WIRE EXPOSED

Investigative Report

&yet

Research led by Philipp “Fippo” Hancke

&yet Chief WebRTC Engineer

WebRTCHacks Contributor

ABOUT THIS REPORT

WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs. The WebRTC.org library allows developers to obtain a high-quality, state of the art networking and media library that is also used in the Google Chrome browser free of charge. As such, it has been used by a number of services to implement Voice and Videocalls.

Is Wire one of those services? Read on...

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Once again, thanks to Serge Lachapelle, Henrik Lundin and Peter Saint-Andre.

While the data capture is trivial, the interpretation is a black art. All remaining errors are my own.

SUMMARY

Wire is an attempt to reimagine communications for the mobile age. It is a messaging app available for Android, iOS and Mac and support audio calls, group messaging and picture sharing. One of it's often quoted features is the elegant design. As usual, this report will focus on the low level apsects and wire, and leave the design aspects up for the users to judge.

Half a year after launching the Wire Android app has been downloaded only between 100k and 500k times. Recently they also launched a web version, powered by WebRTC. Based on this, it seems to be stuck with what Dan York calls the directory dilemma.

What makes Wire more interesting from a technical point of view is that they're strong proponents of the opus codec for audio calls. Maybe there is something to learn here...

The wire blog explains some of the problems that they are facing in creating a good audio experience on mobile and wifi networks:

> The WiFi and mobile networks we all use are "best effort" — they offer no quality of service guarantees. Devices and apps are all competing for bandwidth. Therefore, real-time communications apps need to be adaptive. Network adaptation means working around parameters such as variable throughput, latency, and variable latency, known as jitter. To do this, we need to measure the variations and adjust to them in as close to real-time as possible.

Given the preference of ISAC over opus by Facebook Messenger, the question which led to investigating Wire was whether they can show how to successfully use opus on mobile.

The Wire stack is described as "a derivate of WebRTC and the IETF standardized Opus codec"

It's not quite clear what exactly "derivate of WebRTC" means. The STUN implementation turned out to be based on the BSD-licensed re library by creytiv.com, which is compatible with both the Chrome and Firefox implementations of WebRTC. Binary analysis showed that the webrtc.org media engine along with libopus 1.1 is used for the upper layer.

What we found when looking at Wire was, in comparison to the other apps reviewed, was a more "out of the box" WebRTC app, using the protocols as defined in the standards body. Audio quality did turn out to be top notch, as our unscientific tests on various networks showed.
Testing on simulated 2G and 3G networks showed some adaptivity to the situations there.

Wire is company that prides itself on the user privacy protection that comes from having it's HQ in Switzerland, yet has it's signalling and TURN servers in Ireland. They

get strong kudos for using DTLS-SRTP. To sum it up, Wire offers a case study in how to fully adopt WebRTC for both Web and native mobile.

TERMS OF SERVICE

Wire's Terms of Service, available here state the following:

> 7.5 No Data Mining or Harmful Code. You agree that you will not [...] (ii) intercept, examine or otherwise observe any proprietary communications protocol used by the Apps, the Site and/or the Service, whether through the use of a network analyzer, packet sniffer or other device;

Fortunately, we are not really interested in any of the "proprietary communications protocol" here, just standard internet protocols like STUN, TURN, DTLS, RTP and RTCP and the usage of the WebRTC API. This kind of TOS is subpar for the users. Users simply want to be able to verify that the app does what it promises.

Arguably, the signaling protocol used could be called "proprietary". However, this can be inspected using the browser's network console which intercepts the traffic by default.
It's a websocket-based protocol using JSON payloads. The signaling servers are located in an Amazon datacenter in Ireland.

WEBRTC INTERNALS

File: wire-internals.html

Let's first take a look at the WebRTC API usage in the web application by using chrome://webrtc-internals.

In the getUserMedia call, there are two mandatory constraints:

- googEchoCancellation2: true
- googNoiseSuppression2: false

This enables echo cancellation and disables noise suppression.

From the peerconnection configuration we can also see that a single TURN server is used.

```
{ servers: [turn:54.155.57.143:3478], iceTransportType: all }
```

That means only TURN over UDP on port 3478 is used. This server is located in Ireland which increases the delay somewhat for calls originating from the west coast of the United States.
This seems to be the only server, compared to large number of relay servers found in the Whatsapp analysis.

Now let's look at the API traces. It starts off with adding an audio stream:

```
label: d97b9053-195e-43a8-9b0c-57a9e429a6ac, audio: [WebAudio-
98f09235-49d0-4c78-a474-9ada9b5be39d]
```

Looking at the track label, the audio track seems to have passed through the WebAudio API before it is added to the peerconnection. The WebAudio API allows processing the microphone stream with various filters or to manipulate stream's gain.

```
v=0
o=- 2909924584 1394431680 IN IP4 172.16.42.124
s=-
c=IN IP4 172.16.42.124
t=0 0
a=tool:avs 1.4.652 (arm/linux)
a=ice-options:trickle
a=x-OFFER
m=audio 53622 RTP/SAVPF 96
a=rtpmap:96 opus/48000/2
a=fmtp:96 stereo=0;sprop-stereo=0
a=rtcp:53623
a=sendrecv
a=rtcp-mux
a=ice-ufrag:KNR9nj38hicQ2TZ
a=ice-pwd:yxlUFj64m69Ex7N8McvLHdrmLoYPBpc
a=fingerprint:sha-256 8A:35:AF:5A:1F:77:4A:46:F5:4C:67:C-
B:67:F9:3F:B9:A3:C9:8E:FB:B5:1C:DA:8A:6A:48:61:BA:0D:11:34:A6
a=setup:actpass
a=candidate:1 1 UDP 2113929471 172.16.42.124 53622 typ host
```

This SDP shows that the opus audio codec is the only supported audio codec. As we saw in Messenger's usage of WebRTC, it is tuned for mono usage. The opus inband forward error correction is not used. Additionally, DTLS, rtcp-mux and trickle-ice are used, a host candidate is included in the initial offer.
Looking at the corresponding answer created by Chrome, there is no attempt to munge the SDP to restrict Chrome to mono as well.
The next interesting thing are the two ICE candidates which arrive after roughly two seconds (well, they had to go to Ireland). The order is somewhat interesting here as the relay candidate arrives before the server-reflexive candidate.
What is further interesting is that both Conn-audio-1-0 and Conn-audio-1-1 are printed in bold. While both clients have a direct p2p connection for sending the RTP data, the RTCP-connection is relayed via the TURN server. A rather interesting and sporadic bug it seems.

Let's dig deeper...

CAPTURE SETUP

The capture setup should already be familiar from previous reports but is repeated here for completeness. To capture the traffic, a Wifi Pineapple Wifi router (Mark IV, the Mark V works as well) is used.
The Android and iOS devices running the Wire app are connected via WLAN to the Pineapple which itself is connected via Ethernet to a NAT router that connects to the internet.

Even though the Pineapple has various useful capabilities for intercepting, a tcpdump installation is all that is required. See here for installation instructions.

In this setup, the packets from the WLAN interface (wlan0) need to be forwarded to the LAN interface (eth1). This is accomplished with the wp4.sh script with the following settings:

```
wp4.sh with eth1 and wlan0 instead of defaults:
Pineapple Netmask [255.255.255.0]:
Pineapple Network [172.16.42.0/24]:
Interface between PC and Pineapple [eth0]: wlan0
Interface between PC and Internet [wlan0]: eth1
Internet Gateway [192.168.1.1]:
```

The Pineapple uses a 172.16.42.0/24 network and the Ethernet port connected to the local network at &yet's offices. The IP 192.168.1.157 is assigned to the Pineapple by DHCP.

Two mobile devices are used, an Android device with IP 172.16.42.124 and an iPad with IP 172.16.42.187. Additionally, a laptop with IP 192.168.1.202 is used running the Chrome (42) browser using app.wire.com.

Capturing is done by executing
tcpdump -v -i wlan0 -A -w some.pcap
This should usually be done in the /usb directory with an USB stick attached for storage. The captured PCAP file can then be analyzed in Wireshark or similar tools.

SESSION 1

Dump: wire-ios-chrome.pcap

This session shows the Chrome browser on 192.168.1.202 calling the iPad on 172.16.42.187.

The first interesting packet is #203, showing an allocate request to the TURN server running at 54.155.57.143. It contains a software field, giving a hint about the library used by the Wire app to implement STUN and TURN:

```
libre v0.4.12 (arm64/darwin)
```

libre is a BSD-licensed "toolkit library for asynchronous network IO with protocol stacks including SIP, SDP, RTP, STUN, TURN ICE" available from http://creytiv.com/re.html. It runs on a number of platforms including iOS and Android and was last released in March 2015.
Let's stay with this TURN flow a little by setting the following filter:

```
udp.port eq 3478
```

As usual, the TURN server requests credentials from the client in packet #216 with the REALM attribute set to wire.com. The SOFTWARE attribute shows that the restund TURN server, which we also use on Talky is used. The TURN server (only one was observed during testing) is located in an Amazon datacenter in Ireland.

Packet #227 shows an allocate success response, meaning that the client has successfully allocated a relay candidate on the TURN server. Somehow, the corresponding allocation request seems to be missing from the dump. In packets #228 and #229 the client is creating TURN permissions for the peer's candidates. We can see the username in those packets:

```
d=1430497162.v=1.k=0.t=s.r=mxspfhrqbrmjxfci
```

The packet was captured on April 30 at 09:19:29 UTC. Converting the number 1430497162 into a UTC timestamp yields May 1st at 16:19:22. This suggests the usage of ephemeral credentials as described in this draft with an expiration time of thirty hours.

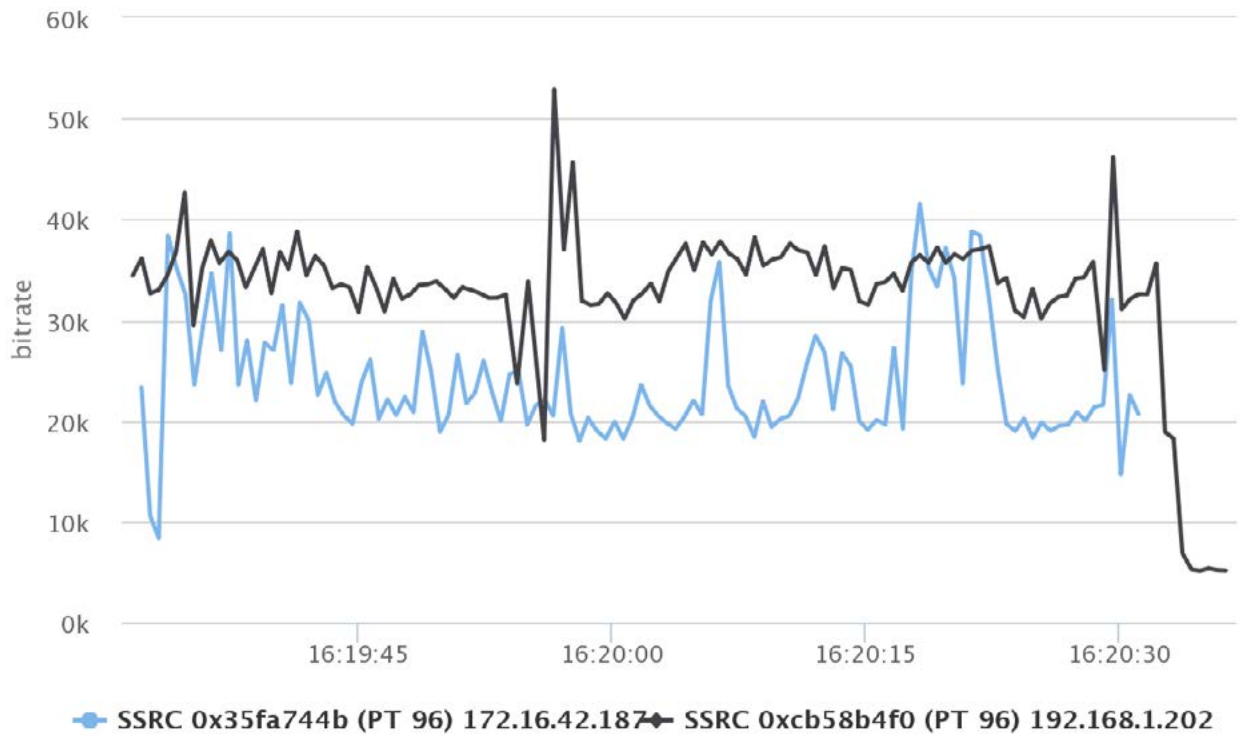
The permissions are created and in packets #248, #264, #265, #266 and #267 we can see some ICE messages wrapped in SEND and DATA indications.
When terminating the call, the client expires the allocation by sending a refresh request with a lifetime of 0 in packet #6605.

Let's go back to non-TURN packets. In packet #204 we can see a STUN binding request going from the iPad to the browser. The username suggests the use of standard RFC 5245, also the request contains a USE-CANDIDATE attribute which means the client implements an ICE feature called aggressive nomination. Subsequently, we can see a number of binding requests flying between the two peers.
In packet #275 we can see a DTLS client hello. Surprisingly, this is not directly between the clients but sent from Chrome via the TURN server. Packets #276 and #281 are also relayed via the TURN server while the concluding CHANGE-CIPHER-SPEC message arrives directly from the iPad in packet #282.
After that, we can see RTP traffic flowing from Chrome (via the TURN server) to the iPad. No traffic is flowing in the other direction until packet #313. There we see RTP traffic going from the iPad directly to Chrome without passing the TURN server. Chrome stops using the TURN server and responds on the P2P connection in packet #581 about three seconds after the first RTP packet. This is likely just a hickup within the ICE process. As a result, some of the RTP packets are arriving out of sequence for about 200ms.

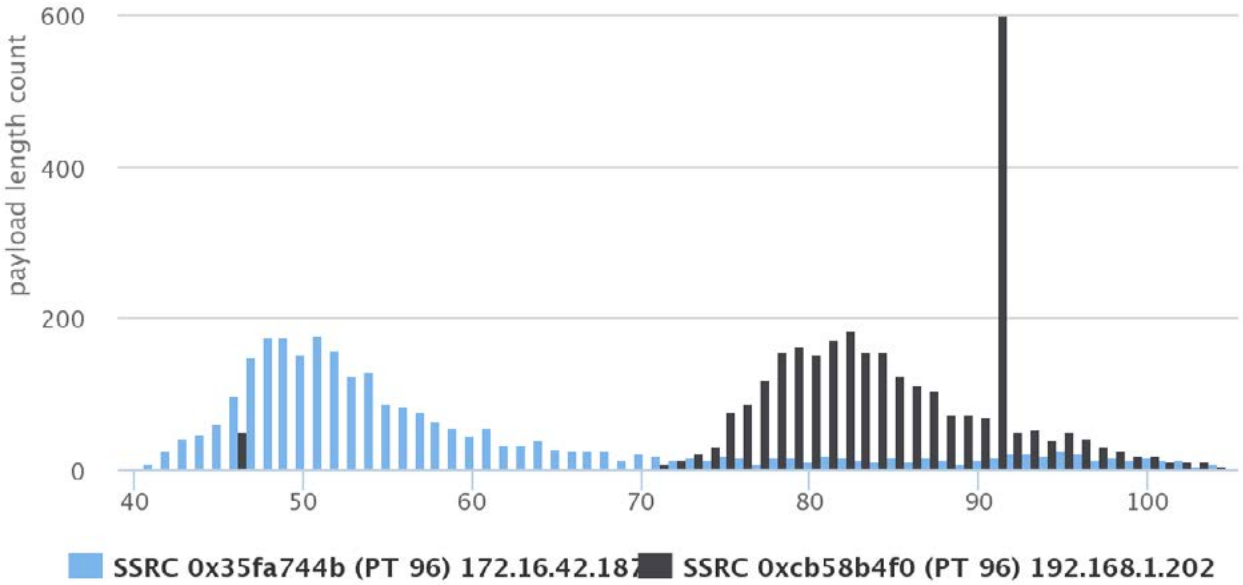
Let's look at the two RTP streams:

- 0x35fa744b sent from the iOS app
- 0xcb58b4f0 sent from Chrome.

Both use the payload type 96, which, as we have seen from the SDP earlier is opus and both have a timestamp difference of 960, i.e. a framesize of 20ms and a packet rate of 50/second.
The bitrate sent by Chrome is roughly at 35kbit/second while the Wire iOS sends about 25kbit/second as can be seen in this graph:



When looking at the distribution of the payload length there is a noticeable difference here:



On average, the packets sent by the iOS app (blue) are about 30 bytes smaller, the size distribution is shifted to the left compared to the Chrome (black) one. This is probably the result of a lower target bitrate setting. The big spike at a payload size of 91 bytes is caused by a long period of time where just this packet size is sent by Chrome.

SESSION 2

Dump: wire-android-ios.pcap

This session shows the Wire App on Android (172.168.42.124) calling the iPad on 172.16.42.187.

The first thing noticeable in the dump is the enormous amount of binding requests from 192.168.1.202. Apparently a leftover from a previous session with Chrome.

Since we know that the sessions typically kicks off with allocating relay candidates, let's set the filter to

```
stun.type == 3
```

to find those requests. They are found in packets #1007 and #1027 for iOS and #1052 and #1076 for Android. It is quite interesting that the username is the same for both clients:

```
d=1430498024.v=1.k=0.t=s.r=pjtvoaaajcuglpqo
```

Possibly the r= parameter (which is different from session 1) is specifying the session id and allows correlating between the TURN credentials and sessions.

Close to packet #1007 in packet #1013 we already see a DTLS client hello sent by the iPad. Let's following the UDP stream by setting the filter to

```
(udp.port eq 41460 and udp.port eq 36578)
```

As it seems, the client exchanged only a single binding request before this happens, in packets #1010 and #1012. That is pretty fast, however the DTLS client hello is never answered. Let's look at it again... compared to the typical DTLS client hello from Chrome or Firefox it seems rather large. The reason for that is the inclusion of an enormous list of 91 TLS ciphersuites, each of which is specified by a two-byte integer. For comparison, both Chrome and Firefox include about ten ciphersuites each.
The list of ciphers includes some very strong and state of the art methods like

```
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
```

This algorithm is slightly stronger than the current IETF recommendation published recently in RFC 7525 even.

If the clients is connecting to another client that supports this cipher, they will choose it.
When connecting to Chrome or Firefox, this will typically end up with something like

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
```

which still satisfies the IETF's current recommendation.

However, the handshake also includes (at the bottom, with a low preference) methods like

```
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
```

Export ciphers are a relict from the time when US laws forbade exporting strong cryptography. They're intentionally weak and do not offer much protection. Modern software never use or include those ciphers.

> **Comment**
> This should be fixed in future versions.

Note that this does not tell anything about the encryption of the voice traffic. This is encrypted with one of the methods specified in the use_srtp extension that is part of the handshake. This is listing a single method

```
SRTPProtectionProfile SRTP_AES128_CM_HMAC_SHA1_80 = {0x00, 0x01}
```

The key for this is derived from the secret generated during the DTLS handshake as described in the DTLS-SRTP specification.

In packet #1100, about one second after the initial attempt, the iPad resends the DTLS client hello. This time, it gets a response in packet #1110 and the handshake proceeds in packets #1112 and #1118.

Let's look at the two RTP streams again:

- 0x0ac58b37 sent from the iOS app
- 0x74dd8245 sent from the Android version.

Both use the payload type 96, which, as we have seen from the SDP earlier is opus. So Wire choses to use opus on mobile as well. Which is not very surprising given their involvement in the creation and standardization of the codec.

Looking at the bitrates of both streams, they seem to be roughly around 35kbit/s with iOS being slightly lower than Android. No adaption of the packet interval as described here could be observed. Maybe this only happens when the app detects it is on a 3g connection.

SESSION 3

Dump: wire-ios-android3g.pcap

This session is again a call between the iPad and the Android device. This time however, the Android device is on a 3g network with an IP 32.142.39.115. We therefore don't get the full traffic from that device.
In packet #381, we see the iOS client sending a binding request to 10.157.81.150. Supposedly, the android client is behind Carrier Grade NAT. This does not bode well...

Next, we see the familiar game of allocating a relay candidate at the TURN server. Since this process is already familiar, let's set the filter to "dtls".
Then, in packet #436 we see a DTLS client hello, followed by another one in packet #442. The client replies in packet #443 with a response containing multiple messages from the same so-called flight. This packet contains the Server Hello, Certificate, Server Key Exchange, Certificate Request and Server Hello Done messages.

In packet #452 about a second later we see the server resending the flight since it did not get a reply from the client. It is splitting up the messages into multiple UDP packets (#452 to #456) in order to avoid exceeding a maximum packet size limit (see here).

Whether caused by that or simply being lucky it gets a response in packet #457. Actually, this response might just be delayed from packet #443.
Subsequently, we can see a fragmented response from the peer arriving as well in #463 - #467 which in turn is answered which messages contained in separate UDP packets in packets #468 - #470.

In packet #472, we see the iOS client sending the first RTP packet of the session with an SSRC of 0x28b01831. This stream is sent with an average bitrate of about 25 to 30kbit/second, a packet rate of 50/second and a timestamp difference of 960.
In the other direction, the first RTP packet is #514 with an SSRC of 0xf674d678. Let's set the filter to

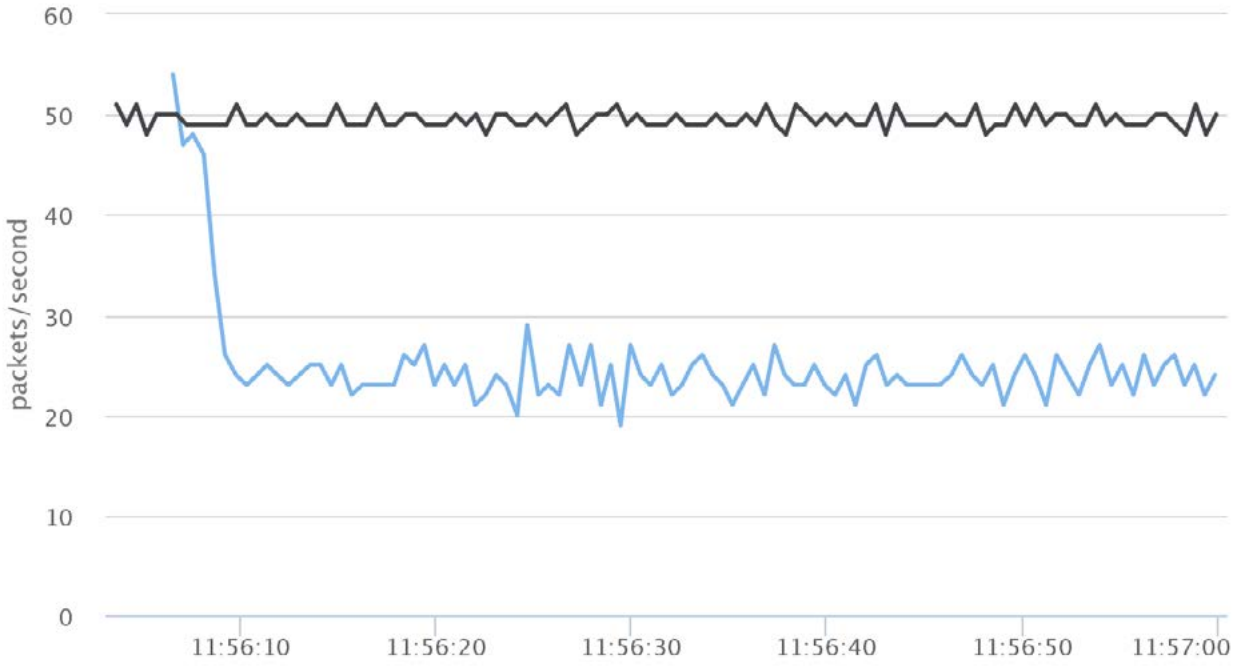
```
rtp.ssrc == 0xf674d678
```

and follow this stream. Also set a time reference by right-clicking on that packet and "set time reference (toggle)". The RTP sequence numbers of the stream look pretty good but notice the large 2.5 second gap between packets #565 and #696. This is just the start, there is another two second gap between packets #741 and #836 and (hold your breath) seven seconds between packets #1073 and #1438. Remember the earlier word about "Carrier Grade NAT".
So the packets are arriving with a very large delay without getting lost.
During those periods, an unobtrusive sound is played which is a nice way to give the user some feedback about why there is no audio from the remote side.

There seems to be a good and loss-free period between packets #1738 and #1828. Looking at the packets there timestamp difference still seems to be 960, so there is no adaption to the network conditions. But the conditions seem so bad, getting things to work seems next to impossible.

SESSION 4

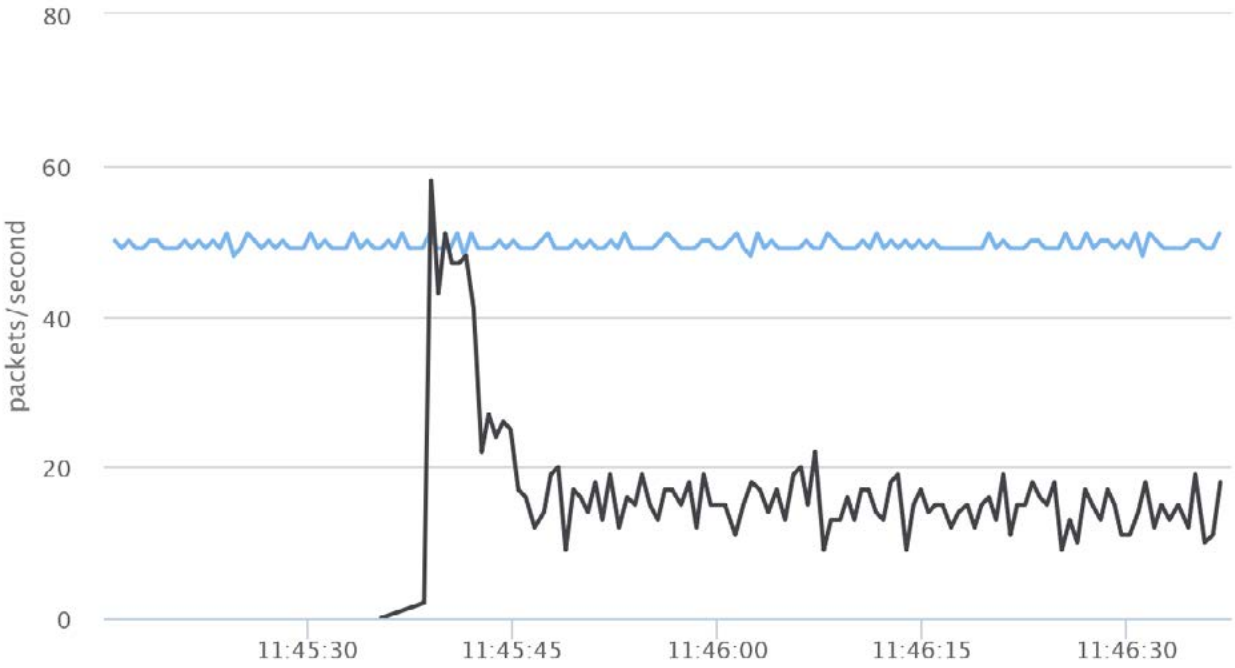
This session is testing Wire with a simulated 3G network on on end. No PCAPs this time, but the most interesting aspect here is the packet rate which is shown below:



Compared with other sessions, which used at rate of 50 packets/second, the blue graph of the client using the 3G networks shows a much lower rate of 25 packets per second. And while the received bitrate varies strongly between 10 and 30kbit/s. This seems to be one of the adapations to the different network properties.

SESSION 5

Lets go to even worse networks, this time 2G and look at the packet rate again.



The client (shown in black) starts with 50 packets/s, drops down to the 25 packets we have seen in the 3G session and then drops down even further to about 18 packets/s.

The payload length distribution also showed some unusually large opus payloads. The reason for that is a dynamic switch between framesizes of 20ms and 60ms. This is similar to what was observed with Facebook Messenger, but seems to be adapting to the the actual network situation.

BINARY ANALYSIS

We already saw that Wire is using libre as a network layer. But what is used for the higher-level media functions? To find that out, let's take a look at the libraries in the APK of the Android version 1.27.121. Unpacking the APK yields a number of folders containing shared libraries. Again, let's use the standard unix `strings` utility to find out more about the contents of one of those libraries, libavs.so with the following command line:

```
strings libavs.so | grep -i webrtc
```

The output shows alot of lines like the following:
mediaengine/webrtc/modules/audio_device/audio_device_impl.cc
pointing to files in the webrtc.org media engine library as well as numerous function signatures.

This suggests that libre is hooked up with the WebRTC voice engine, similary to the way Mozilla choose to use their own network layer instead of the one provided by the webrtc.org library.

Similarly, grepping for opus

```
strings libavs.so | grep -i opus
```

shows strings like

```
libopus 1.1-fixed
```

which suggests that a modified version of the libopus library from https://www.opus-codec.org/ is used. It's not clear however whether this is from the 1.1 release or already includes changes from the 1.1.1 beta release.