografia simétrica.

Para este projeto, foi decidido utilizar uma variação da metodologia do *SCRUM*, onde eram realizadas reuniões semanais ao ínicio, e depois foi sendo feito para ocorrer mais tempo entre reuniões, de forma a que fosse possível ter mais tempo para o desenvolvimento das tarefas. Os três membros da equipa foram divididos entre as três áreas do projeto, nomeadamente: desenvolvimento da aplicação Windows, desenvolvimento da aplicação Android e por último, a configuração da infraestrutura da rede.

O cronograma do projeto foi dividido em duas fases principais: Análise de Requisitos, Desenvolvimento, como é possível ver na figura 1. O relatório foi modificado a cada semana, acrescentando novas informações e atualizações sobre o progresso do projeto.

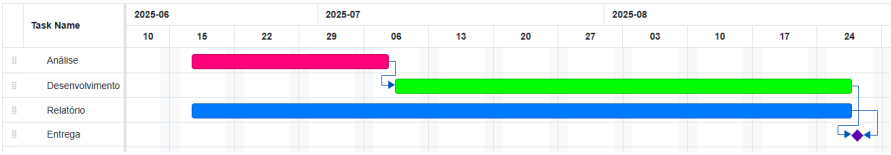


Figura 1: Cronograma do Projeto

Este projeto ficou guardado nos repositórios do GitHub da equipa, onde pode ser consultado por outros alunos e professores, a partir do seguinte link: https://github.com/MartinhoCaeiro/Projeto-Cripto & https://github.com/NarcisoUNK/ProjectFinal_ANDROID.

2 Analise de Requisitos

Para a realização deste projeto, foram necessários os seguintes requisitos:

- **Requisitos Funcionais:**
  - O sistema deve permitir a criação de contas de utilizador.
  - O sistema deve permitir a autenticação de utilizadores.
  - O sistema deve permitir o envio e receção de mensagens entre utilizadores.
  - O sistema deve garantir a confidencialidade, integridade e autenticidade das mensagens trocadas.
- **Requisitos Não Funcionais:**
  - O sistema deve ser seguro e resistente a ataques.
  - O sistema deve ser fácil de usar e intuitivo.
  - O sistema deve ser escalável e capaz de suportar um grande número de utilizadores.

Foram ainda analisadas outras aplicações de troca de mensagens seguras, como o *WhatsApp* (Inc., 2025) e *Telegram* (LLP, 2025), para compreender como estas funcionam. Chegando à conclusão de que utilizam uma encriptação de ponta a ponta para garantir a segurança das mensagens. Neste projeto, será implementada uma abordagem bastante semelhante, porém com a integração da VPN.

3 Tecnologias Utilizadas

Para o desenvolvimento deste projeto, foram utilizadas as seguintes tecnologias:

- **Linguagens de Programação:** C#, Kotlin
- **Frameworks:** .NET, WireGuard
- **Base de Dados:** SQLite
- **Criptografia:** Rijndael (Vencedor AES), Serpent (Segundo lugar AES)
- **Ferramentas de Desenvolvimento:** Git, Visual Studio Code, Android Studio
- **Serviços de Hospedagem:** GitHub

4 Arquitetura do Sistema

O sistema é composto por uma aplicação que funcionará como um chat, onde os utilizadores que estiverem registado na VPN poderão comunicar entre si. Dentro da aplicação, os utilizadores podem saber quem está associado a um endereço IP da VPN, e assim enviar mensagens para esse utilizador. Ao enviar uma mensagem, a aplicação irá contatar o servidor da VPN, para obter o caminho até ao destinatário, e assim enviar a mensagem. Essa mensagem será encriptada antes de ser enviada, para garantir a confidencialidade e integridade da mensagem. Existe a variação da encriptação utilizada na mensagem, de acordo com uma lógica pré-definida na aplicação, sendo que os métodos de encriptação utilizados são o AES e o Serpent.

4.1 Encriptação do Sistema

Sendo assim, a parte de encriptação das mensagens será feita como está demonstrado na figura 2. A mensagem será encriptada consoante um número pseudo-aleatório gerado pela aplicação, sendo assim escolhido o algoritmo de encriptação a utilizar naquele momento, o AES256 ou Serpent. Para encriptar, é necessário fornecer também uma chave de encriptação. Nesta fase, a chave de encriptação é "***Spartacus***". É importante destacar que o seletor está ofuscado, de forma a dificultar a sua identificação, a partir de uma máscara criada através da chave (*salt*) e do número pseudo-aleatório.

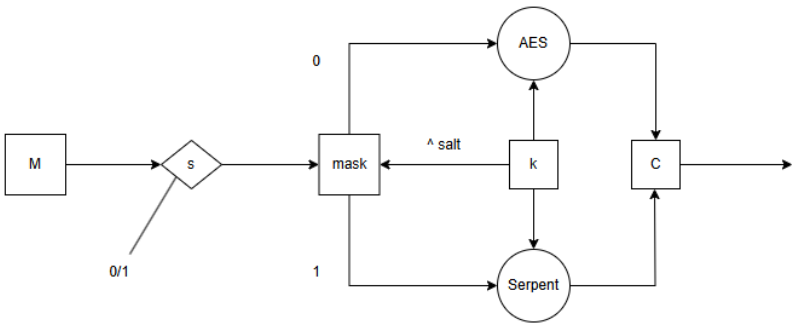


Figura 2: Processo de Encriptação das Mensagens

Para desencriptar, é necessário fazer o processo inverso. Desta forma, o número pseudo-aleatório está anexado à mensagem encriptada, permitindo assim descobrir qual o algoritmo de encriptação utilizado. É possível ver este processo na figura 3.

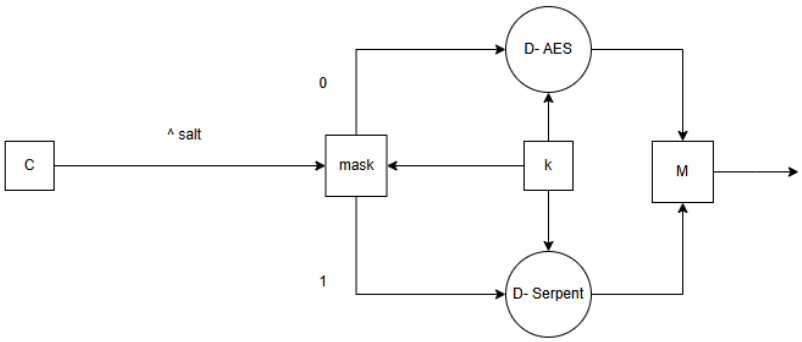


Figura 3: Processo de Decriptação das Mensagens

4.2 Estrutura do Sistema

A estrutura do sistema é focada na troca de informação por dentro da VPN, podendo ser representada na seguinte figura 4:

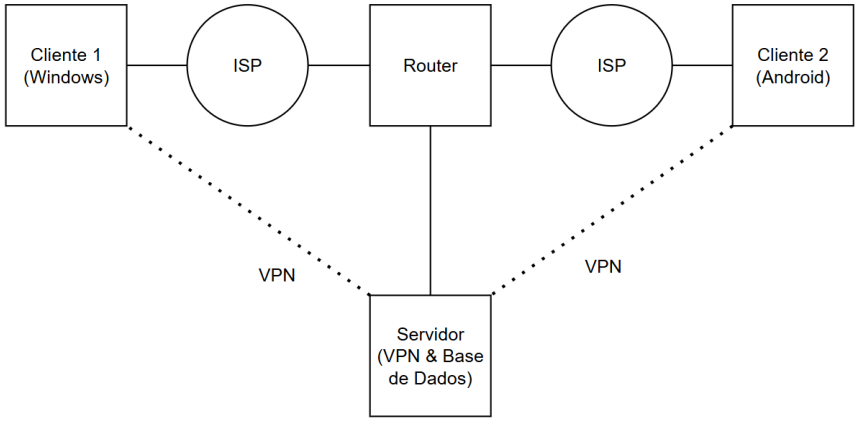


Figura 4: Arquitetura do Sistema

4.3 Implementação da Criptografia (excerto)

O projecto inclui um módulo chamado `DbCrypto` (implementado em Kotlin/Java) responsável por encapsular a encriptação e desencriptação dos conteúdos armazenados. Segue-se um excerto explicativo com os pontos-chave e um trecho do código mais relevante.

Resumo da lógica

- Gera-se um **salt** e um **nonce** aleatórios para cada encriptação.
- Deriva-se uma chave simétrica (256 bits) a partir da passphrase/*masterKey* usando PBKDF2-SHA256 (100k iterações).
- Calcula-se uma máscara (*mask*) para ofuscar qual o algoritmo usado (AES ou Serpent) e guarda-se o valor ofuscado no cabeçalho.
- Usa-se AES-GCM ou Serpent-GCM (via BouncyCastle) com AAD contendo o cabeçalho (magic, versão, algoritmo-ofuscado, iterações, salt e nonce).
- O ficheiro resultante é: cabeçalho || ciphertext || tag.

Trecho de código

```
private fun encryptInternal(data: ByteArray, masterKey: ByteArray, salt:
    ByteArray, nonce: ByteArray, alg: Alg): ByteArray {
    val key = deriveKey(masterKey, salt, ITERATIONS)
    val mask = computeAlgMask(masterKey, salt)
    val algXor = (alg.code.toInt() xor (mask.toInt() and 0xFF)).toByte()
    val aad = buildHeader(algXor, ITERATIONS, salt, nonce) // full header is
        AAD
    val ctWithTag = if (alg == Alg.AES) {
        val cipher = Cipher.getInstance("AES/GCM/NoPadding")
        cipher.init(Cipher.ENCRYPT_MODE, SecretKeySpec(key, "AES"),
            GCMParameterSpec(TAG_SIZE * 8, nonce))
        cipher.updateAAD(aad)
        cipher.doFinal(data)
    } else {
```

```
12          val gcm = GCMBlockCipher(SerpentEngine())
13          gcm.init(true, AEADParameters(KeyParameter(key), TAG_SIZE * 8, nonce,
                aad))
14          val out = ByteArray(gcm.getOutputSize(data.size))
15          var len = gcm.processBytes(data, 0, data.size, out, 0)
16          len += gcm.doFinal(out, len)
17          out.copyOf(len)
18      }
19      return envelope(algXor, salt, nonce, ctWithTag)
20  }
```

Listing 1: Excerto de `DbCrypto` — função de encriptação

Preview / Exemplo de validação Para validar o comportamento foi utilizada a função de desencriptação que verifica o cabeçalho, desfaz a máscara do algoritmo, re-deriva a chave com os mesmos parâmetros e tenta a desencriptação; a produção correcta deverá retornar o conteúdo SQLite original (verifica-se o "magic"do SQLite).

5 Módulos do Sistema

5.1 Interface do Sistema

A interface para computador será desenvolvida em C#, cuja qual é uma linguagem rápida e fácil de aprender, e que permite o desenvolvimento de aplicações desktop de forma rápida e eficiente, enquanto que a interface para Android será desenvolvida em Kotlin, que é uma linguagem de programação moderna e concisa, que permite o desenvolvimento de aplicações Android de forma rápida e eficiente. Esta interface é composta por 4 ecrãs principais:

- **Ecrã de Login:** onde o utilizador pode autenticar-se na aplicação utilizando as suas credenciais, tambem é possivel ver o status da ligação WireGuard.
- **Ecrã de Lista de Chats:** onde o utilizador pode ver a lista de chats existentes, bem como criar novos chats.

- **Ecrã de Chat:** onde o utilizador pode enviar e receber mensagens de outros utilizadores.
- **Ecrã de Configuração WireGuard:** onde o utilizador pode criar um ficheiro para configuração do WireGuard.

5.2 Módulo da VPN

A VPN estará alojada num servidor com o sistema operativo AlmaOS 8.10 (AlmaOS, 2025), e será responsável por gerir as ligações dos utilizadores à VPN, bem como a autenticação dos mesmos. O serviço escolhido foi o WireGuard, que é um serviço de VPN de código aberto, leve e de alto desempenho, que utiliza criptografia moderna para garantir a segurança das comunicações. É necessário configurar previamente o WireGuard no dispositivo do utilizador, para que este possa estabelecer uma ligação à VPN.

5.3 Base de Dados

A base de dados será utilizada para armazenar as informações dos utilizadores, como as suas credenciais, bem como as mensagens trocadas entre os utilizadores, esta será implementada utilizando o SQLite. A base de dados irá ter a seguinte estrutura:

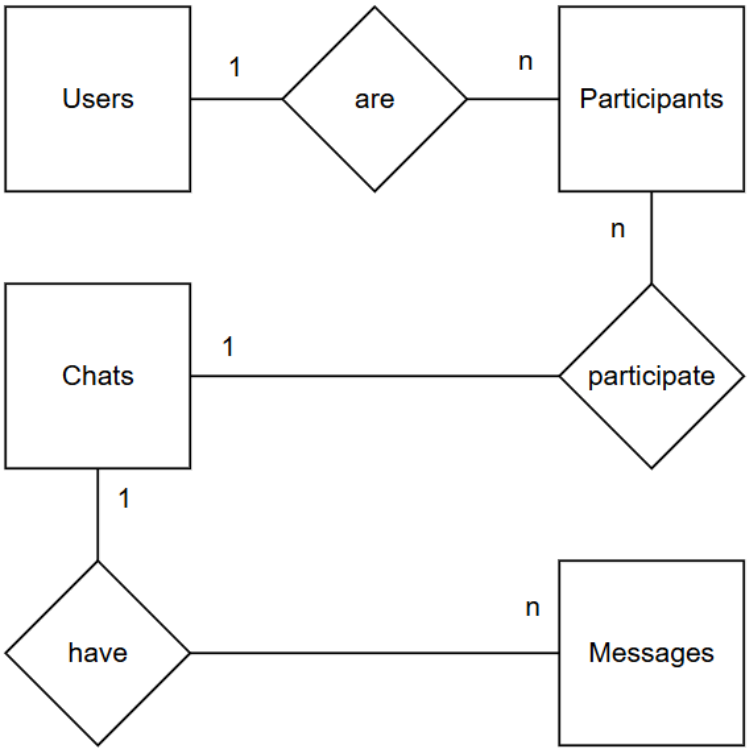
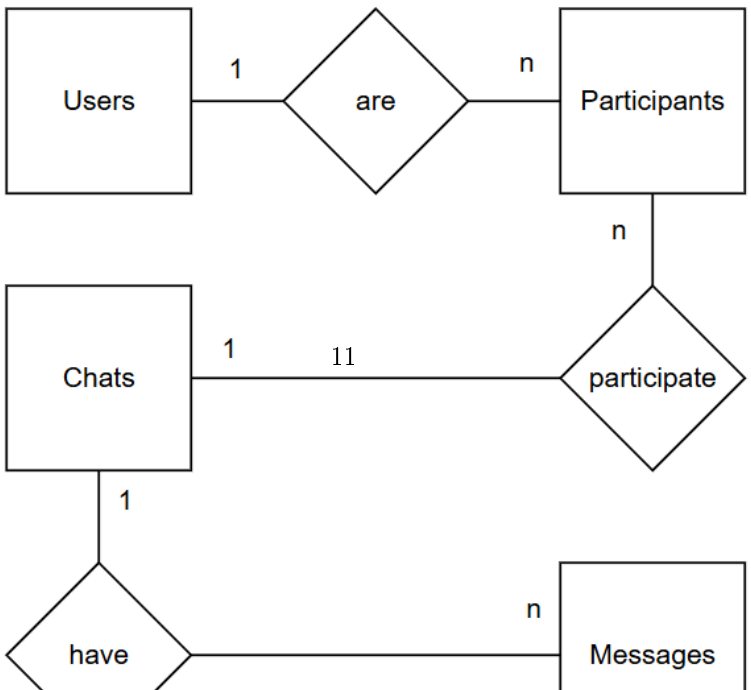


Figura 5: Estrutura da Base de Dados

Sendo que a base de dados é composta por 4 tabelas, cada uma com as suas respetivas colunas, sendo que as tabelas são:

- **User:**
  - UserID (chave primária)
  - Username (string)
  - Password (string)
- **Chat:**
  - ChatID (chave primária)
  - Name (string)
  - AdminID (chave estrangeira, referência à tabela User)
- **Participant:**
  - ParticipantID (chave primária)
  - ChatID (chave estrangeira, referência à tabela Chat)
  - UserID (chave estrangeira, referência à tabela User)
- **Message:**
  - MessageID (chave primária)
  - ParticipantID (chave estrangeira, referência à tabela Participant)
  - Content (string)
  - Date (data e hora da mensagem)
  - SenderUserID (chave estrangeira, referência à tabela Participant)



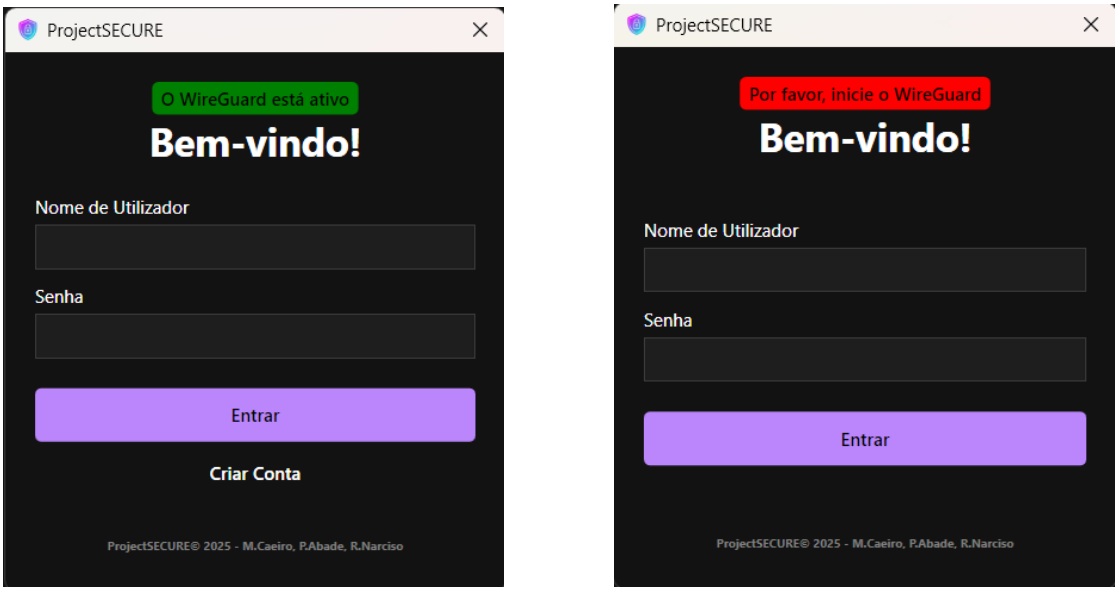
6 Desenvolvimento

6.1 Interface para Computador

Para o desenvolvimento da interface para computador, foi utilizado o Visual Studio Code, que é um ambiente de desenvolvimento integrado (IDE) da Microsoft.

6.1.1 Ecrã de Login

O ecrã de login é composto por um formulário onde o utilizador pode inserir as suas credenciais, e um botão para autenticar-se na aplicação. Caso não tenha uma conta criada apenas tem que preencher o formulario e clicar no botão de criar conta, esta ação só é possivel com o Wireguard ativado. Também é possivel ver o estado da ligação WireGuard, caso esteja ligado ou desligado.

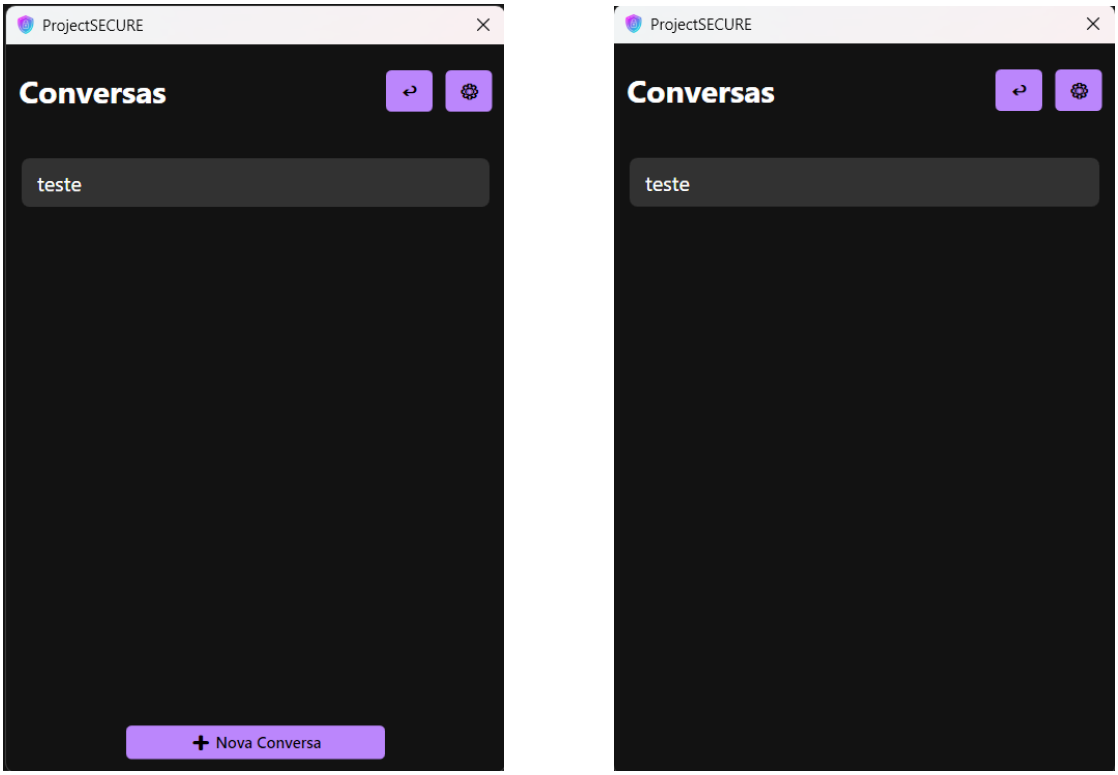


(a) WireGuard Ativado (b) Wireguard Desativado

Figura 6: Ecrã de Login - Computador

6.1.2 Ecrã de Lista de Chats

O ecrã de lista de chats é composto por uma lista de chats existentes e um botão para criar novos chats, este só pode ser criado com o WireGuard ativado. No seu canto superior direito, existe um botão para aceder às definições da aplicação, onde o utilizador pode configurar se quer a aplicação em modo escuro ou claro, e no seu canto inferior esquerdo existe um botão para terminar sessão caso deseja mudar de utilizador.

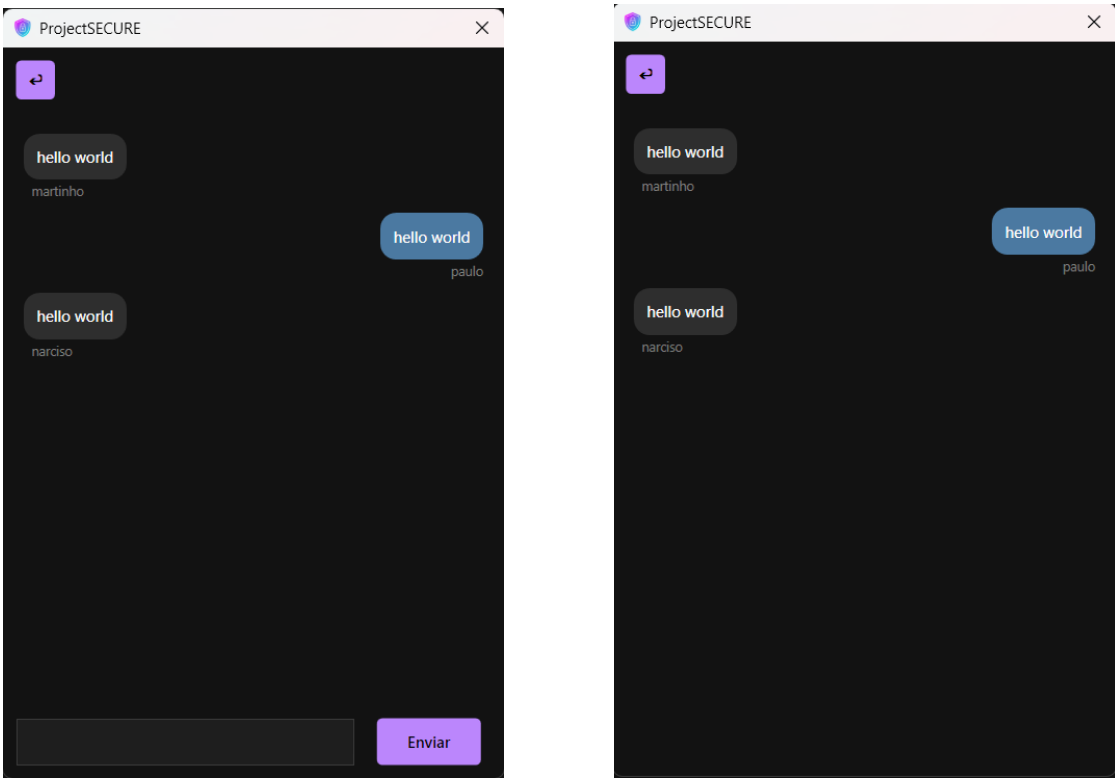


(a) WireGuard Ativado (b) Wireguard Desativado

Figura 7: Ecrã de Lista de Chats - Computador

6.1.3 Ecrã de Chat

O ecrã de chat é composto por uma lista de mensagens trocadas entre os utilizadores, mensagens recebidas ficam do lado esquerdo e mensagens enviadas no lado direito. Apenas é possivel enviar mensagens com o WireGuard ativado, para enviar uma mensagem basta escrever no campo de texto e clicar no botão de enviar.



(a) WireGuard Ativado (b) Wireguard Desativado

Figura 8: Ecrã de Chat - Computador

6.1.4 Ecrã de Configuração WireGuard

O ecrã de configuração WireGuard é composto por um formulário onde o utilizador insere o seu nome, e um botão para descarregar o ficheiro de configuração do WireGuard.

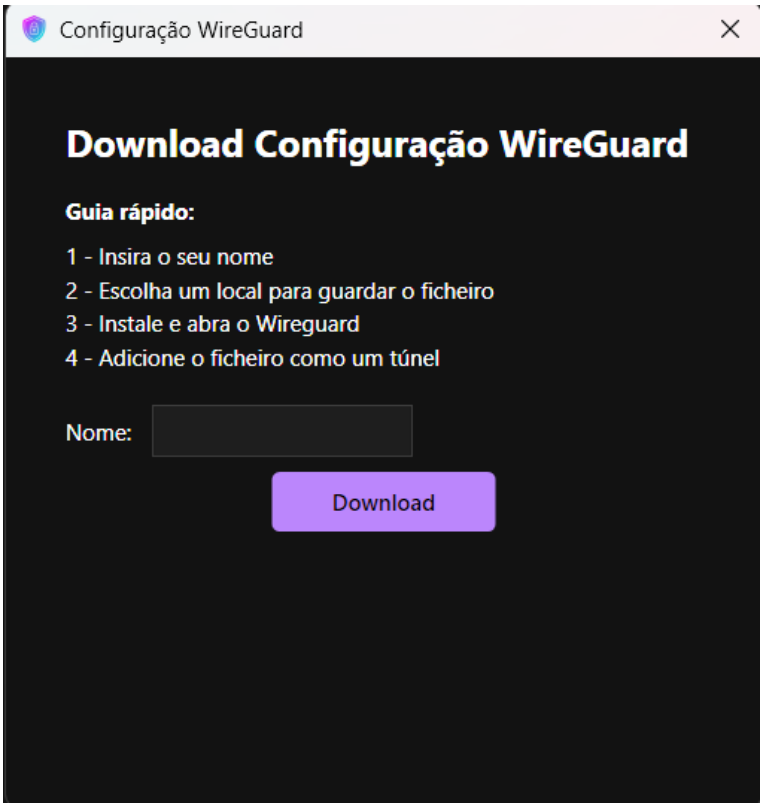


Figura 9: Ecrã de Configuração WireGuard - Computador

6.2 Interface para Android

Para o desenvolvimento da interface para Android, foi utilizado o Android Studio, que é um ambiente de desenvolvimento integrado (IDE) oficial para o sistema operativo Android.

6.2.1 Ecrã de Login

O ecrã de login é composto por um formulário onde o utilizador pode inserir as suas credenciais, e um botão para autenticar-se na aplicação. Este pode iniciar sessão ou criar uma conta, caso não possua uma. Também é possivel ver o estado da ligação WireGuard, caso esteja ligado ou desligado. Existe ainda a possibilidade de estabelecer a comunicação de uma forma mais prática para o utilizador, através do download do ficheiro de configuração do WireGuard.

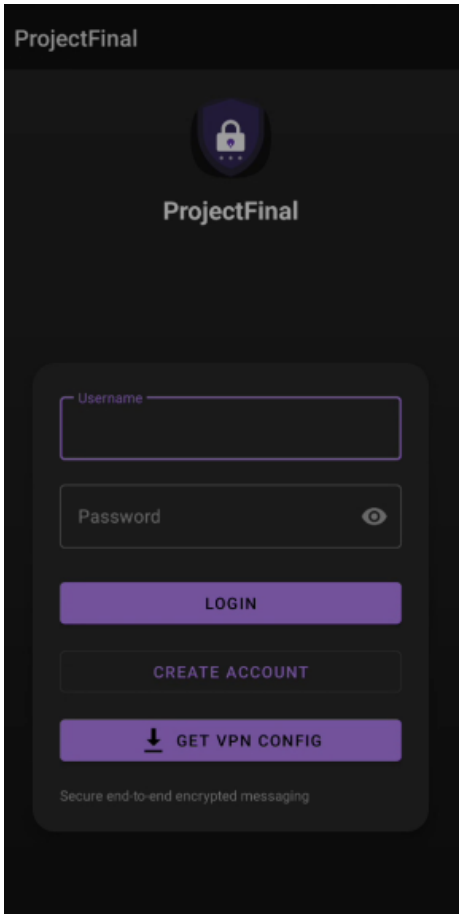


Figura 10: Ecrã de Login - Android

6.2.2 Ecrã de configuração do WireGuard

Neste ecrã, o utilizador pode inserir um nome que está associado à sua ligação WireGuard. Após inserir e enviar o pedido, será fornecido o ficheiro de configuração do WireGuard, que pode ser importado diretamente na aplicação do WireGuard no dispositivo Android.

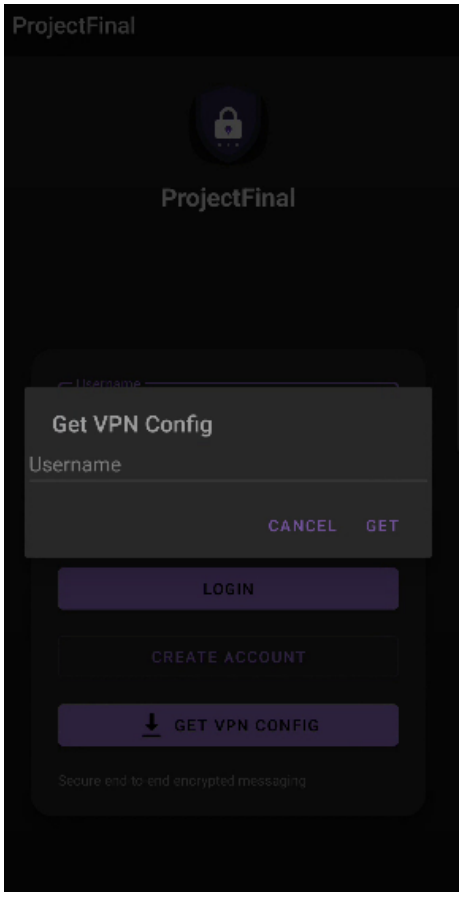


Figura 11: Ecrã de Configuração do WireGuard - Android

6.2.3 Ecrã de Lista de Chats

O ecrã de lista de chats é composto por uma lista de conversas anteriores, onde o utilizador pode selecionar uma conversa para visualizar o seu conteúdo. Neste ecrã, também é possível iniciar uma nova conversa, caso o utilizador deseje.

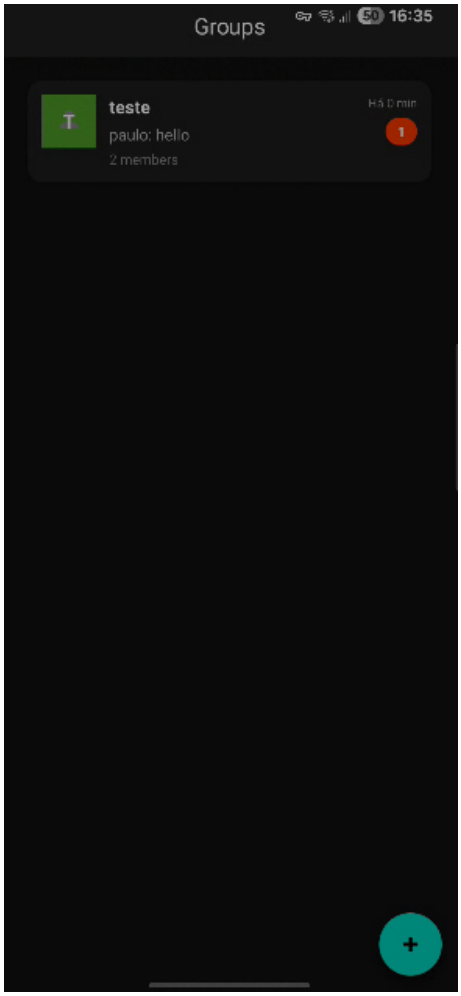


Figura 12: Lista de Chats - Android

6.2.4 Ecrã de Chat

O ecrã de chat é composto por uma lista de mensagens trocadas entre os utilizadores, mensagens recebidas ficam do lado esquerdo e mensagens enviadas no lado direito. Apenas é possivel enviar mensagens com o WireGuard ativado, para enviar uma mensagem basta escrever no campo de texto e clicar no botão de enviar.

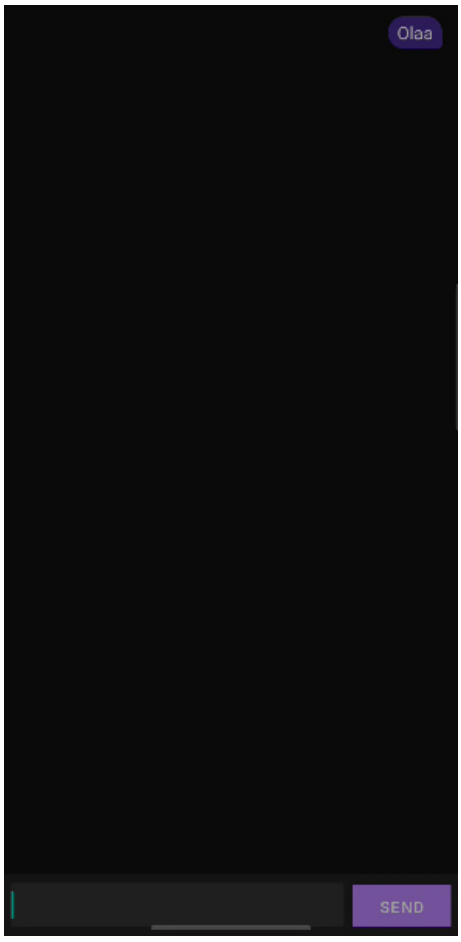


Figura 13: Ecrã de Chat - Android

6.3 AlmaOS - Serviço de VPN

Para configurar corretamente o serviço de VPN, é necessário instalar o WireGuard no servidor AlmaOS 8.10, para isso ser feito, foi necessário seguir os seguintes passos:

1. Adicionar o repositório EPEL - sudo dnf install epel-release
2. Adicionar o repositório ELREPO - sudo dnf install https://www.elrepo.org/elrepo-release-8.el8.elrepo.noarch.rpm
3. Ativar o CodeReady Builder - sudo /usr/bin/crb enable
4. Atualizar os metadados - sudo dnf makecache
5. sudo dnf –enablerepo=elrepo install kmod-wireguard -y
6. Instalar o WireGuardTools - sudo dnf install wireguard-tools -y

Para verificar se o WireGuard está instalado corretamente, pode-se utilizar o comando **"sudo modprobe wireguard"**, se este não devolver nenhum output, significa que o WireGuard está instalado corretamente.

Agora, para configurar o WireGuard, é necessário criar as chaves de criptografia, para isso é necessário ir para a pasta **/etc/wireguard** e executar o seguintes comando:

```
wg genkey | tee server_private.key | wg pubkey > server_public.key
chmod 600 server_private.key
```

Isto irá gerar duas chaves, uma privada e uma pública, que serão utilizadas para autenticar os utilizadores na VPN.

Após isso, é necessário criar o ficheiro de configuração do WireGuard, onde estará definida a configuração da VPN em si, como o endereço IP da VPN, a porta de escuta, as chaves de criptografia, entre outros. Para isso, é necessário criar o ficheiro **wg0.conf** na pasta /**etc**/**wireguard**, e adicionar o seguinte conteúdo:

```
[Interface]
PrivateKey = <Chave Privada do Servidor>
Address = 10.0.0.1/24
ListenPort = 51820
SaveConfig = true
[Peer]
PublicKey = <Chave Pública do Cliente>
AllowedIPs = 10.0.0.2/32
```

Para garantir o bom funcionamento da VPN, é necessário ativar o encaminhamento de endereços IP, para assim permitir o tráfego de rede. Para isso, é necessário fazer o seguinte comando:

```
# Para permitir o tráfego de rede
echo "net.ipv4.ip_forward = 1" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p

# Para permitir o tráfego pela porta 51820
sudo firewall-cmd --add-masquerade --permanent
sudo firewall-cmd --add-port=51820/udp --permanent
sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o enp0s3 -j MASQUERADE
sudo firewall-cmd --reload
```

Para ser possível encaminhar o tráfego de rede, é necessário adicionar uma regra ao Router para que, todos os pacotes que cheguem à porta 51820 sejam encaminhados para o servidor WireGuard. Por fim, basta iniciar o serviço do WireGuard, para isso é necessário executar o seguinte comando:

```
sudo systemctl enable --now wg-quick@wg0
```

Para conseguir estabelecer a troca de ficheiros entre o servidor e o cliente, foi necessário implementar um *script* que gerencia as trocas de ficheiros da base de dados entre os clientes e o servidor. Esse script foi feito em Python, utiliza a biblioteca Flask para criar uma API REST que permite a comunicação entre o cliente e o servidor. Foi necessário permitir a porta 8000 no servidor, para que o cliente possa comunicar com o servidor.

Do lado do cada cliente, vai ser necessário configurar o WireGuard, de maneira a que apenas a informação proveniente da aplicação desenvolvida seja enviada através da VPN, onde assim serão poupados recursos do dispositivo que está a fazer o papel de servidor. Para isso, foi criado um *script* que cria o ficheiro de configuração do WireGuard para o cliente, sendo que este só deve adicionar o ficheiro de configuração na aplicação do WireGuard. O *script* ainda automatiza a adição do cliente nas configurações do servidor. Foi criado ainda um *script* complementar, em Python, para ser obtido o ficheiro de configurações de maneira mais fácil para o utilizador/cliente. Este script segue o mesmo princípio da troca da base de dados, com a diferença de que está a utilizar a seu endereço de IP local em vez da VPN. Sendo assim, para obter o ficheiro de configurações, é necessário estabelecer conexão com o servidor no endereço *192.168.1.106:9595 nome*, onde *nome* é o nome do utilizador que quer aceder à VPN. Ao estabelecer ligação, vai ter um ficheiro de configurações criado por um administrador à espera de ser transferido. O administrador cria esse ficheiro com o auxilio do *script* anterior, para conseguir limitar o acesso às comunicações caso o acesso à VPN seja comprometido.

```
sudo firewall-cmd --add-port=8000/tcp --permanent
sudo firewall-cmd --reload
```

7 Problemas Encontrados

Durante o desenvolvimento do projeto, foram encontrados alguns problemas, que foram resolvidos da seguinte forma:

7.1 Problemas de Conexão

Logo no inicio da implementação da VPN foi necessário achar uma solução para a configuração de encaminhamento de pacotes, pois o servidor Linux não possui um endereço IP público próprio. Para contornar este problema, foi necessário configurar o encaminhamento de pacotes no router, de forma a que todos os pacotes que chegassem à porta 51820 fossem encaminhados para o servidor Linux, e este resolveria a situação encaminhando o resto dos pacotes pela VPN. Outro problema encontrado foi a dificuldade em encontrar uma maneira para estabelecer a conexão pela primeira vez, ou seja, enviar o ficheiro de configuração da VPN para o dispositivo do utilizador, de forma a que este pudesse estabelecer a conexão com o servidor. Para resolver este problema, foi necessário criar um *script* que gerasse o ficheiro de configuração do WireGuard, e que fosse possível enviar esse ficheiro para o dispositivo do utilizador.

7.2 Problemas na aplicação Windows/Android

Durante o desenvolvimento da aplicação, foram encontrados diversos problemas, como por exemplo:

- Problemas de compatibilidade entre tipos de base de dados (SQLite e Schemas);
  - Incompatibilidade de nomes entre tabelas, sendo necessário colocar as tabelas em letra minúscula em ambos;
  - Configurações por defeito diferentes entre as plataformas, sendo necessário deixar explícito em ambos os lados sobre as configurações necessárias, como por exemplo "$NOT\ NULL$";
- Permissões do Android, sendo necessário adicionar todas as permissões necessárias no AndroidManifest.xml para funcionar corretamente;

- Visualização/sincronização incorreta das informações e a impossibilidade de modificar a base de dados enquanto a aplicação está a correr, sendo necessário criar bases de dados temporárias para permitir a edição dos dados.

7.3 Depuração da camada criptográfica e protocolo de *framing*

Contexto. A arquitetura Cliente Android (Kotlin) ↔ Servidor PC (Python) exige confidencialidade e integridade ponta-a-ponta. Foi adotado Serpent-256 em modo AEAD, com AAD a autenticar metadados. Em testes iniciais surgiram falhas consistentes de decriptação (`BAD_DECRYPT`).

Sintomas observados.

- `BAD_DECRYPT` repetido antes de qualquer *plaintext* ser obtido.
- Comprimento de chave correto (32 bytes), AAD com 38 bytes constantes mas não padronizados.
- Falha idêntica entre várias "abordagens de compatibilidade", sugerindo desalinhamento estrutural (nonce/tag/AAD) e não corrupção de dados.
- Envio rápido de duas mensagens causava paragens no receptor ("silêncio") ou exceções intermitentes.

Metodologia de depuração.

a) Instrumentação com *logs* DEBUG antes/depois de: derivação de chave, montagem de AAD, geração/uso de nonce, chamada AEAD.

b) *Hexdump*/*base64* lado-a-lado (cliente/servidor) de AAD, nonce, *ciphertext*, tag; assertivas de tamanho de segmentos.

c) Ensaios com *scripts* Python isolando a decriptação e testes negativos (alteração de 1 byte para confirmar assinatura de falha).

d) Validações com ferramentas auxiliares (`openssl dgst`, `hexdump -C`).

Hipóteses priorizadas.

- Divergência de modo AEAD (GCM/EAX) ou parâmetros.
- Nonce/IV com tamanho/ordem diferentes (p. ex., 12 vs 16 bytes) ou reutilização acidental.
- Tag truncada (12 vs 16 bytes) na serialização.
- Construção de AAD assimétrica (ordem, endianidade, inclusão/omissão de campos).
- Parâmetros de KDF distintos (*salt*/*info*).
- Fragilidade no *framing*: concatenação de mensagens sem delimitador robusto e leituras parciais do *socket*.

AEAD, GCM e EAX AEAD (Authenticated Encryption with Associated Data) combina encriptação e autenticação numa única operação: cifra os dados para que ninguém os leia e, ao mesmo tempo, gera um selo (a chamada *tag*) que comprova que os dados e certos metadados não foram alterados. Pensa nisto como enviar um envelope lacrado e com um selo de segurança — se o selo estiver danificado ou diferente, o recetor percebe que houve manipulação e rejeita a mensagem.

Dois modos comuns para obter AEAD são GCM e EAX. Ambos fazem essencialmente o mesmo: cifram os dados e produzem uma *tag* que o recetor verifica. As diferenças técnicas interessam aos implementadores, mas para entender o problema prático convém saber o seguinte:

- **GCM (Galois/Counter Mode)**: é muito utilizado e rápido. Normalmente espera um *nonce* de 12 bytes e produz uma *tag* de 16 bytes. Um requisito crítico do GCM é a unicidade do *nonce* para cada mensagem quando a mesma chave é usada — reutilizar um *nonce* pode quebrar a segurança.
- **EAX**: é outra construção AEAD que também cifra e gera uma tag, mas trata nonces e metadados de forma ligeiramente diferente. Funciona bem, mas não é compatível com GCM: se um lado usar GCM e o outro EAX, as tags não baterão e a decriptação falhará.

Em termos práticos, isto significa que o emissor e o recetor têm de concordar exatamente em: qual o modo (GCM ou EAX), o comprimento do *nonce*, o comprimento da *tag* e como se

formam os metadados autenticados. Qualquer discrepância — modo diferente, nonce duplicado, tag truncada ou AAD com conteúdo diferente — faz com que a verificação falhe e a mensagem seja rejeitada (por exemplo com `BAD_DECRYPT`).

AAD O AAD (Associated Authenticated Data) são informações que não são cifradas mas entram na criação do selo (tag). Exemplos: versão do protocolo, número de sequência, identificador de utilizador. O AAD protege estes campos contra alterações, mas exige que emissor e recetor tenham exactamente os mesmos bytes em AAD; qualquer diferença causa falha na verificação.

KDF Um KDF (Key Derivation Function) transforma uma chave mestra ou palavra-passe num conjunto de chaves de uso (por exemplo: chave de encriptação, prefixo de nonce). Parâmetros como *salt*, número de iterações e informação adicional (*info*) têm de ser os mesmos nos dois lados; se não, as chaves diferirão e a decriptação não funcionará.

Framing TCP é um fluxo contínuo de bytes: pacotes enviados separadamente pelo emissor podem chegar agrupados ou fragmentados. **Framing** é o acordo que define onde começa e acaba cada mensagem (por exemplo usar um prefixo de comprimento de 4 bytes). Sem framing robusto, o recetor pode tentar decriptar dados incompletos ou várias mensagens coladas, levando a erros de autenticação (`BAD_DECRYPT`) e problemas subsequentes no processamento.

extbfNormalização do envelope e *framing* (explicação detalhada) Para remover ambiguidade em troca de mensagens sobre TCP definimos um contrato binário claro e passos de validação obrigatórios no receptor. O envelope tem sempre um *length-prefix* de 4 bytes (**big-endian**) que indica o comprimento total do restante *frame* e permite ao recetor saber quando tem um *frame* completo.

Formato canónico (campo : tamanho em bytes — descrito por ordem de leitura):

- **LEN (4)**: comprimento (em bytes) de tudo o que vem a seguir no *frame*.
- **MAGIC (2)**: identificador do protocolo (p. ex. 0x50 0x53 para "P S").
- **VER (1)**: versão do envelope/protocolo.
- **ALG (1)**: código do algoritmo (ex.: 1=AES, 2=SERPENT) — armazenado ofuscado no cabeçalho quando aplicável.

- **FLAGS (1)**: flags de uso futuro (compactação, fragmentação, indicadores de diagnóstico).
- **SEQ (8)**: contador monotónico de mensagem (big-endian) — usado no AAD e na derivação do *nonce*.
- **TIMESTAMP (8)**: marca temporal para diagnóstico (opcional na verificação temporal).
- **NONCE (12)**: nonce/IV usado pelo AEAD (a nossa prática: 4 bytes prefixo derivado da chave + 8 bytes de SEQ, total 12 bytes).
- **TAG (16)**: etiqueta de autenticação AEAD (16 bytes, sem truncagem).
- **AAD_LEN (2)**: comprimento em bytes do campo AAD.
- **CT_LEN (4)**: comprimento em bytes do *ciphertext*.
- **AAD**: campo de metadados autenticados (tamanho AAD_LEN).
- **CIPHERTEXT**: dados cifrados (tamanho CT_LEN) imediatamente seguidos do **TAG** já presente acima.

Notas práticas e motivo das escolhas:

- Usar **LEN(4)** evita ambiguidade em streams TCP — o receptor acumula até LEN bytes e só depois processa o *frame* completo.
- Colocar **SEQ** no AAD e usá-lo para formar o **NONCE** (por exemplo: NONCE = SHA256(key)`[:4]` || SEQ) garante unicidade por mensagem e rastreabilidade.
- Reservar um campo **TAG** de 16 bytes e não a truncar reduz riscos de falsificação; qualquer truncagem deve ser documentada e deprecada.
- Documentar explicitamente a ordem e a endianidade de cada campo evita erros por diferenças de implementação.

Regras de segurança e robustez (obrigatórias):

- Validar limites máximos (p. ex. AAD_LEN < 4096, CT_LEN $< 10^7$) antes de alocar buffers para evitar DoS por mensagens gigantes.

- Fazer comparações da TAG em tempo constante; limpar buffers e chaves sensíveis após uso.
- Nunca assumir que um único `read()` do socket corresponde a exactamente um *frame* — por isso o BufferAssembler existe.
- Em caso de `AEAD_FAIL` registar um dump hex do AAD/nonce/tag apenas em modo debug e com limitação de taxa para não vazar segredos em logs de produção.

Implementações e salvaguardas recomendadas

- **BufferAssembler no receptor**: componente que acumula bytes lidos do socket, lê LEN quando possível e só entrega frames completos ao pipeline de decifração.
- **Sequência monotónica (SEQ)**: incluir SEQ no AAD e na derivação do NONCE; rejeitar mensagens com SEQ duplicado ou regressivo (janela aceitável para reenvios controlada).
- **Self-test no arranque**: cifra/decifra um vetor conhecido para garantir compatibilidade do AEAD e dos parâmetros; abortar se o self-test falhar.
- **Backpressure e ACKs**: durante diagnóstico usar ACKs/pausas para evitar ráfagas; em produção garantir que o pipeline aplica limites e filas para escrita na BD.
- **Logs estruturados**: registar contadores de `AEAD_FAIL` e tipos de erro; dumps binários só em debug com rotação e retenção curta.

Bug observado: "duas mensagens consecutivas"— explicação técnica curta

- Causa típica: leituras do socket que retornam parte de um frame seguido de parte do frame seguinte, e um receptor que tenta decifrar imediatamente sem acumular o frame completo; o resultado é que campos deslocam-se (nonce/tag/AAD lidos do lugar errado) e a verificação AEAD falha.
- Mitigação: implementar o BufferAssembler + length-prefix e processar frames apenas quando completos; serializar gravações na BD (worker/fila) e usar transacções atómicas para evitar corrupção em caso de mensagens inválidas.

Exemplo prático: offsets e pseudocódigo de parsing Para tornar o contrato mais palpável, apresentamos um mapeamento de offsets (em bytes) do início do *frame* e um pseudocódigo simples que demonstra como o *BufferAssembler* deve operar.

Exemplo de offsets (início do frame = byte 0):

```
0..3   LEN (4 bytes, big-endian)
4..5   MAGIC (2 bytes)
6      VER (1 byte)
7      ALG (1 byte)
8      FLAGS (1 byte)
9..16  SEQ (8 bytes)
17..24 TIMESTAMP (8 bytes)
25..36 NONCE (12 bytes)
37..52 TAG (16 bytes)
53..54 AAD_LEN (2 bytes)
55..58 CT_LEN (4 bytes)
59..(58+AAD_LEN)    AAD
...      CIPHERTEXT (CT_LEN bytes)
```

Pseudocódigo simplificado (lógica do receptor):

```
buffer = empty
while socket open:
    buffer += socket.read()
    if len(buffer) < 4: continue  # ainda não temos LEN
    LEN = be32(buffer[0:4])
    if len(buffer) < 4 + LEN: continue  # aguardar frame completo
    frame = buffer[4:4+LEN]
    buffer = buffer[4+LEN:]  # remover frame consumido
    # parse campos fixos do frame (MAGIC, VER, ALG, FLAGS, SEQ, TIMESTAMP, NONCE, TAG
    if MAGIC != EXPECTED: log('bad magic'); continue
    if AAD_LEN > MAX_AAD or CT_LEN > MAX_CT: log('oversize'); continue
```

```
aad = frame[offset_aad:offset_aad+AAD_LEN]
ct  = frame[offset_ct:offset_ct+CT_LEN]
if not aead_decrypt(ct, aad, nonce, key):
    incr_counter('AEAD_FAIL')
    log_debug_hex(aad, nonce, tag)
    continue
deliver_payload(decrypted)
ack_if_required()
```

Comentário final: este exemplo mostra por que motivo é crítico não tentar decifrar até o *frame* completo estar disponível e por que a ordem/exatidão dos campos (nonce, AAD, tag) é determinante para a verificação AEAD.

Caso observado: logs e impacto prático Durante a instrumentação foi detectado um padrão de erro quando o servidor tenta decifrar pacotes com Serpent: mensagens como "`Serpent compatibility decryption failed: All Serpent compatibility approaches failed`" seguidas de `BAD_DECRYPT` aparecem no receptor (imagem de debug anexada). Estes sinais correspondem à falha da verificação AEAD — tipicamente causada por divergência nos campos autenticados (AAD), no *nonce* ou por leitura/parsing incorreto do envelope.

Reprodução e efeito sobre a base de dados Verificámos um cenário replicável que provoca corrupção/queda da base de dados no PC quando o cliente Android envia duas mensagens muito próximas em tempo. Exemplo reproduzível:

1. PC: OLA
2. ANDROID: SUP1
3. ANDROID: SUP2

Ao enviar assim (duas mensagens consecutivas do Android) o receptor recebe *frames* concatenados ou incompletos, tenta decifrá-los e falha (`BAD_DECRYPT`), o que, nas implementações observadas, pode conduzir a escrita inválida na base de dados ou a estados em que a BD deixa de responder.

Diagnóstico e medidas imediatas Observações feitas durante a depuração:

- Comparação lado-a-lado (hex) de AAD/nonce/tag mostra diferenças entre emissor e recetor quando ocorre a falha.
- Reenvio isolado do mesmo envelope normalmente decifra correctamente, pelo que o problema não parece ser corrupção permanente dos bytes.
- Condições de corrida no processamento e parsing de *streams* TCP são a causa provável.

Medidas imediatas aplicadas:

- Forçar processamento sequencial no receptor com um *BufferAssembler* que reconstitui frames completos antes de qualquer operação criptográfica.
- Introduzir ACK/backpressure em fases de diagnóstico para evitar ráfagas de mensagens concorrentes.
- Garantir que cada escrita na BD ocorre dentro de transacções atómicas para evitar corrupção parcial.

Recomendações para correção definitiva

- Implementar o *length-prefix* (4 bytes big-endian) e usar um *BufferAssembler* robusto no receptor para evitar decifrações de dados incompletos.
- Serializar o acesso à base de dados através de um worker dedicado ou fila de escrita e usar transacções/WAL para tolerância a ráfagas.
- Validar comprimentos de AAD/nonce/tag antes de chamar a decriptação; registar dumps hex apenas em modo debug e com limitação de taxa.
- Assegurar unicidade de nonce por mensagem (por exemplo: NONCE = prefixo chave || SEQ monotónico) e separar responsabilidades de KDF para evitar reutilização indevida de material criptográfico.

Resultados e validação.

- Eliminação do "silêncio" após ráfagas curtas e ausência de `BAD_DECRYPT` nas execuções instrumentadas.
- *Stress tests* internos com pares de mensagens (intervalos 30–250 ms) sem *crash*; *logs* confirmam **SEQ** crescente e *frames* completos.
- Critérios de fecho definidos: 0 ocorrências `BAD_DECRYPT` em campanhas prolongadas; 100 envios duplos consecutivos sem *crash*; latência média < 150 ms; **SEQ** sem desvios não explicados.

Lista de verificação (antes de nova tentativa).

- Mesmo algoritmo e modo AEAD nos dois lados; **TAG=16B**, **NONCE=12B** único.
- **KDF** com *salt*/*info* idênticos; AAD determinística e documentada.
- **Framing** com **LEN(4)** ativo; tratamento de exceções não encerra o processo.
- **Self-test** no arranque aprovado.

Lições aprendidas. A principal fonte de instabilidade não era apenas a cifra, mas a ausência de um contrato binário formal (layout de envelope + AAD + *framing*). Ao fixar cada campo e a sua semântica, reduziram-se incompatibilidades e simplificou-se a depuração. A unicidade do nonce derivado de **SEQ** e a instrumentação detalhada foram determinantes para estabilizar o sistema.

Fundamentação teórica (AEAD, AAD, nonces e framing).

- **AEAD** (*Authenticated Encryption with Associated Data*) combina confidencialidade e integridade: a decriptação falha integralmente se qualquer bit do *ciphertext*, da **TAG** ou da **AAD** estiver alterado. Isto previne ataques de manipulação e "corta e cola".
- **AAD** é não cifrada mas autenticada; deve incluir metadados que definem o contexto (p. ex., versão, algoritmo, **SEQ**, **USER_ID**). Divergências entre emissor e recetor provocam `BAD_DECRYPT` por falha na verificação da TAG.
- O tamanho da **TAG** (tipicamente 16 bytes) é um compromisso entre segurança e espaço. Truncagens elevam o risco de falsificações bem-sucedidas.

- **Nonces/IVs** em modos como GCM/EAX devem ser *únicos por chave*. Reutilização de nonce compromete a confidencialidade e pode afetar a integridade.

Modos AEAD e requisitos de nonce.

- **GCM**: desempenho elevado; nonce recomendado de 96 bits (12 bytes) e unicidade estrita por chave.
- **EAX/OCB**: alternativas com requisitos semelhantes de unicidade. OCB teve restrições de patente históricas.
- **SIV** (*Synthetic IV*): tolera reutilização de nonce, adequado quando a unicidade não é garantida, à custa de desempenho.
- **Prática adotada**: NONCE = prefixo derivado da chave || **SEQ** (12 bytes), garantindo unicidade se **SEQ** for monotónico por sessão.

Derivação e separação de chaves (KDF).

- Utilize **HKDF** com *salt* e *info* para derivar chaves de sessão a partir de uma raiz, aplicando *domain separation* para *encrypt*, *mac*, *nonce-prefix*, etc.
- Evite reutilizar a mesma chave para múltiplos propósitos; separe por função para reduzir o acoplamento e facilitar auditoria.
- Considere rotação por tempo ou contagem; registe **VER**/**ALG** no envelope para agilidade criptográfica.

Framing em canais de *byte-stream* (TCP).

- TCP pode fragmentar ou coalescer envios; o recetor deve reconstituir *frames* completos.
- Um **length-prefix** fixo (4 bytes big-endian) torna os *frames* auto-delimitados e evita ambiguidades.
- Defina **limites máximos** por *frame* e valide **LEN** antes de alocar/copiar; implemente um *BufferAssembler* que apenas decifra após *frame* completo.

Proteções adicionais.

- **SEQ** monotónico, janela anti-*replay* e **TIMESTAMP** para diagnóstico e controlo de ritmo.
- Comparação de **TAG** em tempo constante; limpeza de chaves e *buffers* sensíveis quando possível.
- Erros genéricos no protocolo e *logs* internos detalhados com limitação de taxa.

Testes e verificação.

- **Vetores de teste** determinísticos no arranque (*self-test*).
- **Fuzzing** do *BufferAssembler* e do desserializador; testes de propriedades (*round-trip*, rejeição a *bitflips*).
- Testes negativos: nonce repetido, TAG truncada, AAD alterada; **stress tests** com ráfagas e latências variáveis.

8 Testes

Os testes realizados tem como base 2 metodos:

- Verificação da base de dados através do servidor
- Sniffing das comunicações com o uso do Wireshark

8.1 Servidor

Os testes realizados no servidor foram focados na verificação da base de dados e na monitorização das comunicações. Para a verificação da base de dados, é feita a tentativa de ler o ficheiro enviado, e como é possivel verificar nas figuras abaixo, ao ler o ficheiro sem criptografia, é possível visualizar o seu conteúdo perfeitamente, ao ler o ficheiro com criptografia, não é possível visualizar o seu conteúdo pois o servidor guarda a base de dados sem desencriptar, isto leva a que o ficheiro nem seja reconhecido como uma base de dados.

```
^C(venv) [root@serverproj wireguard]# sqlite3 ProjectSECURE.db
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.
sqlite> SELECT * FROM messages;
feda92bf-6b88-46d5-933c-2a5538386471|bom dia|4e650a56-4c70-4cc9-b811-e520026bc1c
c|2025-08-17T15:20:22.757182IZ
```

(a) Sem Criptografia

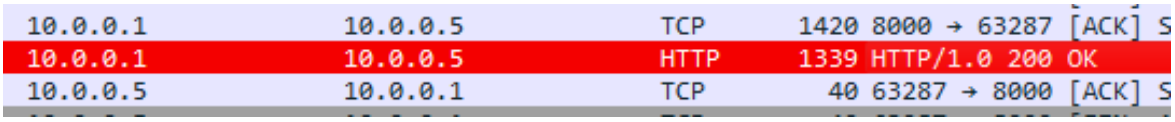
```
^C(venv) [root@serverproj wireguard]# sqlite3 ProjectSECURE.db
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.
sqlite> SELECT * FROM messages;
Error: file is not a database
```

(b) Com Criptografia

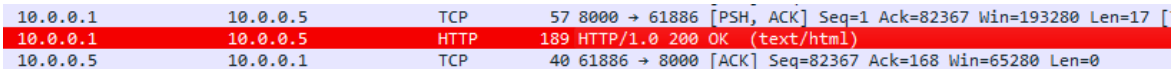
Figura 14: Teste com Servidor

8.2 Comunicações

Para a monitorização das comunicações, foi utilizado o Wireshark para fazer sniffing dos pacotes que estavam a ser enviados e recebidos pelo servidor. Como é possivel ver nas figuras abaixo, estes são os resultados:



(a) Sem Criptografia



(b) Com Criptografia

Figura 15: Teste com Servidor

Para obter esta captura, foi necessário selecionar dentro do Wireshark o adaptador associado à VPN para ver o tráfego que passa lá dentro. Ao analisar o resultado obtido pelo Wireshark não é possível detetar nenhuma diferença óbvia, sem ser a questão do aparecimento do *(text/html)* nas mensagens criptografadas. Isto, só se vai tornar mais visível, quando for selecionada a mensagem, e com o botão direito do rato, selecionar depois a opção *Follow –> TCP Stream*, isto fará com que apareçam os resultados das figuras 16 e 17.

Sendo assim, podemos analisar o conteúdo desta mensagem HTTP, sendo ele na figura 16 a estrutura da base de dados nas fases iniciais do projeto, onde ainda não estava criptografada. Ao analisar o conteúdo após, seria possível visualizar todas as informações da base de dados, incluindo utilizadores, mensagens e outros dados sensíveis.

Figura 16: Wireshark Sem Criptografia

Agora, ao analisar o conteúdo da figura 17, podemos ver que a estrutura da base de dados está ilegível para um ser humano devido à criptografia AES256/Serpent, sendo impossível saber o conteúdo se não for feita a decriptação, e para isto ocorrer é necessário ter acesso à chave, seja para "desenvolopar", seja para desencriptar. Isto garante a segurança das comunicações caso o acesso à VPN seja comprometido, e uma pessoa não autorizada consiga aceder ao tráfego que passa pela VPN.

Figura 17: Wireshark Com Criptografia

9 Melhorias Futuras

Este projeto está sujeito a imensas melhorias, tais como:

- Implementação de novas funcionalidades, como mensagens de voz/chamadas criptografadas;
- Melhoria da interface do utilizador;
- Aumento da segurança das comunicações, adicionando HTTPS em vez de HTTP.

10 Conclusão

Concluimos assim que o projeto foi um sucesso, pois conseguimos desenvolver um sistema de comunicações seguras que permite a troca de mensagens de texto entre os seus utilizadores, sem que haja a possibilidade de terceiros acederem às mensagens trocadas. Em geral, foi um projeto que nos permitiu aprender bastante sobre a área de cibersegurança e comunicações seguras, e que nos possibilitou aplicar os conhecimentos adquiridos ao longo do curso de Engenharia Informática.

Bibliografia

AlmaOS. (2025). *AlmaOS: A Secure Operating System* [Página oficial do AlmaOS]. Obtido agosto 16, 2025, de https://almalinux.org

Bash. (2025). *Bash: The GNU Project's Command-Line Interpreter* [Página oficial do Bash]. Obtido agosto 16, 2025, de https://www.gnu.org/software/bash/

Foundation, P. S. (2025). *Python: A Programming Language* [Página oficial do Python]. Obtido agosto 16, 2025, de https://www.python.org/

Inc., W. (2025). *WhatsApp: A Messaging App* [Página oficial do WhatsApp]. Obtido agosto 16, 2025, de https://www.whatsapp.com/

IPBeja. (2025). *Disciplina: Estágio ou Projeto | IPBeja* [Página EP]. Obtido junho 16, 2025, de https://cms.ipbeja.pt/course/view.php?id=238

JetBrains. (2025). *Kotlin: A Modern Programming Language* [Página oficial do Kotlin]. Obtido agosto 16, 2025, de https://kotlinlang.org/

LLP, T. M. (2025). *Telegram: A Cloud-Based Instant Messaging App* [Página oficial do Telegram]. Obtido agosto 16, 2025, de https://telegram.org/

Microsoft. (2025). *C#: A Modern Programming Language for .NET* [Página oficial do C#]. Obtido agosto 16, 2025, de https://learn.microsoft.com/en-us/dotnet/csharp/

SQLite. (2025). *SQLite: A Self-Contained, High-Performance, Embedded SQL Database Engine* [Página oficial do SQLite]. Obtido agosto 16, 2025, de https://www.sqlite.org/

WireGuard. (2025). *WireGuard: Next Generation Kernel Network Tunnel* [Página oficial do WireGuard]. Obtido agosto 16, 2025, de https://www.wireguard.com/