UDP-WG

This repository contains an emulated implementation of both the UDP network protocol, alongside the WireGuard VPN protocol, as a means to better understand not only these protocols in isolation, but how they might be used in practice. The intention of this repository's creation is primarily as a learning aid to explore Virtual Private Networks through a heavily documented codebase outlining how WireGuard works.

> ⚠ **Warning**
>
> This implementation of UDP and WireGuard is built atop a traditional TCP Linux Socket, and as such these implementations do not work with other WireGuard implementations or servers.

Outline of the Repository

There's three primary ways to approach this repository:

1. Want to dive into the codebase and see how UDP and WireGuard can be implemented in C++? This repository contains heavily documented source files, helping to understand how these protocols work.
    1. Want to better understand UDP? `udp.h` contains our UDP packet implementation, which is used directly by our WireGuard implementation to send packets across the network!
    2. Want to understand how WireGuard works? `wireguard.h` contains our WireGuard implementation, and all the cryptographic algorithms used by it are contained in `crypto.h`. We use both OpenSSL and Sodium for cryptography, allowing you to get a taste of how you might work with both libraries.
    3. Want to understand how to make a network-enabled C++ application? `main.cpp` contains the code for the main application, with `shared.h` containing the functions needed for thread-safe input and output, and `network.h` containing the code for a threaded, Dynamic Network Thread.
2. Want to see these implementations working in action? `main` uses our implementations to allow peers to communicate across the network using UDP packets, alongside connecting to peers acting as WireGuard servers to securely communicate with peers while staying anonymous and avoiding eavesdroppers!
3. Want to view the codebase without descending into the raw C++? A Doxygen site is available in the `docs` folder which provides rendered HTML for all the various namespaces, functions, and classes.

> 🔥 **Tip**
>
> Feeling overwhelmed? Start out with the application! There, you can get a feel on how the code actually comes together and understand the general flow of logic. When you then look into the code itself, or the Doxygen site, you'll have a better appreciation of what the various parts are used for.

Application Walkthrough

This codebase compiles into a network application where peers across the network can communicate with each other using UDP packets. We use WireGuard in a similar fashion to how you might be familiar with VPNs: A private tunnel that masks your IP address and protects your traffic within the tunnel via strong encryption. All peers are capable of becoming a WireGuard server for another peer, which involves the two peers completing a handshake to derive shared transport keys, and then the establishment of a end point for the client on the server's device.

Runtime and Build-Time Dependencies

UDP-WG relies on OpenSSL and Sodium for its cryptographic operations. You'll need to install both if you want to run the application, or build it from source. Due to the nature of shared libraries, the pre-compiled application may be linked against versions of these two libraries that aren't currently on your system. If you see the following error, or something similar:

```
./main: error while loading shared libraries: libsodium.so.23: cannot open shared object file: No such file or directory
```

Then you'll need to compile the application using your own versions. You can build the application by typing `make` within the UDP-WG directory. We use the `g++` compiler from the GNU Compiler Collection (GCC), and use the aforementioned OpenSSL and Sodium libraries. You should be able to find all of these in whatever package manager your distribution uses; below is a few of the most popular distributions:

- Debian-Based Distributions: `sudo apt-get install g++ libssl-dev libsodium-dev`
- Fedora/RHEL: `sudo dnf install gnu-c++ openssl-devel libsodium`
- Arch-Based Distributions: `sudo pacman -Syu gcc openssl libsodium`

We provide two pre-compiled binaries, based on the version of Sodium. Sodium 23 is the current version used by Ubuntu, and this version is accordingly named `main23`. The latest version of Sodium, Sodium 26, has also been built for rolling release distributions, and is named `main26`. If both executables have errors in dynamic linking, you'll need to compile the application so that it links to your specific versions of OpenSSL and Sodium.

Upon the start of program execution, UDP-WG tests its cryptographic implementations to ensure that they work as expected. You should see output like the following:

```
Key Generation: Success!
Key Exchange: Success!
Encryption: Success!
Decryption: Success!
AED: Success: Modified message! Refusing to decrypt!
Hash: Success!
HMAC: Success!
HMAC Authenticity: Success!
MAC: Success!
MAC Authenticity: Success!
If there were any failures, the program is unstable and may not work properly! Ensure that
OpenSSL is on the latest version, and try reco
mpiling/using the precompiled version if issues persist!
Press Enter to Continue
```

If any of the text is in red, or do not specify "Success," then there's an issue with the application that will cause errors should you continue the application. If this happens, try compiling the application for your system by running `make`. The output binary `main`, should work correctly.

Command Line Arguments

To run the application simply run `./mainXX` from your terminal, where `XX` is the pre-compiled version. If you're using a self-compiled version, the executable should just be `main`. To see what options are available on the command line, use the `--help` flag:

```
main (--port=5000) (--addr=127.0.0.1) (--help) (--verbose) (--packet) (--log=main.log)
 --port=5000: The port to listen on for new connections
 --addr=127.0.0.1: The address to listen on
 --verbose: Print verbose information, including from the network thread
 --packet: Print the entire packet for new messages, not just the data.
 --log: Log information to main.log
 --help: Display this message
```

- `--port` specifies the port that the application will bind to for network communication. If you launch multiple instances of the program on the same computer, you'll need to provide a unique port for each. If the port is already in use, the application will report such an error and close; by default, the application binds to port 5000.
- `--addr` specifies the address that should be used by the application. If you're only talking to instances of the program on a single computer, leave this as the localhost address; if you want to talk across a network, use the public IP of the computer.
- `--verbose` prints verbose information to the console.
- `--packet` will print the entire packet in a neatly visualized block, rather than simply printing the source and data contained.
- `--log` will write out status messages to the provided log file, such as `--log=server.txt`
- `--help` displays the help message.

Sending UDP Packets

Once you've confirmed that the cryptographic operations work correctly, you should be brought to the home screen:

```
UDP-WG
Public Key: 228ab162c75472dd6c84636f5a5dd
Not connected to a WireGuard Server
What would you like to do?
1. Send a UDP message
2. View new messages (0)
3. Connect to a WireGuard Server
4. Quit
Input:
```

For sending UDP packets, we're only interested in option one and two. This home page constantly updates, and the (0) value in option two contains the amount of unread messages that you have received. When a new message is received, the value will turn green to alert you! Let's send a message to another computer; to do this, we'll need to have the program running on both machines, and ensure that firewalls and other security systems will permit the traffic to our chosen port. First, we choose option 1, which will bring us to the following dialog:

```
Enter IP:PORT or Alias
```

Here, you can either provide the raw IP:Port combination, or an Alias if you've already sent a message to this peer before. For our case, we're going to send a packet to the instance of this program running on the machine located at

`192.168.101.221`, bound to the default port of 5000:

> **Tip**
>
> Example output in this document will preface user provided input with the > character!

> **Warning**
>
> This program does not resolve domain names, so `mydomain.com:5000` will not work! If you aren't sure what the IP of your peer is, use `ping`!

```
Enter IP:PORT or Alias
> 192.168.101.221:5000
Enter an alias name to associate with this peer!
> peer
```

Here, we specify our machine, and provide an alias. This way, when we want to send another message to the peer, we can simply provide the alias:

```
Enter IP:PORT or Alias
> peer
What would you like to send?
> Hello!
```

You should then be brought back to the home screen, and your data will be packaged into a UDP packet, and sent across to the other peer. Let's take a look at their home screen!

```
UDP-WG
Public Key: f4c87b8d3472b7a5ebec26ef824da87
Not connected to a WireGuard Server
What would you like to do?
1. Send a UDP message
2. View new messages (1)
3. Connect to a WireGuard Server
4. Quit
Input:
```

Notice the new message (Unfortunately, this document doesn't allow us to have color in the code blocks). Let's see what the peer sent by selecting 2!

```
Message from: 16777343:5000
Hello!

Would you like to reply (y/N)
```

The IP address might look a little usual, but this is simply the internal representation of the `127.0.0.1`

> **Note**
>
> The standard IPv4 representation is actually just four bytes separated by dots (Hence why they only go to 255). You can very easily convert these into hexadecimal by simply converting each 255 pair into a two digit hexadecimal value (Max `0xFF` or 255), and putting them together in reverse order. For `127.0.0.1`, we'd convert it into Hex as `7F.00.00.01`. Since network addresses are stored in Big-Endian format, we put them in reverse order `0100007F`, which in decimal is 16777343!

Here, we see the source, alongside the data that we sent. If you want slightly more information, turn on the `--packet` argument!

```
0======1======2======3======4
|       PSEUDO-HEADER       |
=============================
|         16777343          |
|        3714427072         |
|  0   |  17  |     26      |
=============================
|           HEADER          |
=============================
|    5000     |    5000     |
|     26      |    41472    |
=============================
|            DATA           |
=============================
| Hello!                    |
=============================

Would you like to reply (y/N)
```

Here, we see the contents of the Psuedo-Header, including the source address, destination address (3714427072 = `0xDD65A8C0` = `C0.A8.65.DD` = `192.168.101.221`), UDP identifier = 17, and the length of the whole packet 26. In the Header, we see the source and destination port, both 5000, the length again, and the checksum value of 41472. Finally, we have the data. You may notice that the source address is the local address. You may expect an attempt to reply to be send back to itself, since it would be pointed to `localhost:5000`, or the running program, but you can actually reply! Let's reply:

```
Would you like to reply (y/N)
> y
What would you like to send?
> Hi!
```

Sure enough the message gets routed, let's look back at first instance:

```
UDP-WG
Public Key: 228ab162c75472dd6c84636f5a5dd
Not connected to a WireGuard Server
What would you like to do?
1. Send a UDP message
2. View new messages (1)
3. Connect to a WireGuard Server
4. Quit
Input: 2

Message from: 16777343:5000
Hi!

Would you like to reply (y/N)
```

> **Info**
>
> Why does this work? The Network Thread that facilitates all network communications, upon discovering a new destination, automatically gets into contact with the remote Network Thread, and establishes a *File Descriptor* that the two can communicate over. Both Network Threads associate this FD with the claimed source of the other peer, which means the first network thread associates this FD with the destination `192.168.101.221`, whereas the second peer associates the FD with the first peer's claimed source IP, which defaults to `127.0.0.1`. Therefore, all packets sent to `127.0.0.1:5000` will be sent to the other peer. To fix this use the `--addr` command line argument to provide your public facing IP! For example: `./main --addr=192.168.101.1` for the first program, and `./main --addr=192.168.101.221`.

Sending UDP Packets Across WireGuard

Let's say we have three users of the program, Alice at `192.168.101.1:3000`, Bob at `192.168.100.1:4000`, and Carol at `192.168.100.193:5000`. Alice wants to communicate with Carol, but wants to employ the WireGuard protocol to secure the communication. Why might we want to do this?

- Perhaps Eve is also on the `192.168.101.0/24` subnet, and has set her network interface card to promiscuous mode to capture all packets on the network. Alice cannot trust her local subnet, but `192.168.100.0/24` is a trusted network. If she can create a tunnel into the `192.168.100.0/24` network, she can safely communicate within it. She can use WireGuard to create a secure tunnel into this secure network, thwarting Eve by encrypting her traffic within the insecure network.
- Perhaps Alice doesn't trust Carol. Maybe Alice's computer uses a static IP, and she worried that if Carol knew her true address, she may try to use a Denial of Service attack to take Alice off the network. Alice could instead use a WireGuard server to rout her traffic through the server, obscuring her actual IP. If Carol tried to attack her, Alice could simply close the instance with the server.

In this application, WireGuard is used to connect a *Client* with a *Server*. Once a Handshake has been performed, the Server creates an *End-Point* for the Client on their machine. The Client can then encrypt their packets and send them to the End Point. Once received, the Server will decrypt the packets, read the embedded UDP packet, and then send that packet to its intended target. Likewise, other peers on the network can send plaintext packets to the End Point, and the Server will encrypt them and send them to the client. Let's consider the two example situations from before:

- To securely encrypt her packets, Alice will create a WireGuard connection with Bob, who will act as the server. Alice will then encrypt her packets before sending them within the `192.168.101.0/24` subnet. When Bob receives the packets, he will decrypt the content and send the original packet, designated for Carol, in plaintext across the trusted `192.168.100.0/24` subnet. Trying to eavesdrop upon the connection, Eve will be only able to capture encrypted WireGuard packets, and her attempts to read what Alice is sending will be thwarted.
- To hide her IP, Alice will once again create a WireGuard connection with Bob. She will then send her packets to Bob encrypted, who will decrypt them and send them to the distrusted Carol. When Carol receives the packets, they will be sourced from the End Point. While she will seamlessly be able to reply through the End Point, allowing two-way communication (Something simple IP spoofing would be unable to accommodate), if Carol ever tried to attack Alice, she could simply terminate the WireGuard connection with Bob. Bob will then close the End Point, and Carol will have no ability to further communicate with Alice.

> **Note**
>
> The WireGuard protocol can effectively be thought of us as a Key-Exchange, akin to Diffie-Hellman. This means that, in the first example, Alice could just as easily create a WireGuard connection with Carol directly, and cut out Bob as a middle man; this would have the benefit of not allowing Bob to read the decrypted plaintext packets, which is useful if Bob may not be someone who can be fully trusted. However, our application does not accommodate this manner of communication. This isn't because WireGuard does not support it, nor because our implementation can't, but merely because of how we decided to showcase WireGuard in a functional application!

So, how does Alice establish a WireGuard connection with Bob? It's actually quite easy. First, let's have each peer start an instance of the program with the correct arguments:

- Alice: `./main --addr=192.168.101.1 --port=3000`
- Bob: `./main --addr=192.168.100.1 --port=4000`
- Carol: `./main --addr=192.168.100.193 --port=5000`

Then, Alice selects `3. Connect to a WireGuard Server`:

```
Enter IP:PORT or Alias
> 192.168.100.1:4000
Enter an alias name to associate with this peer!
> Bob
Waiting for peer to respond
```

Bob will then receive a notification:

```
Peer: 23439552:3000 Wants to connect to your WireGuard server.
Their provided Public Key: 9e5dd6ca98a6741c4d59013a021d752
Your Public Key: 2fc343ab3fead63c718829d2061de
Accept (y/N)? Or C to send a cookie
> y
```

In this application, public keys are sent across the wire in plaintext. This presents a possible vulnerability in which Eve could intercept the packets and initiate her own Handshake, one with Alice, and another Bob. Then, she could

transparently encrypt the packets using Transport Keys derived from both peers. This vulnerability does not exist in the official WireGuard implementation, as both the public key and configuration is stored in a configuration file, rather than be communicated across the network; rather than require users to generate and point to these configurations, UDP-WG simplifies the exchange by sending the public keys across the wire to initiate the handshake.

With that said, we presume that Bob and Alice have some means to verify the authenticity of the public keys, as the first 16 bytes of the truncated key are displayed both on the home screen, and during the key exchange. If Bob recognizes the key as belonging to Alice, he has three options on how to respond:

1. Accept the Request for a Handshake. This will result in Bob opening an End Point on his computer, located at a random port, that is then used for secure WireGuard communication to and from Alice.
2. Refuse the Connection. One principle of WireGuard is that of *Stealth*. Servers and Clients should not reveal themselves unless the correct information is presented, which means that if the Server does not want to accept the connection, the Client wants to terminate the connection for any reason, or an issue occurs in the Handshake, the responsible peer does not communicate any such termination. Instead, they simply ignore the peer, letting the connection timeout. That way, if Bob doesn't want to allow Eve to establish a WireGuard connection into the subnet herself, he can simply ignore her requests, and Eve would be unable to deduce if Bob actively refused her connections, or if she simply contacted a server that wasn't exposing a WireGuard server.
3. Send a cookie. When under load, a WireGuard server may want to defer a Handshake; rather than simply refusing the connection, Bob would send back an encrypted cookie, generated from a random value that only the server knows, hashed with the IP and Port of Alice's computer. Alice would decrypt this cookie, and after waiting a bit once again request a Handshake, this time providing the cookie. Bob would recognize this cookie, and give Alice priority on the handshake.

Bob isn't currently under load, and recognizes Alice's public key, so accepts the Handshake. He then sends his own public key over to Alice, who will get her own notification:

```
The Server's Public Key: 2fc343ab3fead63c718829d2061de. Is this correct? (y/
N)
> y
```

Again, Alice needs to confirm that she is truly speaking with Bob, and confirms that the public key matches what she is expecting. Once she confirms, UDP-WG will perform the WireGuard handshake, and both Alice and Bob will derive a set of common transport keys that they can use to securely encrypt data to send to one another. These keys are cycled every two minutes by using a set of Ephemeral Keys on top of their Static Keys (The public portion of which was sent across the wire).

> **Note**
>
> The Ephemeral Keys exist to ensures perfect forward secrecy, as even if Alice and Bob have their Static Keys leaked, Eve would be unable to decrypt the prior messages sent across the insecure network as she would not have the Ephemeral Keys, which are randomly generated and constantly changed. Keys are exchanged every two minutes, or after 2^{60} messages

Alice's home screen will then update to reflect this new WireGuard connection:

```
/ /_/ / /_/ / ____/_____/ |/ |/ / /_/ /
\____/_____/_/          |__/|__/\____/
Public Key: 9e5dd6ca98a6741c4d59013a021d752
Connected to server!
What would you like to do?
1. Send a UDP message over WireGuard
2. View new messages (0)
3. Disconnect from the WireGuard Server
4. Quit
Input:
```

Let's go over her new options:

1. `Send a UDP message over WireGuard` will function identically to the original UDP Message option, where Alice can provide the address and port of Carol, give her an alias, and provide the data to send. However, rather than sending this packet directly to Carol, UDP-WG will instead encrypt this packet using the shared Transport Key and send it to the End Point Bob created for her. When this packet receives Bob, he will decrypt it using his own copy of the Transport Keys, get the UDP packet placed within the WireGuard Packet, and send that packet to Carol as the intended destination. By spoofing the source to the End Point, Carol will receive the message as having been sent from the End Point, and her plaintext reply will be sent back to Bob, who will use his shared Transport Key to encrypt the packet before sending it back to Alice, who can then decrypt it and read Carol's reply, again formatted such that source appears as Carol's IP address, rather than the End Point, and the destination is Alice's actual IP address, not the End Point. This ensures a seamless communication, where neither Alice or Carol need to be aware of the WireGuard server routing their packets.
2. `Disconnect from the WireGuard Server` terminates the connection between Alice and Bob. When Alice closes the connection, Bob will notice and immediately close the End Point to prevent any further traffic from being processed.

> **Tip**
>
> The re-keying algorithm that occurs every two minutes occurs silently in the background, avoiding Alice and Bob having to directly engage as they did with the initial Handshake. Upon completion of the handshake, the *mode* changes, so you have a visual indication of when it happens!

Cookies

In the above example, what if Bob was under load and handling numerous WireGuard connections simultaneously? In this case, Bob would want to send a cookie to defer the Handshake process with Alice until a later date, which is mandated to at least five seconds after the initial attempt to handshake. Fortunately, this process is entirely automatic. Recall the prompt that Bob received when Alice first initialized the connection:

```
Peer: 23439552:3000 Wants to connect to your WireGuard server.
Their provided Public Key: 9e5dd6ca98a6741c4d59013a021d752
Your Public Key: 2fc343ab3fead63c718829d2061de
Accept (y/N)? Or C to send a cookie
> C
```

By providing `C` or `c`, Bob will not send a Response Packet, but instead a *Cookie*. The Cookie is a random value that changes every two minutes, hashed with Alice's IP+Port, and further encrypted with the Bob's public key using the original `mac1` value as Additional Authenticated Data. In essence, Bob returns a cryptographic value tied to the initial Handshake between him and Alice, that Alice will promptly decrypt and store. After the timeout period, when Alice requests a second handshake, she will automatically pass that value by calculating a second MAC, stored in the `mac2` section of the Packet, that is a MAC using the cookie value as the key, and the rest of the Packet as data. Bob can then verify that this `mac2` address is valid, both in the context of Alice's previous handshake, and also in the context of the random secret (As if Alice took longer than two minutes, then random secret on Bob's server will have changed and thus invalidated the cookie).

> **Note**
>
> Cookies act like a sort of priority queue for the server. If the server is sent a packet with either an empty or invalid `mac2`, (IE no cookie has been sent or the sent cookie expired), they can send a cookie in response. However, if the `mac2` *is* valid, they will continue the handshake process. This ensures that Alice will only need to wait at a maximum of five seconds, plus the time to actually complete the handshake itself.

Codebase Walkthrough

> **Tip**
>
> This Walkthrough assumes a basic understanding of C++. If you need a refresher, my previous project AES-DH, has some general tips and explanations. That being said, this codebase was written such that you *should* be able to understand the process through the documentation alone, even if the specific idiosyncrasies of C++'s syntax does not make it clear through the code itself.

The codebase is broken up into logical files, each containing a namespace sharing the same name as the file itself. Therefore, if you see functions like `wireguard::Handshake`, you know the file that this function belongs to is `wireguard.h`:

- The `wireguard.h` file contains all the functions and variables used by the WireGuard implementation, including the functions for initiating a handshake, and the structure of the various packets sent between peers.
- The `udp.h` file contains the `udp::packet`, our implementation of the UDP protocol.
- The `main.cpp` file contains the main application.

If you're interested in looking at some of the auxiliary files:

- `crypto.h` contains the implementations of the cryptographic algorithms needed by WireGuard. They use implementations from both OpenSSL and Sodium.
- `shared.h` contains various shared objects and functions, particularly mediated input and output.

> **Tip**
>
> Unsure which files to start with? Consider starting with `shared.h`, which provides the common definitions used by all the subsequent files. Then, move to `crypto.h` to get an understanding of the underlying cryptographic functions and data structures. With that, you'll be equipped to understand `udp.h` and `wireguard.h`, in that order, and can finally finish with `network.h` and `main.cpp`. That said, feel free to jump to whichever section

most interests you, the documentation should be more the sufficient such that you don't need to read the entire codebase to understand a particular part!

UDP

> **Note**
>
> The implementation of the UDP protocol was created in reference to RFC 768

The WireGuard Protocol passes data over the UDP protocol, and as such this repository provides a sample implementation of UDP for these packets to be sent over. The `udp::packet` consists of three major components:

1. The Psuedo-Header: This contains the source and destination IP addresses, alongside a protocol number (17 for UDP), and a length for the entire packet.
2. The Header: This contains the source and destination port the length of the entire packet, and a checksum to ensure that the values in the Psuedo-Header and Header were not changed in transit.
3. The Data: Raw bytes that compose the content of the packet.

> **Note**
>
> The Psuedo-Header is typically treated as an ephemeral part of the packet, meaning that it's not actually sent as part of the packet. The reason this is possible is because both the IP addresses of source and destination are already available a layer down, within the IP Header. The Psuedo-Header exists such that these addresses can be included in the checksum calculation. UDP-WG does sent the Psuedo-Header, as its emulated network stack does not have access to the raw IP headers that are used by the Socket.

WireGuard

> **Note**
>
> The implementation of the WireGuard protocol was made in reference to the original whitepaper

All the code related to the WireGuard implementation are available in the `wireguard` namespace, and can be broadly classified into the following groups:

1. The Static Keys: UDP-WG generates a set of static keys on program start. This simplifies the handshake process, and removes the need to generate physical files that are then sent between the respect peers and loaded into the program.
2. Constants: WireGuard uses many fixed constants as the start for cryptographic operations in the handshake. Some examples include the `CONSTRUCTION` constant used to create the Chaining Key Value used to eventually derive the Transport Keys and the `LABEL_MAC1` and `LABEL_COOKIE` that are to create the `mac1` value of the handshake packets, and the cookie respectively. Another type of constant is the `REKEY` and `RJECT` constants, which specify either the amount of time in seconds, or messages sent, that require a `REKEY`, or an outright rejection of the connection.

> **Info**
>
> According to the WireGuard reference, a WireGuard server will continue to handle packets after the `REKEY` threshold has been reached, although it will continually prompt the client to re-key. Once the `RJECT` threshold has exceeded, the server will refuse to handle packets until a successful re-key occurs.

3. Configuration: During a Handshake, UDP-WG will construct a `config` structure that contains all the relevant information needed to communicate with the peer. For a client, this value is stored in the `wireguard_server` variable in `main.cpp`. For the server, this value is passed onto the newly create WireGuard Thread that manages the End Point.
4. Random Value: The cookie that is sent by the server when under load uses a random value that changes every two minutes. the `Rm` is a class that implements this behavior, with the `cookie_random` an instance of this object that is used when creating a cookie.
5. Packets: The WireGuard Reference contains four unique packets that are sent during handshake and subsequent communication:
    1. The Initial Packet, sent from the Initiator of a handshake to the Responder.
    2. The Response Packet, sent from the Responder back to the Initiator.
    3. The Cookie Packet, which is sent by the Responder to the Initiator in lieu of a Response Packet when under load
    4. The Transport Packet, which is sent after a handshake has completed, and includes the data encrypted by the shared Transport Keys.

> **Info**
>
> These packets are implemented as derivatives of an abstract `Packet` class. This is due to the need to convert the `cryto::string` used to store secrets and other values in WireGuard, to raw bytes that can be sent across the network. The `Packet` class contains two methods: `Packet::Serialize()`, which convert a `crypto::string` packet into a string of bytes, and `Packet::Expand()` which performs the inverse of `Serialize`.

6. Handshake Functions: The WireGuard Handshake occurs in two stages. In the first stage, `Handshake1`, the Initiator generates their set of ephemeral keys, ties the eventual Transport Keys to these values and their Static Keys, before sending it across to the Responder who inverses the process using the Initial Packet sent by the Initiator to arrive at a shared value `C` and `H`. In `Handshake2` the Responder generates their own set of Ephemeral Keys, and ties both them and their Static Keys to these shared values. The Initiator preforms the inverse of the process using the Response Packet sent by the Responder, and both arrive at a set of shared Transport Keys. This communication is controlled by the `Handshake` function. `Handshake` also manages generation and parsing of cookies.

Cryptographic Functions and Others

The auxiliary cryptographic functions used by the WireGuard implementation relies on two libraries:

1. OpenSSL, which provides the implementation of:
    1. `HASH` using BLAKE2s hashing algorithm
    2. `MAC` using BLAKE2s' keyed-hash functionality

    3. `HMAC` using HMAC-BLAKE2s.
2. Sodium, which provides the implementation of:
    1. `DH` and `DH_GENERATE` using the Curve25519 ECC algorithm.
    2. `ENCRYPT` and `DECRYPT` using the ChaCha20-Poly1305 stream cipher and message authentication code.
    3. `XENCRYPT` and `XDECRYPT` using the XChaCha20-Poly1305 stream cipher and message authentication code.

> **Note**
>
> The WireGuard Reference uses the name `AEAD` (Authenticated Encryption with Associated Data) and `XAEAD` to refer to the encryption and decryption functions using the standard ChaCha20, and extended XChaCha20 functions respectively. The Difference between ChaCha20 and its extended variant come from the increased size of the nonce value. ChaCha20 uses a 96 bit nonce, whereas XChaCha20 uses a 192 bit nonce; when a nonce is chosen at random, the latter is a stronger mode of encryption. ChaCha's name is a delightful reference to the algorithm that it was based upon: Salsa.

Additionally, there are other members of importance:

1. The `KDF` function implements the `HKDF` key derivation algorithm as outlined in the WireGuard reference, utilizing the above functions.
2. The `crypto::string` is a class for storing unsigned cryptographic bytes that are wiped when leaving scope.
3. The `crypto::keypair` is a general purpose pair of `crypto::strings` that are used both in a private/public configuration, such as the Static and Ephemeral keys, and as a unrelated collection of two values, such as the ciphertext and nonce returned by `XENCRYPT`.

These cryptographic functions exist within the `crypto` namespace.

Another part of the WireGuard standard is the use of timestamps, to which the TAI64N format is mandated. The implementation of this format, and auxiliary functions for working with it, exist in the `shared` namespace.

UDP-WG

Code pertaining to the creation of the actual application utilizing the UDP and WG implementations is available in:

1. The `shared` namespace contains various functions and structures used throughout the program, particularly handling thread-safe input and output.
2. The `main.cpp` file contains the code that drives the application.
3. The `network` namespace contains the Network Thread and associated `network::queue` used for sending packets across the network, discovering new peers, and maintaining existing connections.