Background:

Research has been done for many years into more efficient bandwidth probing techniques compared with the conventional method of "flooding" a link to determine the maximum bandwidth. They focus on quicker, less data-wasteful estimations of available bandwidth by attempting to identify the bottleneck link capacity. Since the late 90's, the most prominent such techniques researched have been dividing packets into pairs and trains – the basic idea being to measure the delay in receiving packets of the same pair/train and inferring available bandwidth from this time delta.

However, CRUSP creators Raida et al. cite Singaporean research that packet pair and packet train methods are unideal in wireless networks. As such, they propose a technique that divides packets into bursts and estimates available bandwidth using the delay between bursts rather than intra-packet pairs. It can be seen as a hybrid between the pair and train techniques. I'll provide results from lab this semester into bandwidth measurements in various scenarios. We will treat the iperf3 utility as a baseline ground truth to compare results to.

Sources:

CRUSP research paper: https://publik.tuwien.ac.at/files/publik_271502.pdf

CRUSP source code & documentation: https://gitlab.com/gitlabwolf/infrastructure-for-mobile

My adapted CRUSP setup: https://github.com/VinnieKhanna/CRUSP

- Changes were only out of necessity to run the tool locally. They included changing the cargo configuration to run outside the given Docker container, disconnecting auxillary services, and replacing the Rust http request library. The actual workings/calculations of measuring bandwidth should be unaffected.
- Also contains raw output and scripts for collecting and visualizing output.

Setup:

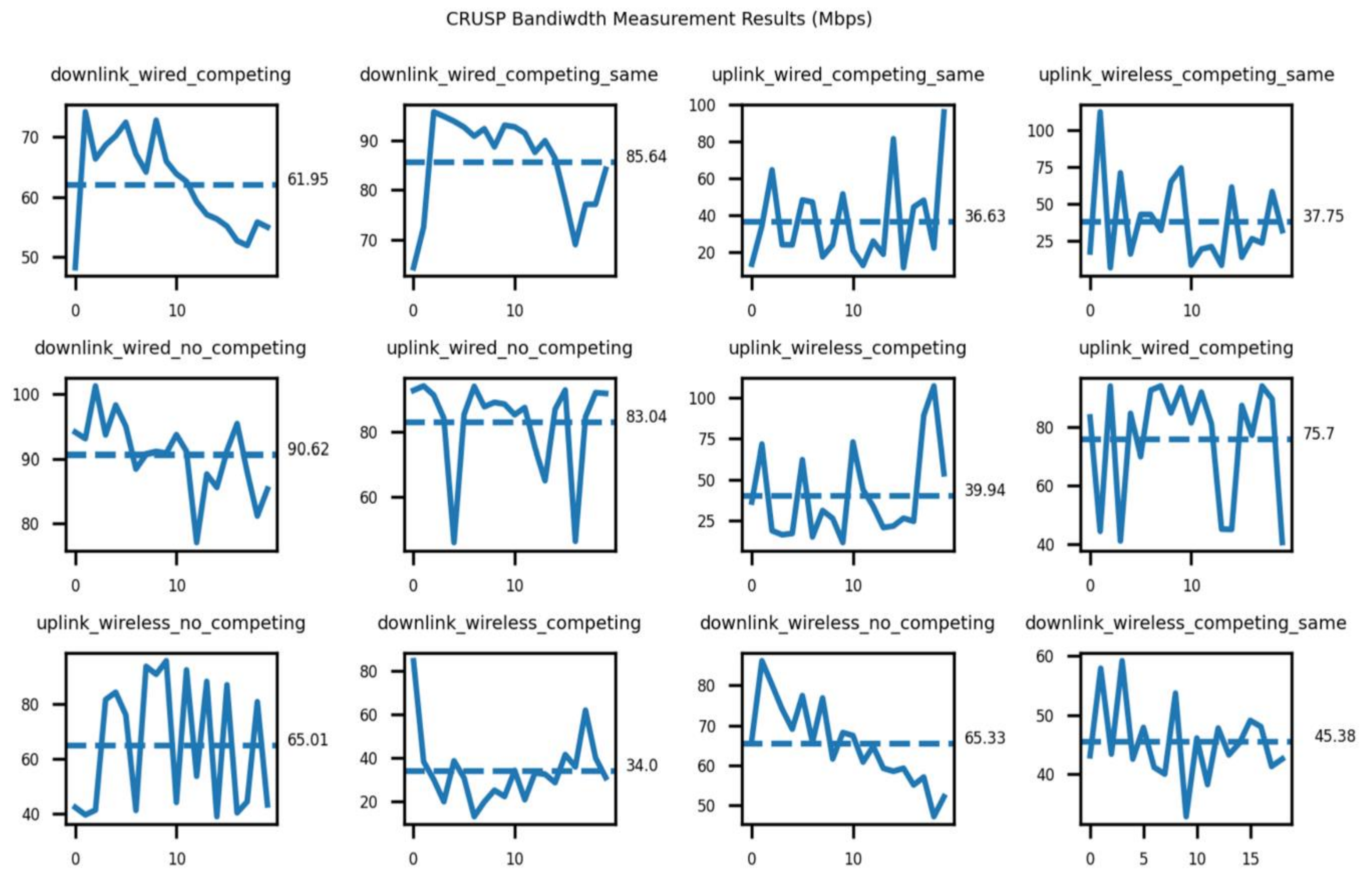
- LAN/WLAN through tp-link2 router (nuk, laptop, phone)
- CRUSP measurement server running on EPL nuk
- CRUSP standalone client (excluding supporting services like settings verification, database, etc.) running on laptop
- Simulating competing network traffic using mobile device and iperf3 client

Results:

I performed tests altering three independent variables:

1. Wired vs wireless: this refers to whether the measurement client (my laptop) was connected to the router through ethernet or WiFi. Naturally we might expect a wireless connection to have less reported bandwidth, and we **did** observe this.
2. Downlink vs uplink: CRUSP-terminology for measuring sending vs receiving rate. Of course, ethernet/WiFi are bidirectional and should theoretically have the same results – though this is very much **not** what we observed.
3. Competing traffic: since we are estimating *available* throughput, we very much expect our results to downwardly adjust given competing traffic – namely from a constant iperf3 request from my phone to the nuk/server. We **did** observe this.
    a. In our raw result graphs, the "no_competing" vs "competing" suffixes should be clear, but you may wonder what "competing_same" means. That is the scenario when the competing iperf3 traffic is spawned from my laptop itself to the nuk, i.e. the same device running the CRUSP client, rather than my phone.

Below are the results from running my measurement bash script in 6 different scenarios (performs both uplink and downlink tests), which runs the CRUSP measurement 20 times each. Reading output and visualization was done with Python/Seaborn/Matplotlib.

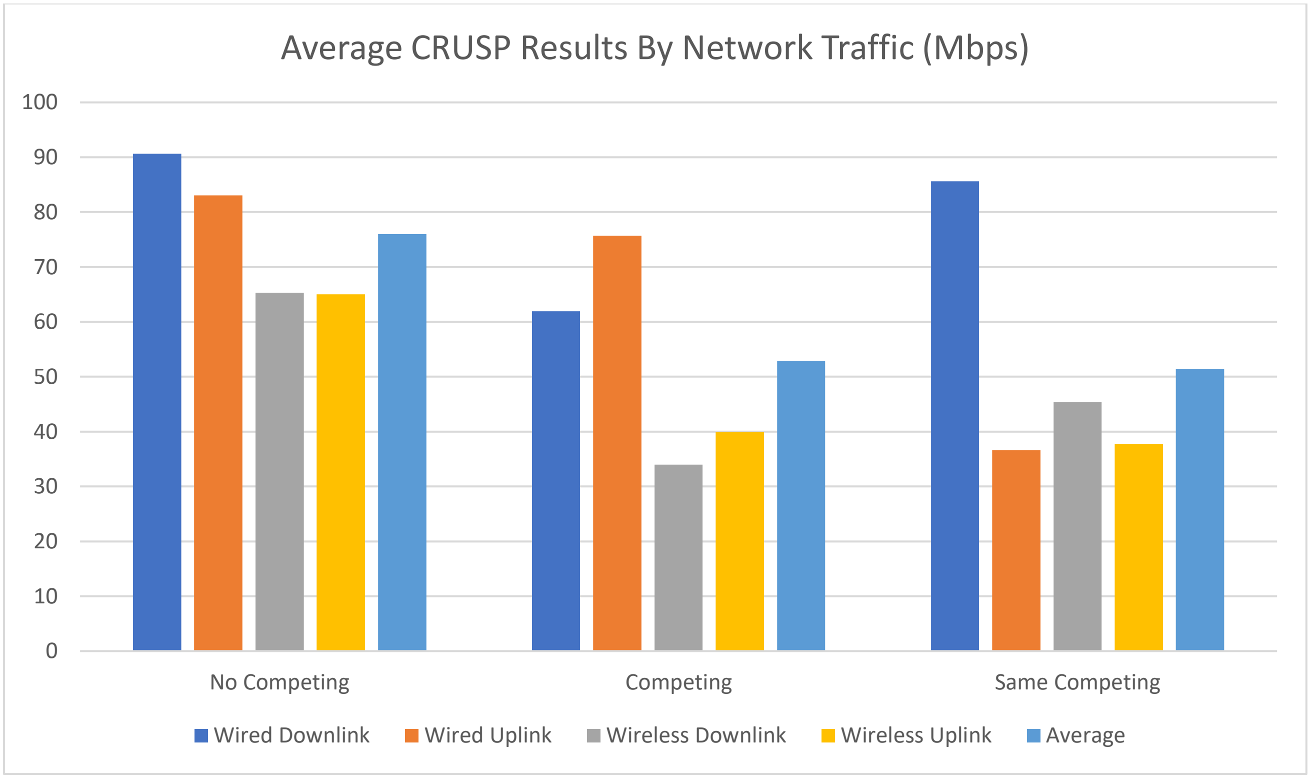


First, I want to note that in the classic wired, downlink setting, CRUSP estimated a 90.62 Mbps available bandwidth with no competing traffic – i.e. a maximum bandwidth estimation. We compare this with the iperf3 ground truths, tested in both the wired and wireless setup, which both returned 97.1 Mbps. This is a roughly 6.6% error, or we can say the estimate is ~93.4% accurate. A single CRUSP run is near-instantaneous, and transfers only in the realm of 900 packets with a default size of 1000 bytes – that's less than 1MB of data transfer! That's a very promising result considering our research problem.

The raw results can be a little hard to read through, so here are some summarizations:

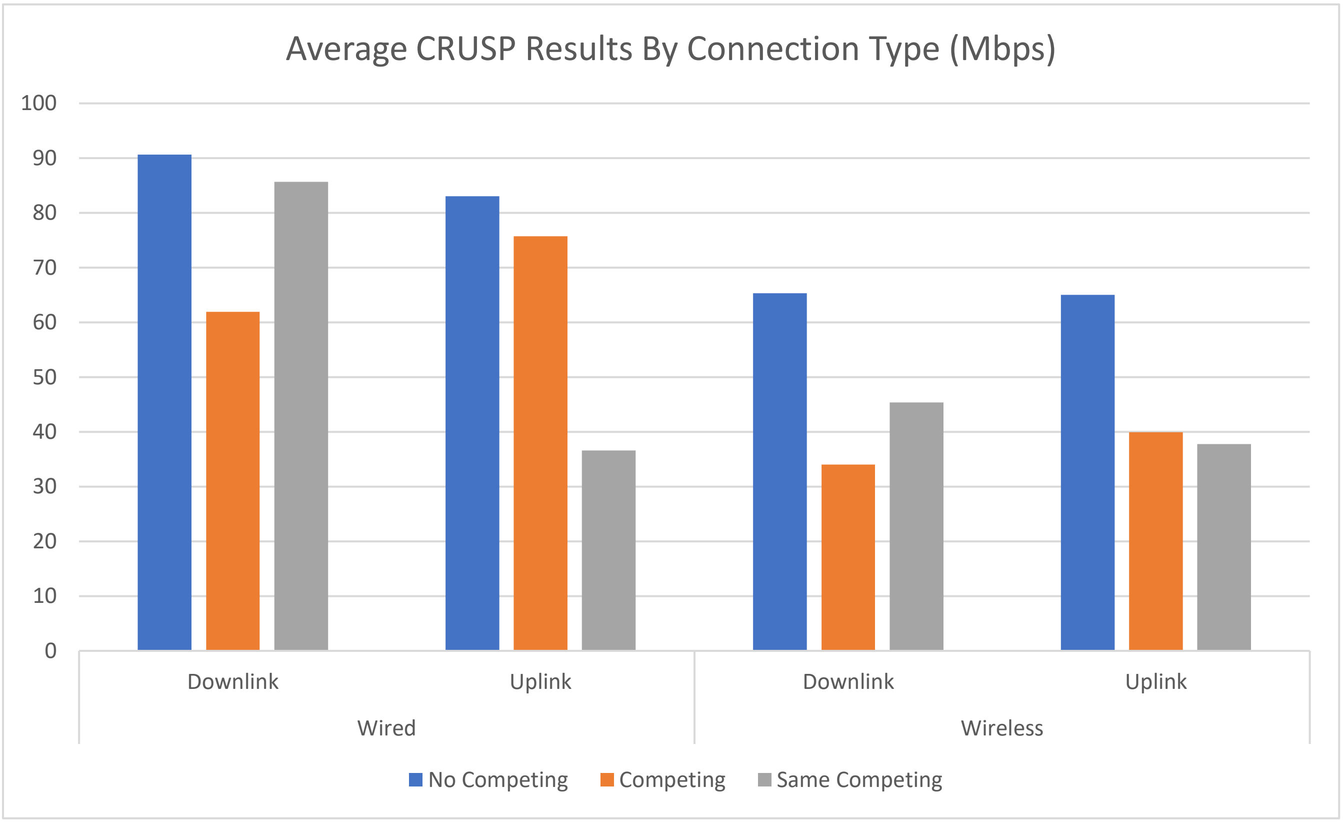
Average CRUSP Bandwidth Measurement Results (Mbps)

	Wired Downlink	*Wired Uplink*	*Wireless Downlink*	*Wireless Uplink*	***Average***
No Competing	90.62	83.04	65.33	65.01	76
Competing	61.95	75.7	34.0	39.94	52.8975
Same Competing	85.64	36.63	45.38	37.75	51.35



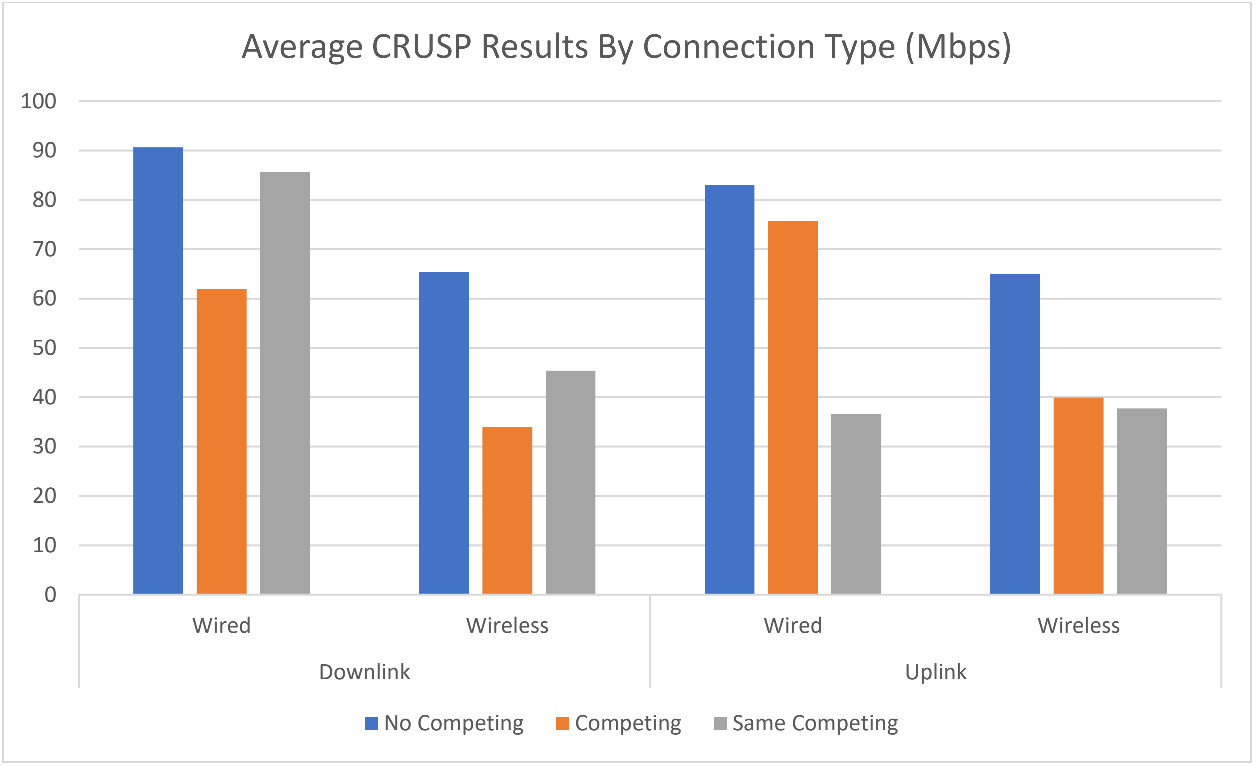
There’s still a lot to digest here, but broadly, we absolutely see that a massive factor, as expected, was competing network traffic. Without competing traffic, the estimated bandwidth across all scenarios was around 76 Mbps, compared with 52.8 and 51.3 on average in the scenarios with competing traffic.

We can also slice our data down other dimensions. Let’s try to compare wired vs wireless measurements now:



Once again, we see, as expected, a very significant difference between the wired and wireless setting. On average, wired bandwidth measurements came in at 72.26 Mbps, while wireless came in at just 47.9 Mbps, or just 66% the wired measurement. It’s well established that in reality WiFi supports lower throughput than ethernet, but considering the theoretical bandwidth per iperf3 remained 97.1 in the wireless setting, this should not be what we are encountering. Instead, I presume that our actual propagation delay is a significant portion of our overall RTT, which is the time delta these packet pair/train/burst methods use. This would explain why CRUSP experiences longer delay between packets – since the wireless link literally takes longer for packets to traverse than the ethernet wire. Hopefully, this would not be as much of an issue in real-world use cases, since the bottleneck should not be a local link.

Finally, let's slice down our last dimension: uplink vs downlink.



Downlink average: 63.82 Mbps

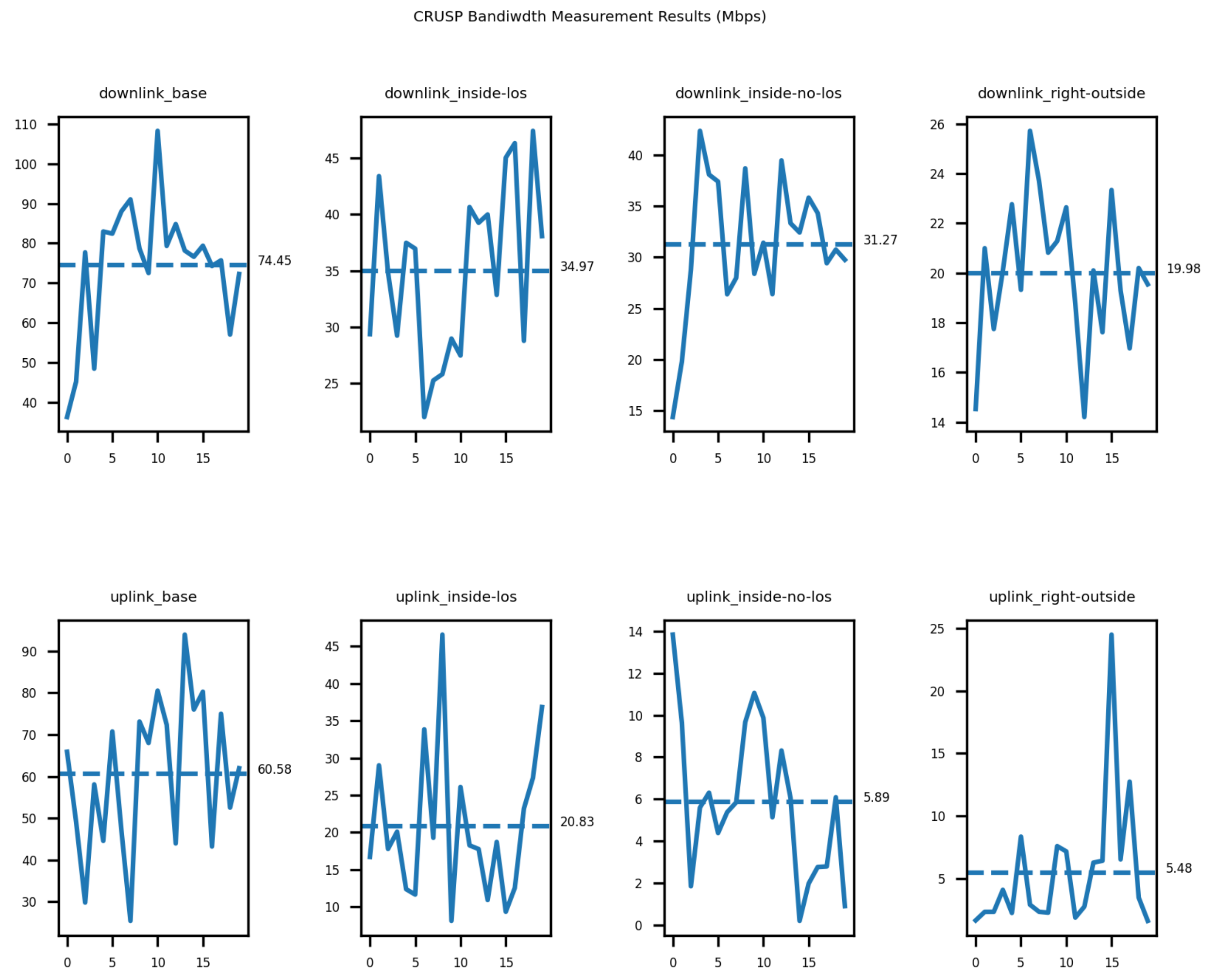
Uplink average: 56.35 Mbps

I'm not sure how to explain the difference for this one. Hypothetically, as I mentioned, the difference should be within the margin of error, but this is a little outside that range. Perhaps the uplink_wired_competing_same case is the issue, seeing as it came out to be less than half its downlink counterpart.

Conclusion:

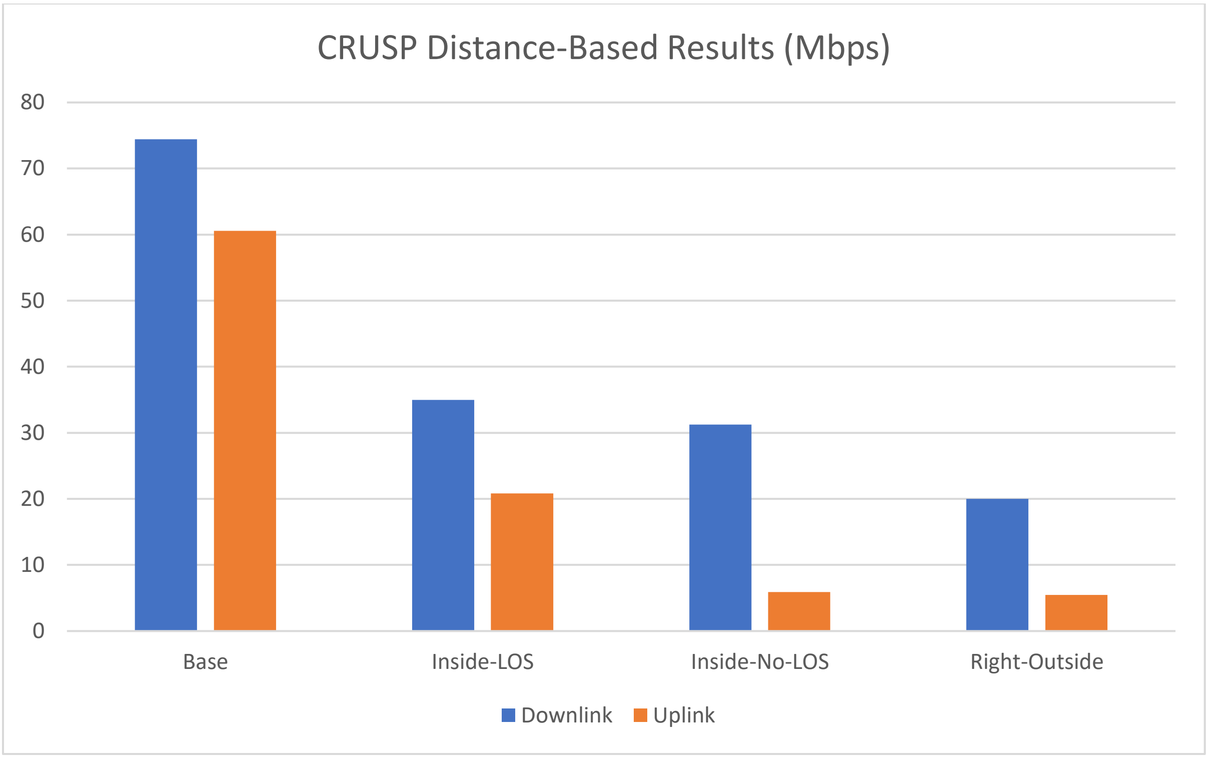
Our measurements went mostly as expected, and the speed and data volume with which CRUSP performed gives us some very promising results for justifying its use in the real world. However, there are some hiccups that need to be worked out next semester. Most importantly, we need to see if the underestimated bandwidth in the wireless setting continues when expanding past a local network. Less importantly, we can see why downlink vs uplink results are not as identical as they theoretically should be.

Wireless Distance-Based Measurements:



CRUSP Distance-Based Measurement Results Summary (Mbps):

	BASE	INSIDE-LOS	INSIDE-NO-LOS	RIGHT-OUTSIDE
DOWNLINK	74.453	34.966	31.27	19.984
UPLINK	60.584	20.828	5.885	5.476
IPERF3 RECEIVER (SERVER DOWNLINK)	42.6	12.9	2.46	3.69
IPERF3 SENDER (CLIENT UPLINK)	43.7	13.4	3.45	4.15

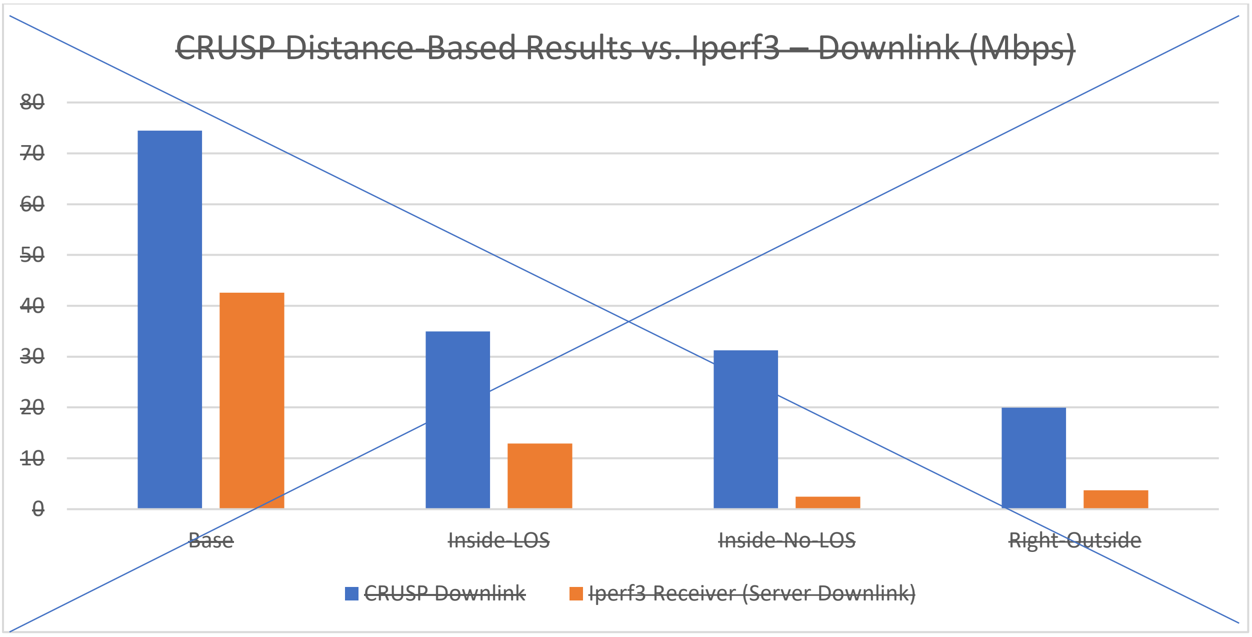


As we can see from the above, available bandwidth is decreasing as we expect as the client moves further away from the router and the server. We test the following scenarios, in all of which the server remains right next to the router:

1. Base – client is right next to router
2. Inside-LOS – client is inside lab about 15-20 feet away with a line of sight to router
3. Inside-No-LOS – client is inside lab about 20 feet away from router, behind a wall
4. Right-Outside – client is right outside lab, two walls away from router

To get a clearer picture as to how *accurate* these results are, we can compare to the iperf3 baselines:

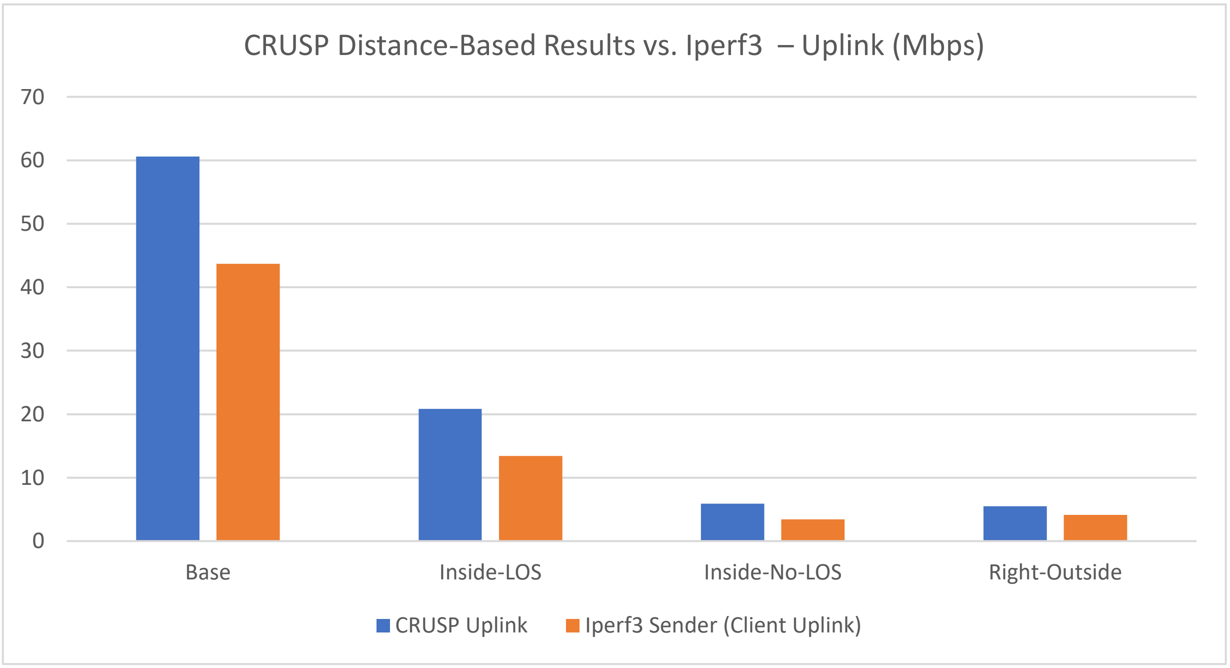
	~~**BASE**~~	~~**INSIDE-LOS**~~	~~**INSIDE-NO-LOS**~~	~~**RIGHT-OUTSIDE**~~
~~**CRUSP DOWNLINK**~~	~~74.453~~	~~34.966~~	~~31.27~~	~~19.984~~
~~**IPERF3 RECEIVER**~~ ~~(SERVER DOWNLINK)~~	~~42.6~~	~~12.9~~	~~2.46~~	~~3.69~~



IGNORE ABOVE – Iperf3 receiver means server bandwidth of downloading client packets, while CRUSP downlink means client bandwidth of downloading server packets.

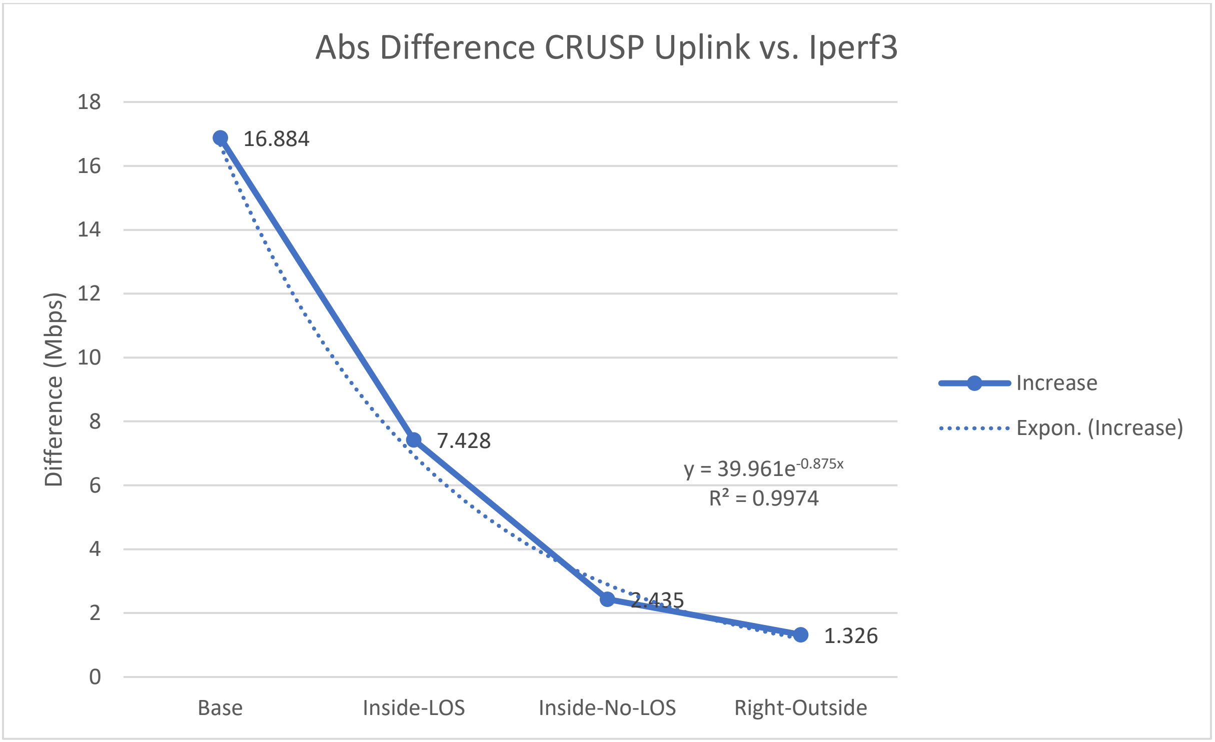
Let's take a look at the uplink case:

	BASE	INSIDE-LOS	INSIDE-NO-LOS	RIGHT-OUTSIDE
CRUSP UPLINK	60.584	20.828	5.885	5.476
IPERF3 SENDER (CLIENT UPLINK)	43.7	13.4	3.45	4.15



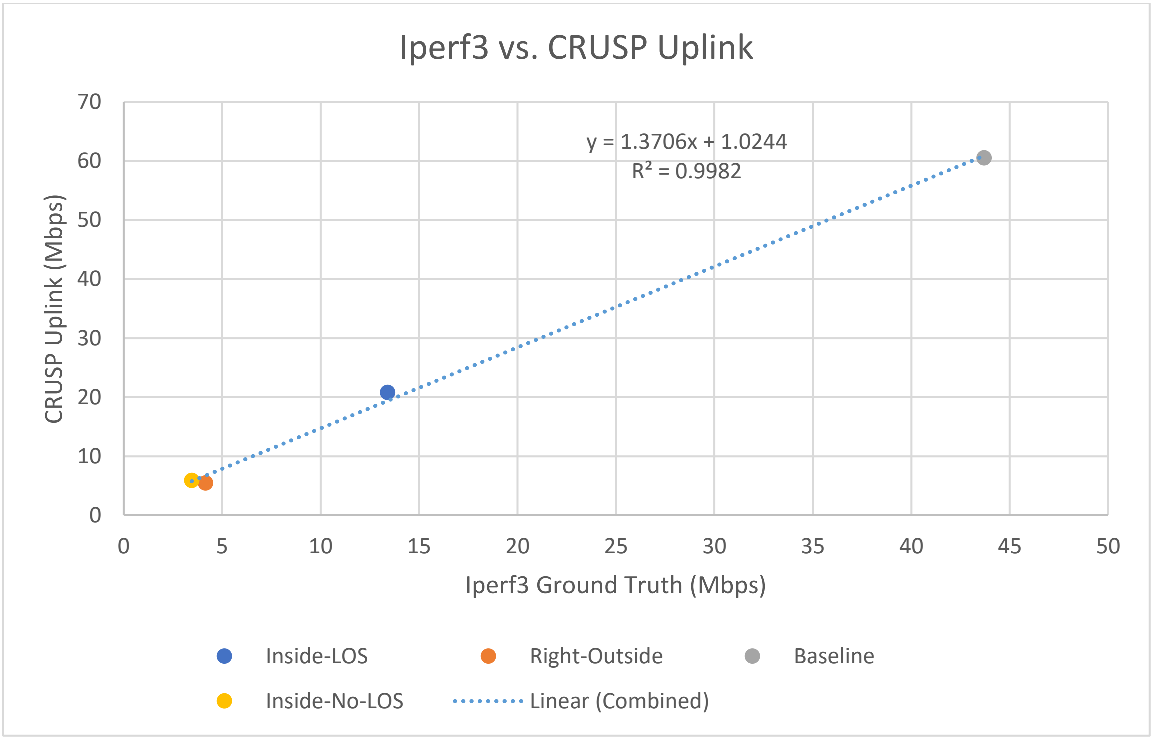
We can clearly see just glancing at the graph that CRUSP does a nice job remaining consistent with the ground truth here, always slightly overestimating the available bandwidth. Recall that for CRUSP, the uplink case refers to the bandwidth of the client sending packets to the server, and the downlink case refers to bandwidth of the client receiving packets from the server.

	Base	**Inside-LOS**	**Inside-No-LOS**	**Right-Outside**
Iperf3 Sender (Client Uplink)	43.7	13.4	3.45	4.15
CRUSP Uplink	60.584	20.828	5.885	5.476
% Increase	38.636	55.433	70.58	31.952

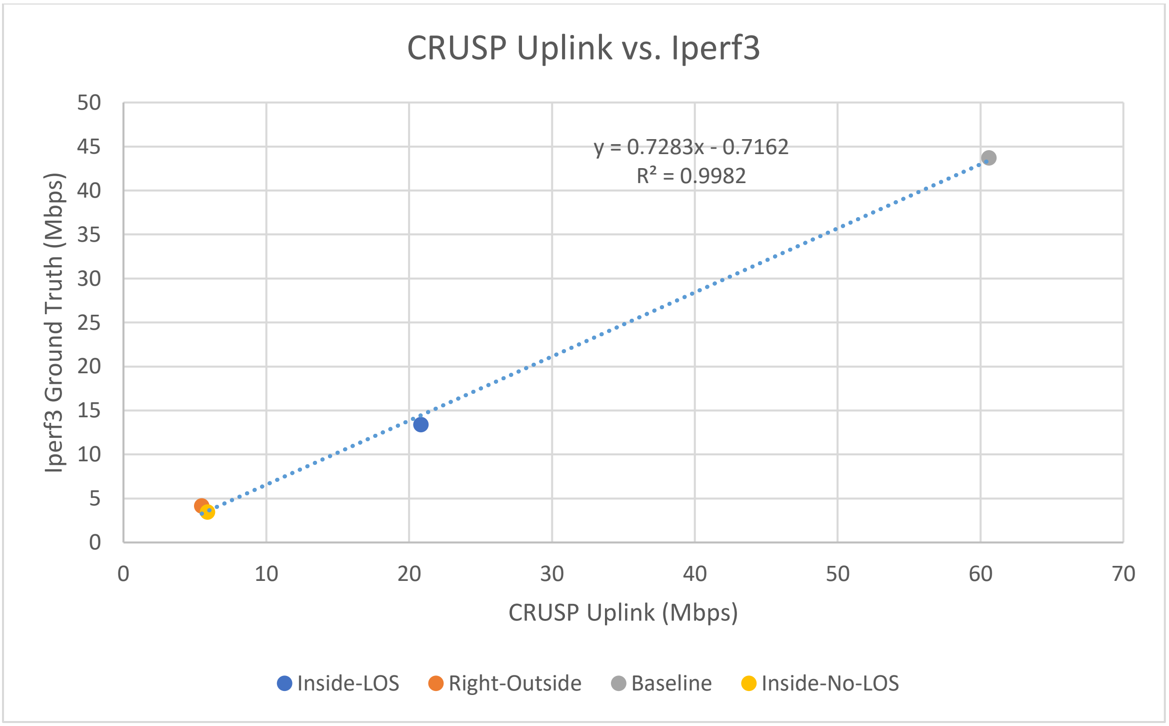


Although CRUSP uplink averages 49% overreporting of available bandwidth compared to iperf3 across these four experiments, if this overreporting is consistent, we can adjust for it mathematically. In other words, as long as the trend is strong, we can model CRUSP as an approximator of available bandwidth. From these four experiments, we can say an exponential curve models the function of the *difference* between CRUSP and iperf3.

Another, maybe more intuitive, way to look at this is by simply graphing iperf3 values on the x-axis and CRUSP values on the y-axis:



Now, we have what we want in a usable form: we can construct a linear best fit line that is well-explained, i.e. the coefficient of determination is very close to 1. Theoretically, we would be able to use this best fit line to predict the CRUSP output for available uplink bandwidth given iperf3 measurements as the regressors. In actuality, we want to determine the ground truth bandwidth from the CRUSP measurements.. let's switch the axes:



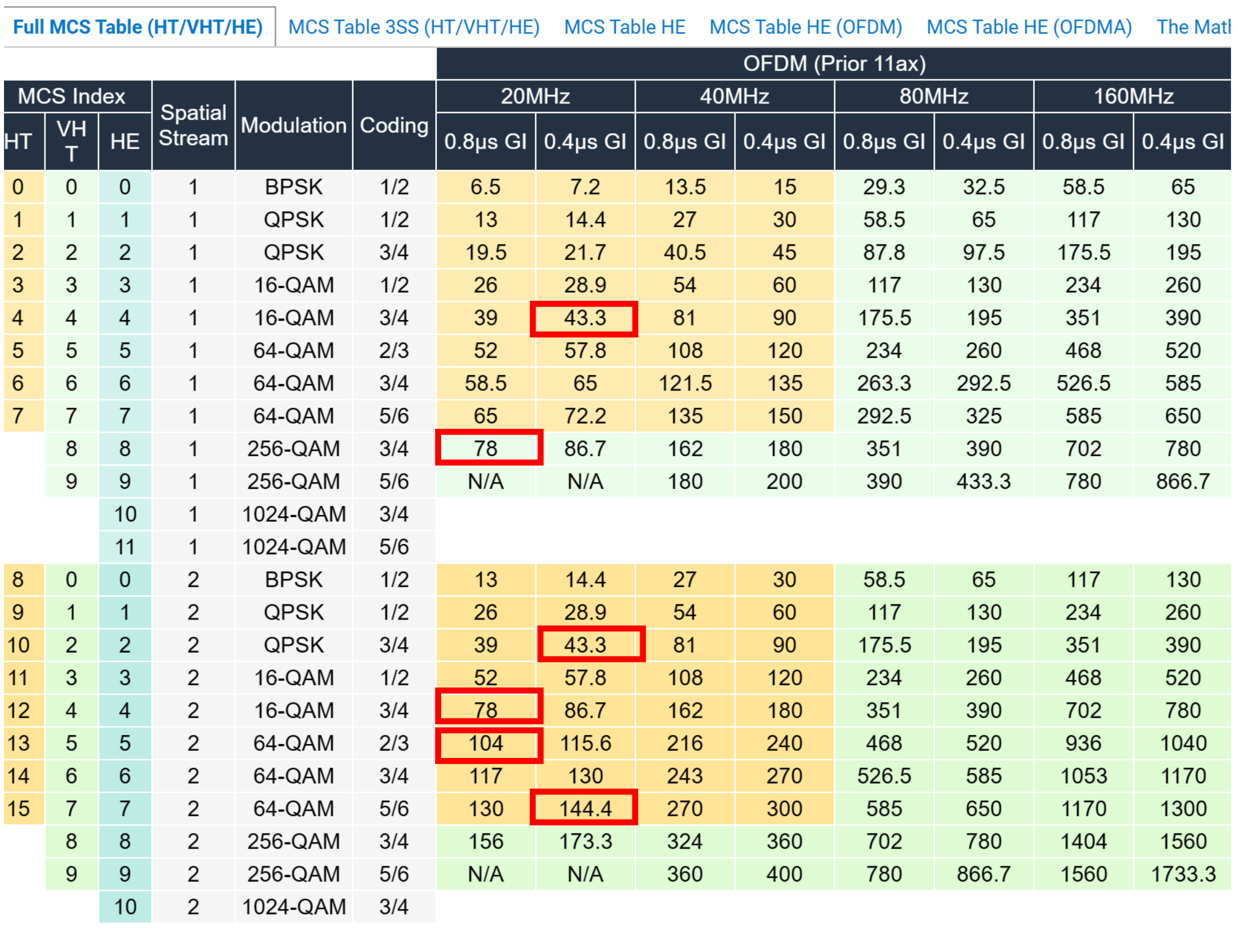
So finally, we have the following equation:

$$c \leftarrow CRUSP\ ABW$$

$$ABW_{ground\ truth} = 0.7283c - 0.7162$$

MCS Indices:

While I have effectively no knowledge of wireless signal processing, we can look at some basic metrics of our WIFI channel to further explain/contextualize our bandwidth measurements.


Full MCS Table (HT/VHT/HE) | MCS Table 3SS (HT/VHT/HE) | MCS Table HE | MCS Table HE (OFDM) | MCS Table HE (OFDMA) | The Matl

						OFDM (Prior 11ax)							
MCS Index			Spatial Stream	Modulation	Coding	20MHz		40MHz		80MHz		160MHz	
HT	VHT	HE				0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI
0	0	0	1	BPSK	1/2	6.5	7.2	13.5	15	29.3	32.5	58.5	65
1	1	1	1	QPSK	1/2	13	14.4	27	30	58.5	65	117	130
2	2	2	1	QPSK	3/4	19.5	21.7	40.5	45	87.8	97.5	175.5	195
3	3	3	1	16-QAM	1/2	26	28.9	54	60	117	130	234	260
4	4	4	1	16-QAM	3/4	39	43.3	81	90	175.5	195	351	390
5	5	5	1	64-QAM	2/3	52	57.8	108	120	234	260	468	520
6	6	6	1	64-QAM	3/4	58.5	65	121.5	135	263.3	292.5	526.5	585
7	7	7	1	64-QAM	5/6	65	72.2	135	150	292.5	325	585	650
	8	8	1	256-QAM	3/4	78	86.7	162	180	351	390	702	780
	9	9	1	256-QAM	5/6	N/A	N/A	180	200	390	433.3	780	866.7
		10	1	1024-QAM	3/4								
		11	1	1024-QAM	5/6								
8	0	0	2	BPSK	1/2	13	14.4	27	30	58.5	65	117	130
9	1	1	2	QPSK	1/2	26	28.9	54	60	117	130	234	260
10	2	2	2	QPSK	3/4	39	43.3	81	90	175.5	195	351	390
11	3	3	2	16-QAM	1/2	52	57.8	108	120	234	260	468	520
12	4	4	2	16-QAM	3/4	78	86.7	162	180	351	390	702	780
13	5	5	2	64-QAM	2/3	104	115.6	216	240	468	520	936	1040
14	6	6	2	64-QAM	3/4	117	130	243	270	526.5	585	1053	1170
15	7	7	2	64-QAM	5/6	130	144.4	270	300	585	650	1170	1300
	8	8	2	256-QAM	3/4	156	173.3	324	360	702	780	1404	1560
	9	9	2	256-QAM	5/6	N/A	N/A	360	400	780	866.7	1560	1733.3
		10	2	1024-QAM	3/4								

Using the Windows network properties interface, we find the following link speeds:

Base	Inside-LOS	Inside-No-LOS	Right-Outside
144	104	78	43

I highlight where these entries could be in the MCS table; in all likelihood, all of them come from the bottom 4 rectangles highlighted, all of which are in the section of 2 special streams instead of 1. The key differences as we move further away and further obstruct the signal between the router and the client is the coding decreases within the same modulation and/or the modulation changes itself. From my very brief research, the coding rate represents the amount of information transmitted that is actually useful (think goodput vs throughput). This seems to be mainly for the purposes of error detection. My theory, then, is that as the channel becomes more error prone, more information is wasted in redundant bits, even when using the same modulation.

However, we also move to different modulations altogether, from 64-QAM to 16-QAM to QPSK. The key difference in these modulations is the number of different states or phases their carrier waves can represent. In the case of 64-QAM, there are 64 distinct constellations, in 16-QAM, there are 16, and in QPSK, there are 4. These generally mean a relatively larger number of bits can be transmitted depending on the modulation.

Regression Validation

Given we now have a regressor, we should validate it using independent measurements – after all, we only used four data points to fit the model.

CRUSP Validation Distance-Based Measurements (Mbps):

	rtest1	rtest2	rtest3
CRUSP Uplink	27.87	25.06	31.97
Iperf3 Sender (Ground Truth)	17.2	16.8	32.7

Using our CRUSP uplink measurements, we can compute our predicted ground truth bandwidth using our linear regression formula ($ABW_{ground\ truth} = 0.7283c - 0.7162$):

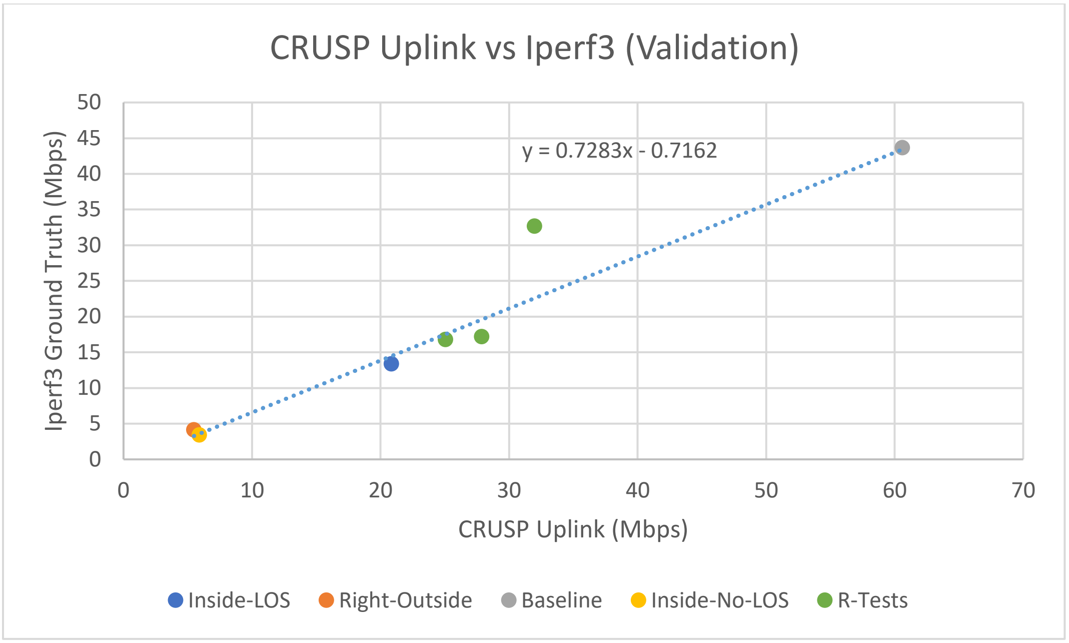
	rtest1	rtest2	rtest3
Regression Predicted	19.581521	17.534998	22.567551
Iperf3 Sender (Ground Truth)	17.2	16.8	32.7
% Difference	13.846052	4.3749881	30.986083

Here we can clearly see how far off our model is for these three scenarios. We can compute the MAPE (Mean Absolute Percentage Error) as a metric for our model's performance:

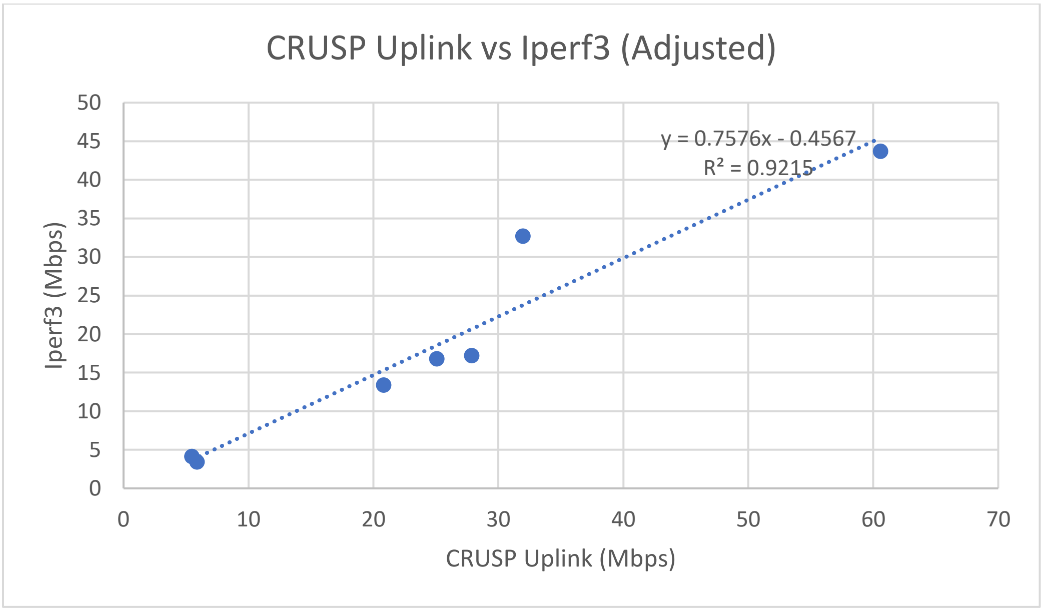
$$M = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{A_t - F_t}{A_t}\right|$$

$$\boxed{\boldsymbol{M = 16.40\%}}$$

Graphically, we can get an idea as to where this error is coming from:



The above graph shows our old best fit line with the validation/testing data in green. As we can see, most of the error comes from, ironically, CRUSP predicting more accurately than we expected; for rtest3, CRUSP was less than 5% off… it didn't actually follow the trend we use of downward adjusting by a factor of about 30%. It's hard to say why we see this outlier; only more testing can shine light on whether this is an isolated incident or it is indicative of something inherent in CRUSP.

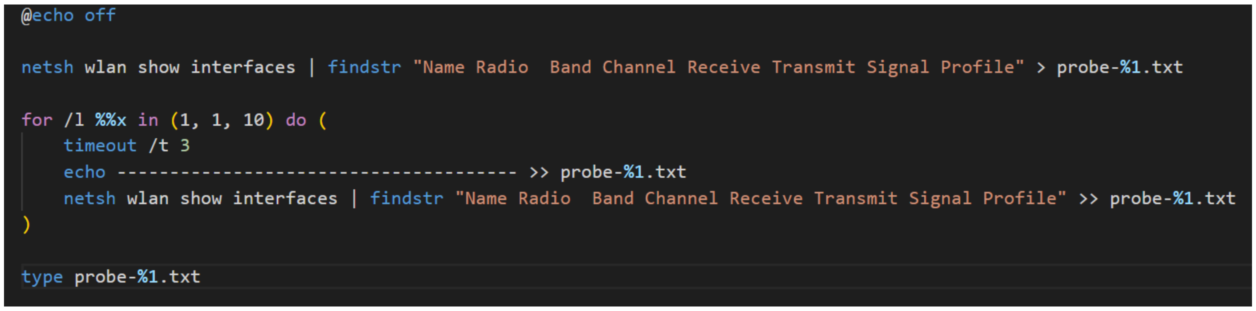


We could also use this new data to re-fit a linear regression model. As expected, almost entirely due to rtest3, the new model does less downward adjusting of CRUSP, multiplying by a factor of .7576 instead of .7283. Our R^2 does notably drop, but the correlation remains strong regardless.

Automated Link Probing

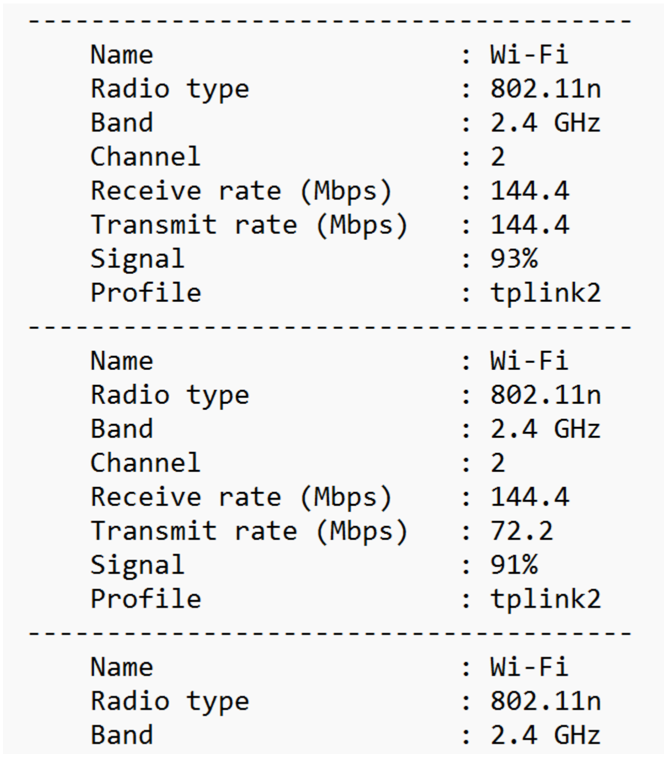
In an effort to gleam information about how the link and channel state changes over measurement periods, I created a batch script that calls Windows' netsh utility at regular intervals:

network_probe.bat



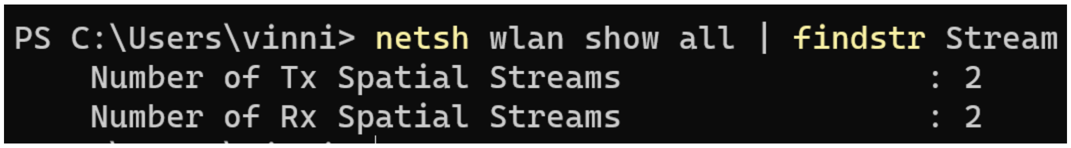
```
@echo off

netsh wlan show interfaces | findstr "Name Radio  Band Channel Receive Transmit Signal Profile" > probe-%1.txt

for /l %%x in (1, 1, 10) do (
    timeout /t 3
    echo -------------------------------------- >> probe-%1.txt
    netsh wlan show interfaces | findstr "Name Radio  Band Channel Receive Transmit Signal Profile" >> probe-%1.txt
)

type probe-%1.txt
```

Sample output:



```
 ---------------------------------------
    Name                   : Wi-Fi
    Radio type             : 802.11n
    Band                   : 2.4 GHz
    Channel                : 2
    Receive rate (Mbps)    : 144.4
    Transmit rate (Mbps)   : 144.4
    Signal                 : 93%
    Profile                : tplink2
 ---------------------------------------
    Name                   : Wi-Fi
    Radio type             : 802.11n
    Band                   : 2.4 GHz
    Channel                : 2
    Receive rate (Mbps)    : 144.4
    Transmit rate (Mbps)   : 72.2
    Signal                 : 91%
    Profile                : tplink2
 ---------------------------------------
    Name                   : Wi-Fi
    Radio type             : 802.11n
    Band                   : 2.4 GHz
```

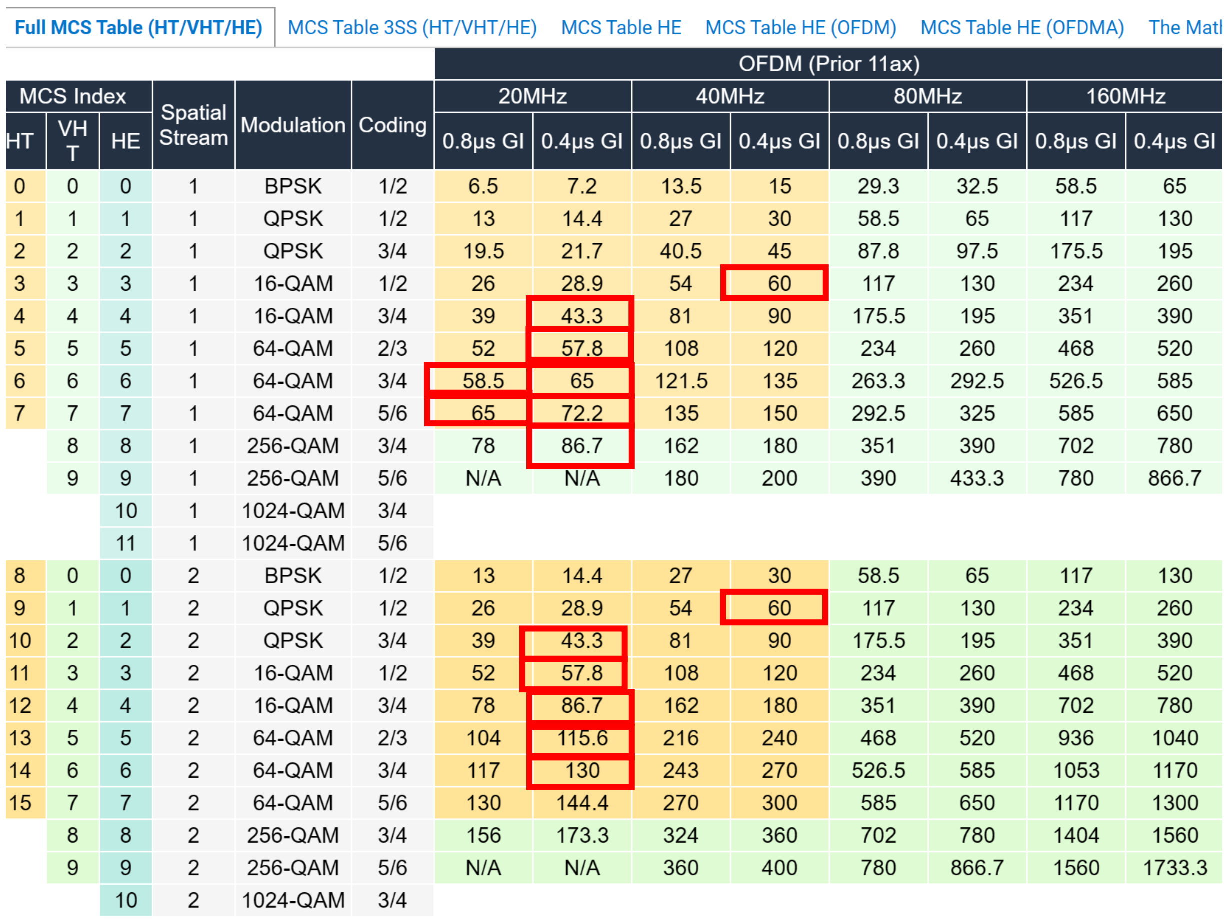
netsh also allows us to check the number of Spatial Streams our network device is using.



```
PS C:\Users\vinni> netsh wlan show all | findstr Stream
    Number of Tx Spatial Streams                 : 2
    Number of Rx Spatial Streams                 : 2
```

Transmit Rate Over Time (Mbps)

rtest1	rtest2	rtest3
60	43.3	72.2
86.7	43.3	72.2
86.7	65	72.2
72.2	65	130
72.2	65	130
60	72.2	86.7
72.2	58.5	86.7
72.2	65	86.7
72.2	115.6	115.6
72.2	57.8	115.6


Full MCS Table (HT/VHT/HE)
MCS Table 3SS (HT/VHT/HE)
MCS Table HE
MCS Table HE (OFDM)
MCS Table HE (OFDMA)
The Matl

						OFDM (Prior 11ax)							
MCS Index			Spatial Stream	Modulation	Coding	20MHz		40MHz		80MHz		160MHz	
HT	VHT	HE				0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI	0.8µs GI	0.4µs GI
0	0	0	1	BPSK	1/2	6.5	7.2	13.5	15	29.3	32.5	58.5	65
1	1	1	1	QPSK	1/2	13	14.4	27	30	58.5	65	117	130
2	2	2	1	QPSK	3/4	19.5	21.7	40.5	45	87.8	97.5	175.5	195
3	3	3	1	16-QAM	1/2	26	28.9	54	60	117	130	234	260
4	4	4	1	16-QAM	3/4	39	43.3	81	90	175.5	195	351	390
5	5	5	1	64-QAM	2/3	52	57.8	108	120	234	260	468	520
6	6	6	1	64-QAM	3/4	58.5	65	121.5	135	263.3	292.5	526.5	585
7	7	7	1	64-QAM	5/6	65	72.2	135	150	292.5	325	585	650
	8	8	1	256-QAM	3/4	78	86.7	162	180	351	390	702	780
	9	9	1	256-QAM	5/6	N/A	N/A	180	200	390	433.3	780	866.7
		10	1	1024-QAM	3/4								
		11	1	1024-QAM	5/6								
8	0	0	2	BPSK	1/2	13	14.4	27	30	58.5	65	117	130
9	1	1	2	QPSK	1/2	26	28.9	54	60	117	130	234	260
10	2	2	2	QPSK	3/4	39	43.3	81	90	175.5	195	351	390
11	3	3	2	16-QAM	1/2	52	57.8	108	120	234	260	468	520
12	4	4	2	16-QAM	3/4	78	86.7	162	180	351	390	702	780
13	5	5	2	64-QAM	2/3	104	115.6	216	240	468	520	936	1040
14	6	6	2	64-QAM	3/4	117	130	243	270	526.5	585	1053	1170
15	7	7	2	64-QAM	5/6	130	144.4	270	300	585	650	1170	1300
	8	8	2	256-QAM	3/4	156	173.3	324	360	702	780	1404	1560
	9	9	2	256-QAM	5/6	N/A	N/A	360	400	780	866.7	1560	1733.3
		10	2	1024-QAM	3/4								

Note how in the above MCS table there are often two possible options for which modulation and coding scheme, depending on how many spatial streams are being used. Also note that we are only concerned with 20MHZ and 40MHZ channel width, as that is all that is supported by the tplink2 router.

Let's first fill in the unique entries in the table that we *know* for sure: 58.5, 72.2, 115.6, and 130.

Modulation and Coding Scheme (MCS) Over Time (Known)

Streams, Modulation, Coding, Channel Width, Guard Interval

rtest1	rtest2	rtest3
60	43.3	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
86.7	43.3	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
86.7	65	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	65	2 streams, 64-QAM, 3/4, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	65	2 streams, 64-QAM, 3/4, 20MHZ, .4 us
60	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	86.7
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, ¾, 20 MHZ, .8 us	86.7
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	65	86.7
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	2 streams, 64-QAM, 2/3, 20MHZ, .4 us	2 streams, 64-QAM, 2/3, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	57.8	2 streams, 64-QAM, 2/3, 20MHZ, .4 us

Now, for the rest of the entries, we have to make a few inferences…

1. For 86.7 Mbps, note how the entry using 1 spatial stream uses a 256-QAM modulation. However, we can see from our sample netsh output above that we are using the 802.11n radio specification, which supports only up to 64-QAM modulation.
   a. https://en.wikipedia.org/wiki/IEEE_802.11n-2009
   b. What's New in 802.1X Authenticated Wireless Access | Microsoft Learn

   Therefore, we can be sure any entries listing 86.7 Mbps are using 2 spatial streams.
2. For 57.8 Mbps, in reality this could be 1 spatial stream or 2, but we note that the timeslice before it was exactly double (115.6)… thus it would make sense that the connection dropped from 2 to 1 streams instead of modulation changing dramatically.

3. The existence of 43.3 and 65 together in rtest2 might imply that we are in the 1 spatial stream section of the table; it'd be less likely that all of streams, modulation, and coding changed as opposed to just modulation (16-QAM to 64-QAM). This settles 43.3 in HT 4. By the same logic, we could assume the 65 entry is HT 6 instead of 7.
4. We really can't be sure about 60 Mbps. We will mark entries we are particularly unsure about with a ?

Modulation and Coding Scheme (MCS) Over Time (Inferred)

Streams, Modulation, Coding, Channel Width, Guard Interval

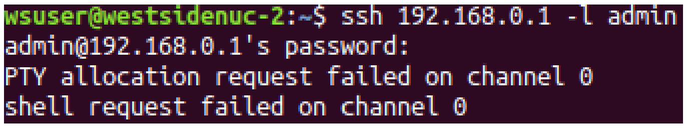
rtest1	rtest2	rtest3
60 ?	1 stream, 16-QAM, ¾, 20MHZ, .4 us	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
2 streams, 16-QAM, ¾, 20MHZ, .4 us	1 stream, 16-QAM, ¾, 20MHZ, .4 us	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
2 streams, 16-QAM, ¾, 20MHZ, .4 us	1 stream, 64-QAM, ¾, 20MHZ, .4 us	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, ¾, 20MHZ, .4 us	2 streams, 64-QAM, 3/4, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, ¾, 20MHZ, .4 us	2 streams, 64-QAM, 3/4, 20MHZ, .4 us
60 ?	1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	2 streams, 16-QAM, ¾, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, ¾, 20 MHZ, .8 us	2 streams, 16-QAM, ¾, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, ¾, 20MHZ, .4 us	2 streams, 16-QAM, ¾, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	2 streams, 64-QAM, 2/3, 20MHZ, .4 us	2 streams, 64-QAM, 2/3, 20MHZ, .4 us
1 stream, 64-QAM, 5/6, 20 MHZ, .4 us	1 stream, 64-QAM, 2/3, 20MHZ, .4 us	2 streams, 64-QAM, 2/3, 20MHZ, .4 us

From these results, it's no surprise that rtest3 had the highest ABW, as it most consistently utilized 2 spatial streams over 1 – essentially meaning two independent data streams could be communicated simultaneously. And, though the difference between rtest1 and rtest2 may not quite be significant enough to make any conclusions, we can see, broadly, that the 5/6 coding rate was more common in rtest1 and ¾ moreso in rtest2, which very well may explain the slightly higher ABW in rtest1 than rtest2.

Manipulating MCS Index with iw + Flashing with OpenWRT

We've explored how MCS changes over the length of a measurement period and speculated as to how it relates to the measured ABW. Now, we want to fix this aspect of the connection – namely the channel's modulation scheme, coding rate, number of streams, and data width – and measure how it affects our ABW measurements. To do so, we will use the Linux iw command: en:users:documentation:iw [Linux Wireless] (kernel.org)

Note: the proprietary tp-link2 router did not allow configuration of MCS through its GUI nor allowed terminal/ssh access; as such I was forced to flash the firmware with OpenWrt.



```
wsuser@westsidenuc-2:~$ ssh 192.168.0.1 -l admin
admin@192.168.0.1's password:
PTY allocation request failed on channel 0
shell request failed on channel 0
```

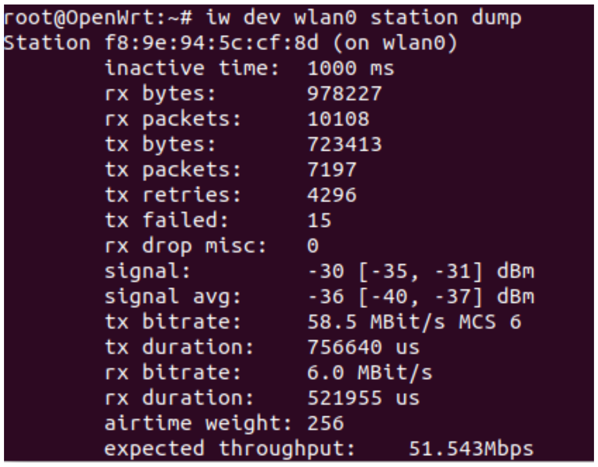
Please check the below OpenWrt links for information on which version the tplink was flashed with:

- [OpenWrt Wiki] Techdata: TP-Link Archer A6 v3
- [OpenWrt Wiki] TP-Link Archer C6 v3

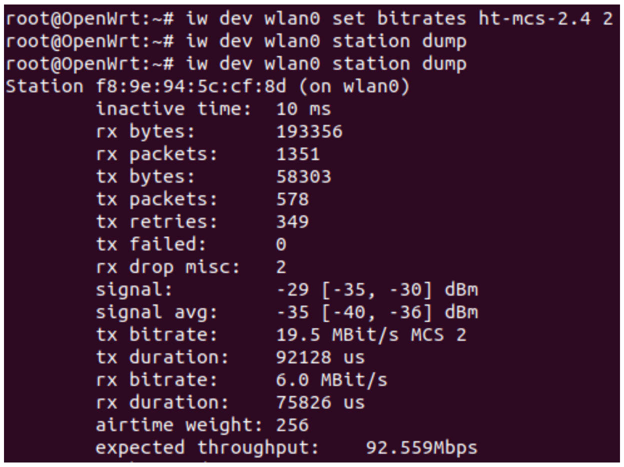
A couple notes: I separated the network into two SSIDs: OpenWrt-802.11n and OpenWrt-802.11ac for each respective radio. I set the password to each as what it used to be - `epltest123` - which is also the password for the router interface page at 192.168.1.1 or when logging in through ssh for root@192.168.1.1.

I.e. ssh access would look like ssh 192.168.1.1 -l root then provide the password when prompted.

Now, we can use the iw command to query and set aspects of our network devices/interfaces – most importantly for us being MCS index.



```
root@OpenWrt:~# iw dev wlan0 station dump
Station f8:9e:94:5c:cf:8d (on wlan0)
        inactive time:  1000 ms
        rx bytes:       978227
        rx packets:     10108
        tx bytes:       723413
        tx packets:     7197
        tx retries:     4296
        tx failed:      15
        rx drop misc:   0
        signal:         -30 [-35, -31] dBm
        signal avg:     -36 [-40, -37] dBm
        tx bitrate:     58.5 MBit/s MCS 6
        tx duration:    756640 us
        rx bitrate:     6.0 MBit/s
        rx duration:    521955 us
        airtime weight: 256
        expected throughput:    51.543Mbps
```



```
root@OpenWrt:~# iw dev wlan0 set bitrates ht-mcs-2.4 2
root@OpenWrt:~# iw dev wlan0 station dump
root@OpenWrt:~# iw dev wlan0 station dump
Station f8:9e:94:5c:cf:8d (on wlan0)
        inactive time:  10 ms
        rx bytes:       193356
        rx packets:     1351
        tx bytes:       58303
        tx packets:     578
        tx retries:     349
        tx failed:      0
        rx drop misc:   2
        signal:         -29 [-35, -30] dBm
        signal avg:     -35 [-40, -36] dBm
        tx bitrate:     19.5 MBit/s MCS 2
        tx duration:    92128 us
        rx bitrate:     6.0 MBit/s
        rx duration:    75826 us
        airtime weight: 256
        expected throughput:    92.559Mbps
```

Example setting MCS using iw set bitrates

Automated MCS Inference on Windows

So, we've now completed the prerequisite work for being able to manipulate aspects of the network from the router itself. In order to validate that these mutations were actually being reflected on devices connected to the network, I wanted to check that my Windows machine, which I am using as the CRUSP client, was communicating with the router using the correct Modulation, Coding, # Spatial Streams, etc. Virtual Machines have issues directly accessing metadata about the Windows/Host OS network device, so using iw was not a possibility. Even with the network_probe.bat script I describe above, it's very tedious to manually find all possible MCS values. I also could not find a Windows network tool that would quickly and easily give me the network metadata I wanted. So, I took it upon myself to create a script that would do the brunt of the work for me.

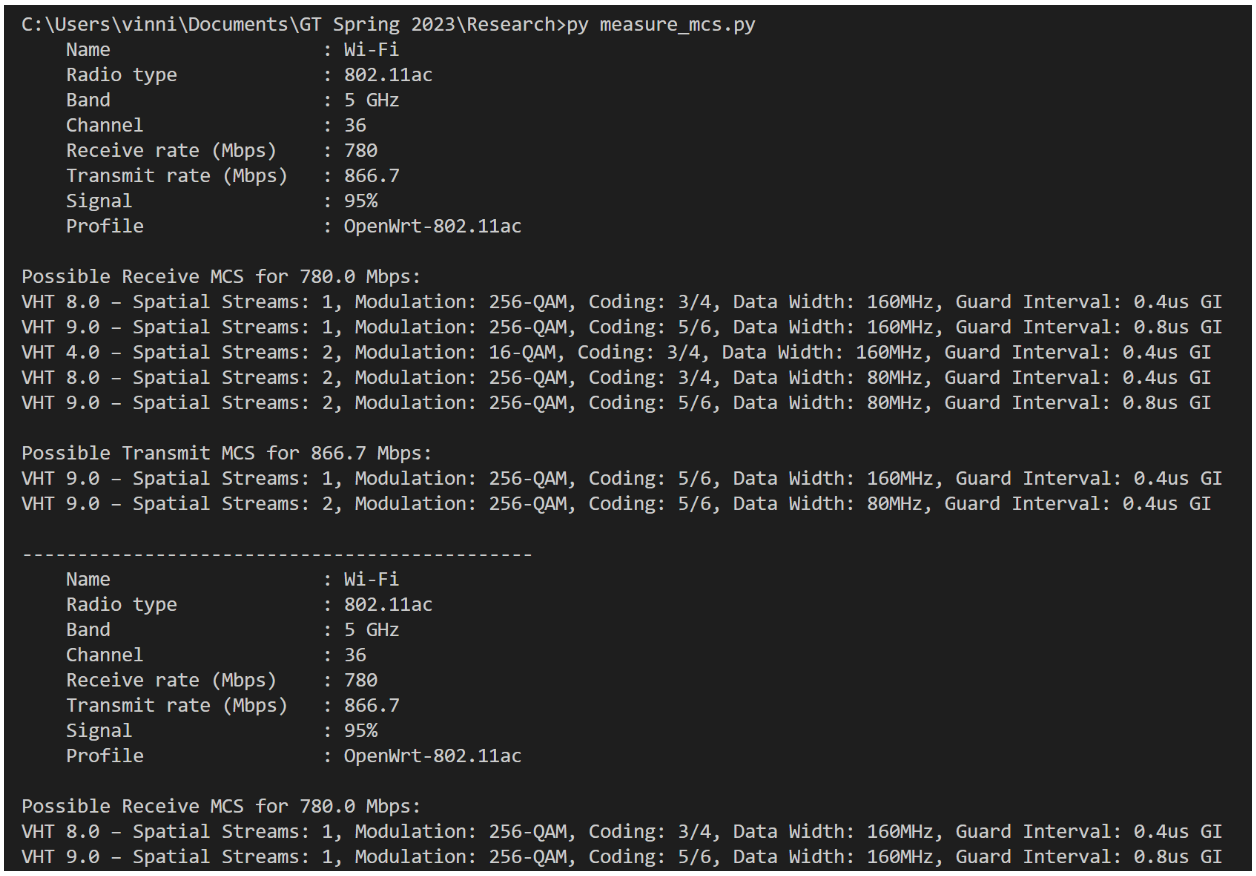
First, I created a Python script map_mcs_table.py that effectively translates the MCS table into a Pandas DataFrame, then creates a map of bandwidth rate to possible MCS configuration. The resulting map is stored in JSON, in possible_mcs_map.json, so that this script does not need to be re-run; instead, it is meant to be loaded from into a Python dict for use. The map_mcs_table script also creates an mcs_index_map.json file to map 3-tuples of # streams, modulation, and coding to HT/VHT/HE indices. Finally, I created measure_mcs.py, which calls network_probe.bat to obtain link information and lists the possible receive and transmit possible MCS indices in real-time from the Windows-reported link rates.

A couple notes about the scripts:

1. The JSON files I created only contain MCS table entries **for 1 or 2 spatial streams.** You need to update map_mcs_table.py if you want to use the entire table or a different subset.
2. You can play around with the VERBOSE and DEBUG constants in measure_mcs.py, but do note some functionality may not be complete. Ideally just leave the script as is.
3. If you don't need so many timesteps of measurements (1 may be sufficient), just update the for loop in network_probe.bat.

Again, if you want to use the automated MCS monitoring for Windows, it should suffice to simply run measure_mcs.py.

Sample output:



```
C:\Users\vinni\Documents\GT Spring 2023\Research>py measure_mcs.py
    Name                   : Wi-Fi
    Radio type             : 802.11ac
    Band                   : 5 GHz
    Channel                : 36
    Receive rate (Mbps)    : 780
    Transmit rate (Mbps)   : 866.7
    Signal                 : 95%
    Profile                : OpenWrt-802.11ac

Possible Receive MCS for 780.0 Mbps:
VHT 8.0 – Spatial Streams: 1, Modulation: 256-QAM, Coding: 3/4, Data Width: 160MHz, Guard Interval: 0.4us GI
VHT 9.0 – Spatial Streams: 1, Modulation: 256-QAM, Coding: 5/6, Data Width: 160MHz, Guard Interval: 0.8us GI
VHT 4.0 – Spatial Streams: 2, Modulation: 16-QAM, Coding: 3/4, Data Width: 160MHz, Guard Interval: 0.4us GI
VHT 8.0 – Spatial Streams: 2, Modulation: 256-QAM, Coding: 3/4, Data Width: 80MHz, Guard Interval: 0.4us GI
VHT 9.0 – Spatial Streams: 2, Modulation: 256-QAM, Coding: 5/6, Data Width: 80MHz, Guard Interval: 0.8us GI

Possible Transmit MCS for 866.7 Mbps:
VHT 9.0 – Spatial Streams: 1, Modulation: 256-QAM, Coding: 5/6, Data Width: 160MHz, Guard Interval: 0.4us GI
VHT 9.0 – Spatial Streams: 2, Modulation: 256-QAM, Coding: 5/6, Data Width: 80MHz, Guard Interval: 0.4us GI

----------------------------------------------
    Name                   : Wi-Fi
    Radio type             : 802.11ac
    Band                   : 5 GHz
    Channel                : 36
    Receive rate (Mbps)    : 780
    Transmit rate (Mbps)   : 866.7
    Signal                 : 95%
    Profile                : OpenWrt-802.11ac

Possible Receive MCS for 780.0 Mbps:
VHT 8.0 – Spatial Streams: 1, Modulation: 256-QAM, Coding: 3/4, Data Width: 160MHz, Guard Interval: 0.4us GI
VHT 9.0 – Spatial Streams: 1, Modulation: 256-QAM, Coding: 5/6, Data Width: 160MHz, Guard Interval: 0.8us GI
```

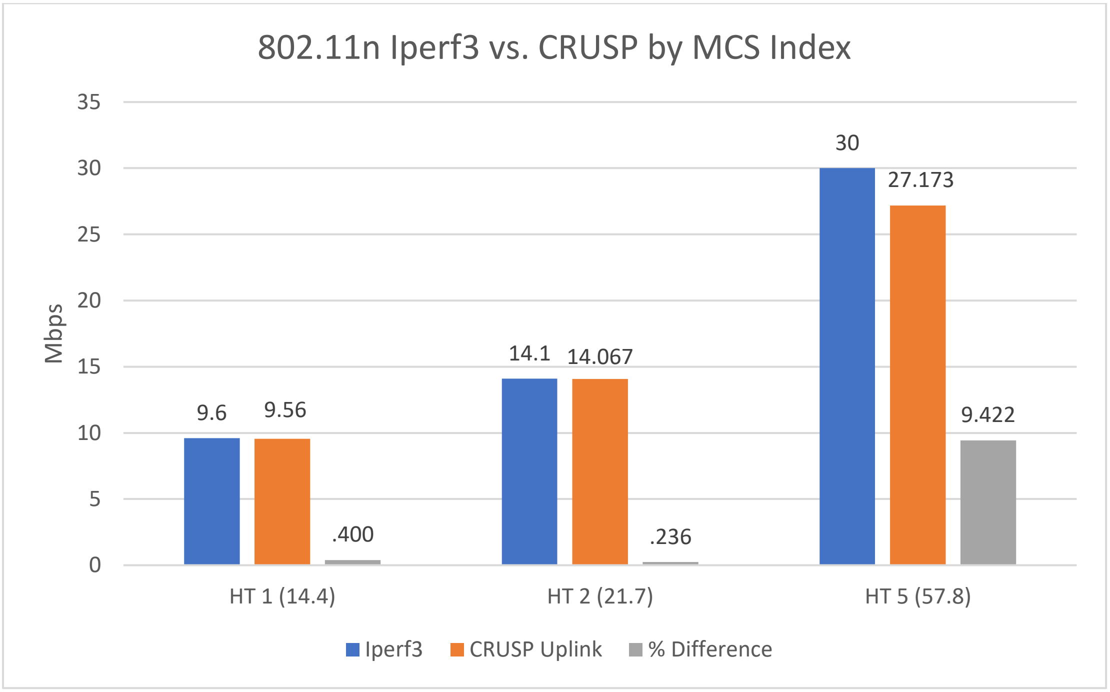
MCS-Based Experiments

Using the iw command, I left the client and server in the same physical location – i.e. keeping the independent variable of distance between/traveled fixed – and changed the MCS index on the router for both the 802.11n and 802.11ac network channels. With the MCS fixed, I ran workloads of 100 CRUSP measurements, dumping to an appropriate output file, and averaged the resulting available bandwidth.

For example, for an 802.11n test, I would ssh into the router and run iw wlan0 set bitrates ht-mcs-2.4 <index>, then run CRUSP/measurement_client_standalone/run_experiments.sh to have the client call the service and append raw output to appropriate text files.

I modified analysis.py to compile/average statistics about the experiment runs.

MCS	Iperf3	CRUSP Uplink	% Difference
HT 1 (14.4)	9.6	9.56150945	0.400943229
HT 2 (21.7)	14.1	14.06674464	0.235853617
HT 5 (57.8)	30	27.17343616	9.421879467

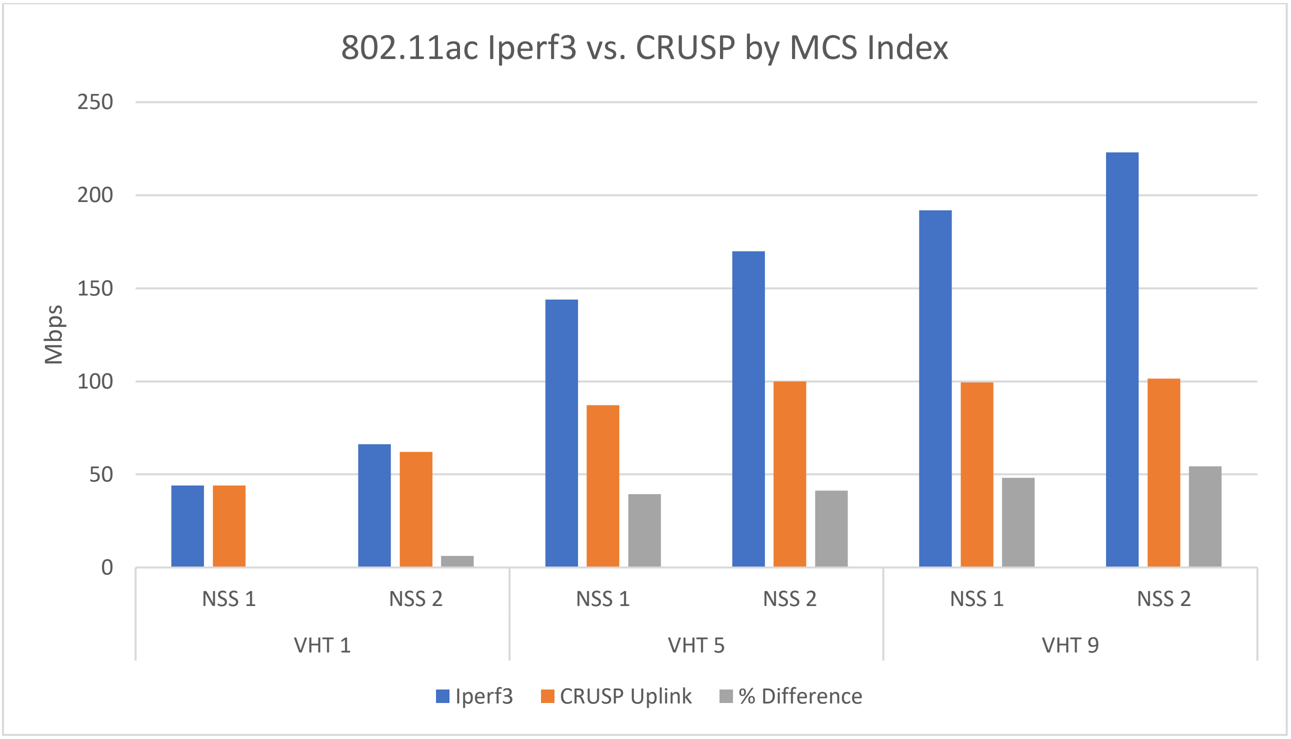


MAPE (Mean Absolute Percent Error) = 3.35%

Above, we can see the summary statistics for the 802.11n experiments. We can clearly see that for smaller MCS values, CRUSP did an excellent job of accurately predicting available bandwidth. Even for a more reasonable/real-world value of an MCS of 5 with 57.8 Mbps theoretical maximum bandwidth, we only differed by < 10%. Our MAPE of 3.35% is a significant improvement over our distance-based result of ~16%.

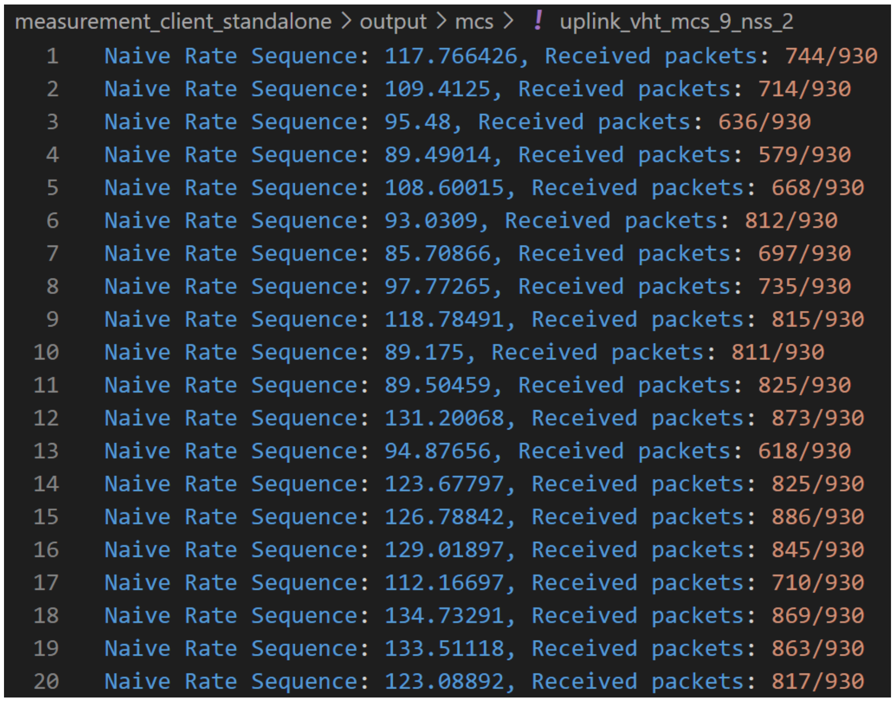
We should notice that this result is consistent with our distance-based measurement results, in that we noticed an inverse exponential relationship between the absolute difference of CRUSP vs. the Iperf3 ground truth and the distance of the client from the router + server. That is, at closer distances, when higher bandwidths were expected, CRUSP tended to underreport/underestimate by more than at further distances with lower expected bandwidth. These MCS results further contextualize that – the root cause of this discrepancy is likely that for lower MCS indices, which are preferred when a connection is more error-prone as earlier discussed, CRUSP seems to be much more accurate. For reasons that may be outside the scope of this report, at higher expected bandwidth, with more complex modulation schemes and higher coding ratio, CRUSP seems to underreport bandwidth.

MCS	Spatial Streams	Iperf3	CRUSP Uplink	% Difference
VHT 1	NSS 1	44	44.11125567	0.252853795
	NSS 2	66.3	62.1950452	6.19148537
VHT 5	NSS 1	144	87.25569918	39.40576446
	NSS 2	170	99.87070667	41.25252549
VHT 9	NSS 1	192	99.46719667	48.1941684
	NSS 2	223	101.5766731	54.44992236



As we expect, then, 802.11ac proves harder for CRUSP to predict accurately, as it supports higher bandwidths. The pattern of lower MCS indices being easier for CRUSP to handle and higher causing consistent underreporting of ABW is shown once again, only even more dramatically than in 802.11n. For VHT-1, we have just about the same ~3% MAPE, but overall, our MAPE = 31.6% (not meant to compare directly to 802.11n due to differing MCS indices tested).

What we discovered particularly in the 802.11ac testing is that CRUSP has a sort of asymptotic limit of around 100 Mbps that it will report, on average. Of course, individual data points within that 100 can report higher; see here:



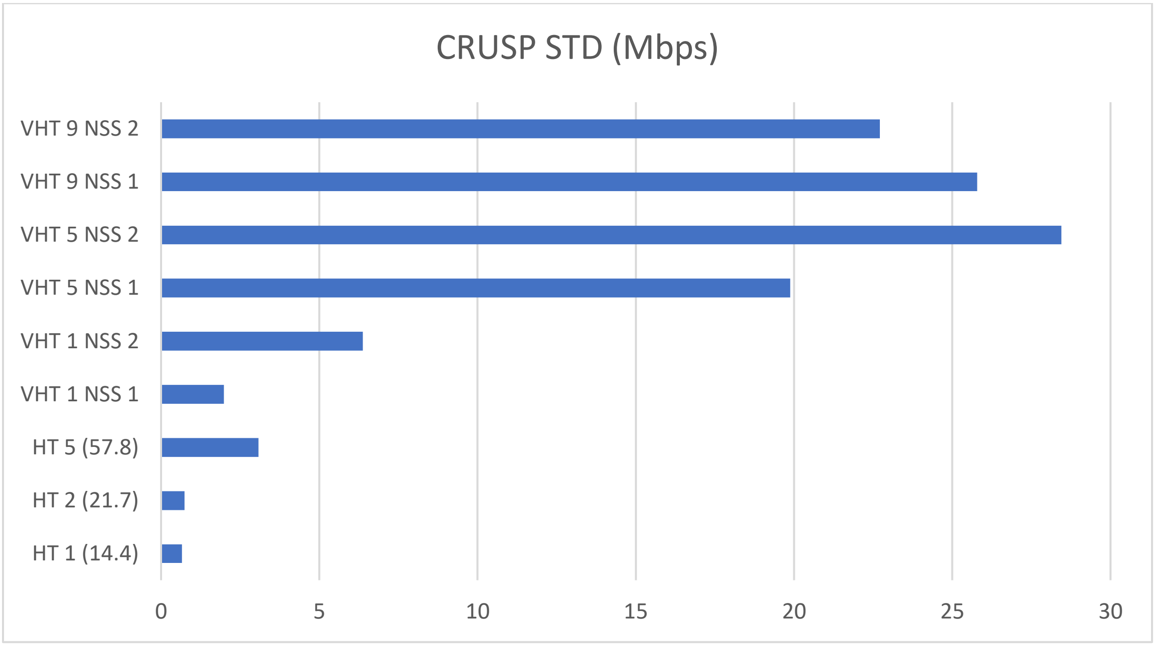
```
measurement_client_standalone > output > mcs > ! uplink_vht_mcs_9_nss_2
Naive Rate Sequence: 117.766426, Received packets: 744/930
Naive Rate Sequence: 109.4125, Received packets: 714/930
Naive Rate Sequence: 95.48, Received packets: 636/930
Naive Rate Sequence: 89.49014, Received packets: 579/930
Naive Rate Sequence: 108.60015, Received packets: 668/930
Naive Rate Sequence: 93.0309, Received packets: 812/930
Naive Rate Sequence: 85.70866, Received packets: 697/930
Naive Rate Sequence: 97.77265, Received packets: 735/930
Naive Rate Sequence: 118.78491, Received packets: 815/930
Naive Rate Sequence: 89.175, Received packets: 811/930
Naive Rate Sequence: 89.50459, Received packets: 825/930
Naive Rate Sequence: 131.20068, Received packets: 873/930
Naive Rate Sequence: 94.87656, Received packets: 618/930
Naive Rate Sequence: 123.67797, Received packets: 825/930
Naive Rate Sequence: 126.78842, Received packets: 886/930
Naive Rate Sequence: 129.01897, Received packets: 845/930
Naive Rate Sequence: 112.16697, Received packets: 710/930
Naive Rate Sequence: 134.73291, Received packets: 869/930
Naive Rate Sequence: 133.51118, Received packets: 863/930
Naive Rate Sequence: 123.08892, Received packets: 817/930
```

That brings up 2 concerns, of course:

1. The smaller differences as we approach 100 Mbps can map/project to significantly larger real-world ground truth differences. If a hypothetical model is not very precise, it may not be reliable enough to trust.
2. The **variance** of CRUSP's measurements itself also make it difficult to rely on in its present state.

Let's dive a little deeper into point 2 by looking at the standard deviation across all 100 samples from each of our 9 experiments:

MCS	STD (Mbps)
HT 1 (14.4)	0.657083898
HT 2 (21.7)	0.745121981
HT 5 (57.8)	3.073524038
VHT 1 NSS 1	1.980512346
VHT 1 NSS 2	6.374331059
VHT 5 NSS 1	19.87693166
VHT 5 NSS 2	28.44650165
VHT 9 NSS 1	25.78379087
VHT 9 NSS 2	22.7069163



While the standard deviation for the lower MCS indices are passable – i.e. the variance for any one CRUSP measurement should not be so far off the mean – the higher indices again bear out the problem. A 25+ Mbps standard deviation given a 200 Mbps ground truth can put us fairly far off target, especially if we are making adjustments/predictions using a regressor, for example.

All this is to say that more research is needed into why CRUSP underreports in the high bandwidth cases, and whether it can be adjusted for likely determines whether it is a viable tool to use.

Discussion + Future Work (Logistic/Polynomial Regression)

When I originally started research last Fall, the plan was to do a sort of survey of novel bandwidth probing techniques, eventually settling on one in place of iperf3 and/or other link-flooding techniques to recommend to use for the mesh network that the undergrads were working on. After much searching, I came to realize that there were very few novel techniques with open-source codebases, and even fewer with well-documented code that I could even hope to get up and running locally. In the end, I chose to dive much deeper into CRUSP instead, evaluating it from different perspectives in an effort to truly test its usability for real-world scenarios.

Now, a word on results: my conclusion is that CRUSP is extremely promising considering the massive advantage it has in terms of data volume and measurement time compared to iperf3. For lower MCS indices and longer distances, i.e. lower quality links, CRUSP is accurate enough to be used as a replacement for iperf3. For higher bandwidth links, such as the higher MCS indices for 802.11ac radios, it tends to underreport available bandwidth significantly. It's important to note that this is much better than the alternative of overestimating available bandwidth, and the pattern seems consistent enough that we can account for it and estimate the ground truth fairly closely using regression.

If I had the time, I would've liked to produce a more sophisticated model using logistic or polynomial regression to take into account other features, where I could have a labeled dataset of features as { CRUSP measurement, distance from router, MCS index, packets lost, … } and the ground truth iperf3 bandwidth as the label. I am fairly confident it would outperform my simple linear regressors. The only problem is accurately labeling such a dataset – a problem I could not quite solve was how to concurrently get the ABW and run a CRUSP measurement without one affecting the other, since if we perform them at different times the link state could change.

My research took me down many unexpected routes, and I learned about some things I had never even heard of! Thank you to Manasvini and Anirudh for trusting me to figure it out as I went and provide guidance when needed.