1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

10 Appendix

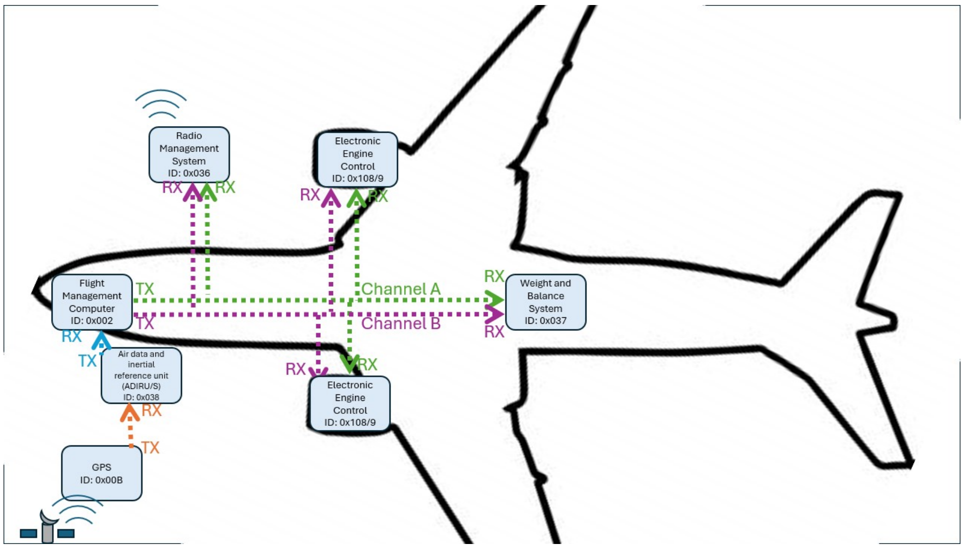

Figure 1: Project system model.

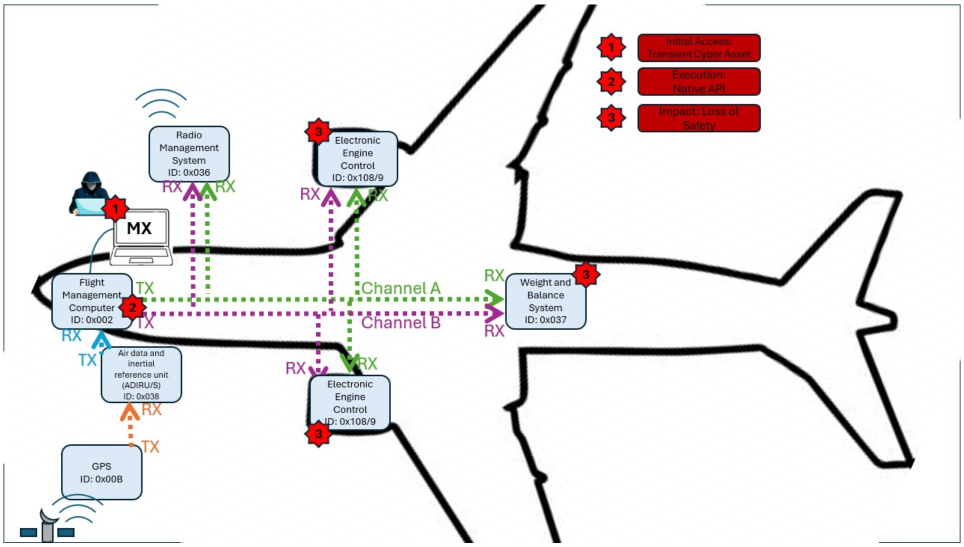

Figure 2: Project threat model.

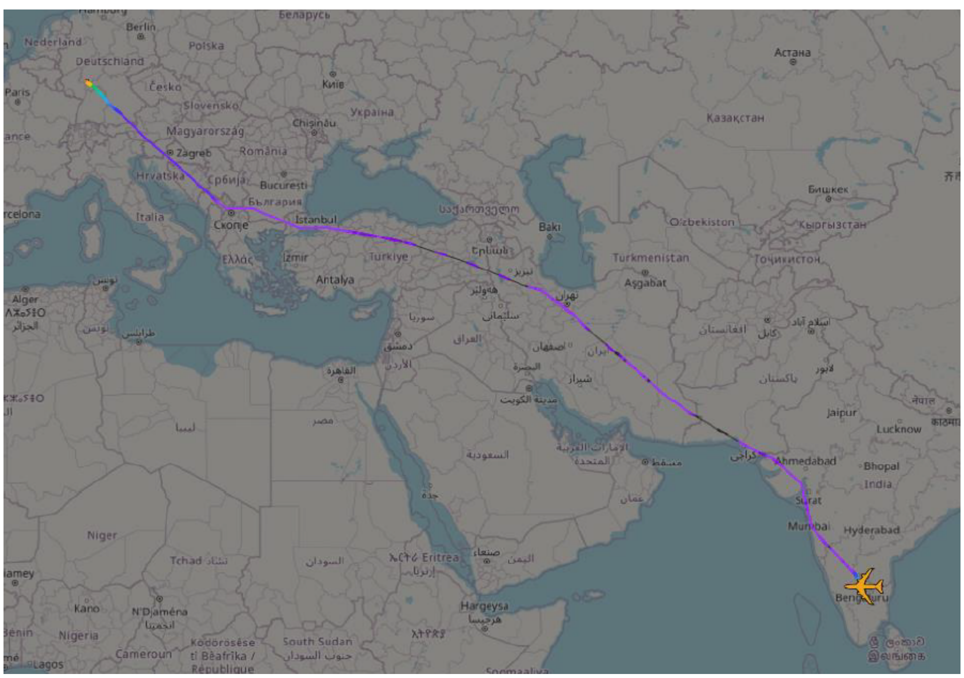

Figure 3: Project threat model.

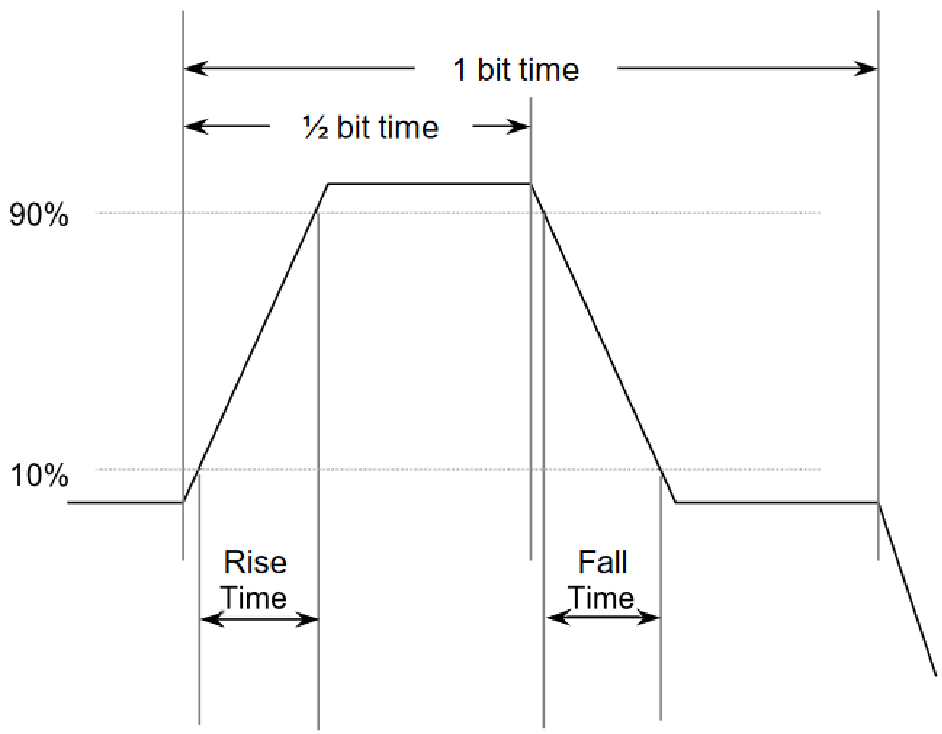

Figure 4: A graph specification for the '1' bit.[5]

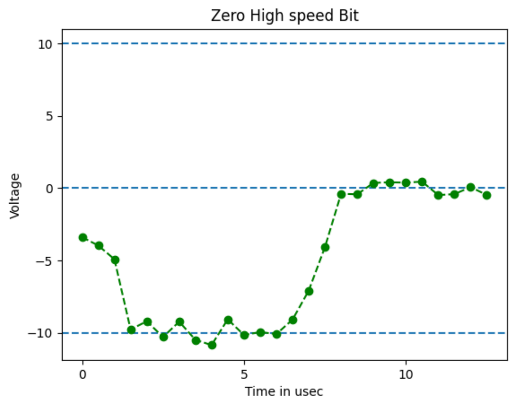

Figure 5: Voltage output samples for a ‘0’ bit at high speed.

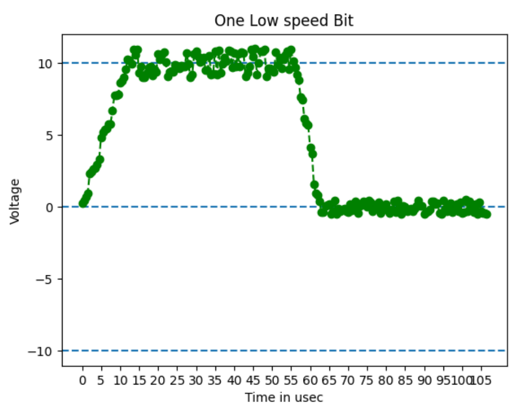

Figure 6: Voltage output samples for a ‘1’ bit at low speed.

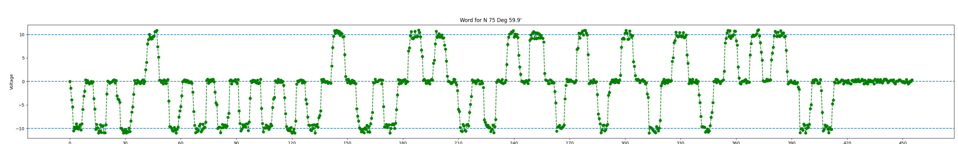

Figure 7: A word representing the latitude N 75 Deg 59.9’, high speed.

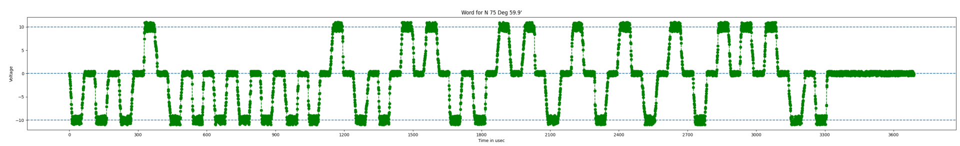

Figure 8: A word representing the latitude N 75 Deg 59.9’, low speed.

MSB **LSB**

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
P	**SSM**		**MSB**										**Data**								**LSB**	**SDI**					**Label**				

ARINC 429 32-bit Word Format

8	7	6	5	4	3	2	1	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
				Label				**SDI**		**LSB**						**Data**												**MSB**	**SSM**		**P**

ARINC 429 Word Transfer Order

Figure 9: ARINC 429 word format and bit transfer order[5]

```python
def BNR_encode(self, value:str, res:float, sig_digs:int, v_range:tuple) -> str:
    if(res >= 1.0):
        round_digs = 0
    else:
        round_digs = self.get_rounding_digits(sig_digs, v_range, self.affix_resoultion(res))
    # Error handling for bounds that is bad.
    if(round_digs != 0):
        larger_bounds = v_range[1]
        temp = int("1"*sig_digs, 2) / (10 ** round_digs)
        real_larger_bounds = temp * self.affix_resoultion(res)
        if(real_larger_bounds < larger_bounds and value > real_larger_bounds):
            value = real_larger_bounds # round down.
    # Taking from the ADIRU.
    val = float(value) / self.affix_resoultion(res)
    # Having and alg for finding this round_digs is key to this whole algorithm.
    val = round(val, round_digs)

    padding = "0" * (19-sig_digs)
    if(sig_digs == 20):
        padding = "0" * (20-sig_digs)
    # Positive sign
    SSM = "00"
    if(sig_digs == 20):
        # Sometimes it sigdigs cuts into this.
        SSM = "0"
    # Negative sign.
    if(val < 0.0):
        SSM = "11"
        if(sig_digs == 20):
            SSM = "1"
    # Start encoding to string of "0"s and "1"s.
    val = str(val).strip("-")
    if(val.__contains__("e")):
        tempV = int(val.split("e")[1])
        if(tempV > 0.0):
            val = val.split("e")[0] + "0"*(tempV-1) + ".0"
        if(tempV < 0.0):
            val = "0." + "0"*(tempV-1) + val.split("e")[0]
    # get right of a X.0 if the resolution is 1+
    round_digs_lacking = 0
    if(res >= 1.0):
        val = val.split(".")[0]
    else:
        #print(val)
        round_digs_lacking = round_digs - len(val.split(".")[1])
    # get rid of any decimal
    val = val.replace(__old: ".", __new: "")
    # get the bitstring from that value now
    val = bin(int(val + ("0"*round_digs_lacking)))[2:]
    # add leading zeros as necessary
    val = "0" * (sig_digs - len(val)) + val
    # get the full data field
    data = padding + val
    # reverse it because everything is fucking reverse order
    data = data[::-1]
    if(len(data) > 19 and sig_digs <= 19):
        #print(data)
        data = data[:19]
        #print(data)
    elif(len(data) > 20 and sig_digs <= 20):
        print(data)
        data = data[:20]
        print(data)
    return(data+SSM)
```

Figure 10: The general BNR Encoding function.

```python
def BCD_digs(self, value, res: float) -> str:

    #SDI = "00"
    #SSM = "00"
    #if(value < 0):
    #    SSM = "11"

    digits = str(value).strip("-")
    if (res >= 1.0):  # remove the stuff after 0.000000
        digits = digits.split(".")[0]
    digits = digits.replace( __old: ".",  __new: "")
    digits = "0" * (5 - len(digits)) + digits
    digits = digits[::-1]

    # e.g. 06572 knots
    # 11 - 14 -> 2
    # 15 - 18 -> 7
    # 19 - 22 -> 5
    # 23 - 26 -> 6
    # 27 - 29 -> 0

    digit5 = int(digits[0])
    dig5 = bin(digit5)[2:]
    dig5 = "0" * (4 - len(dig5)) + dig5
    dig5 = dig5[::-1]

    digit4 = int(digits[1])
    dig4 = bin(digit4)[2:]
    dig4 = "0" * (4 - len(dig4)) + dig4
    dig4 = dig4[::-1]

    digit3 = int(digits[2])
    dig3 = bin(digit3)[2:]
    dig3 = "0" * (4 - len(dig3)) + dig3
    dig3 = dig3[::-1]

    digit2 = int(digits[3])
    dig2 = bin(digit2)[2:]
    dig2 = "0" * (4 - len(dig2)) + dig2
    dig2 = dig2[::-1]

    digit1 = int(digits[4])
    dig1 = bin(digit1)[2:]
    dig1 = "0" * (3 - len(dig1)) + dig1
    dig1 = dig1[::-1]

    partial_data = dig5 + dig4 + dig3 + dig2 + dig1  # + SSM
    return (partial_data)
```

h

Figure 11: The general BNR Encoding function.

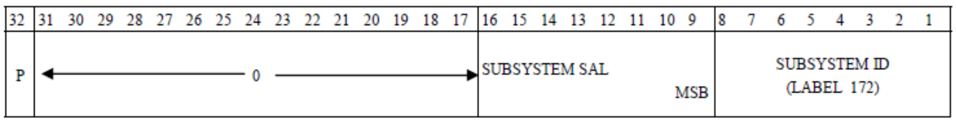

Figure 12: General SAL word format.

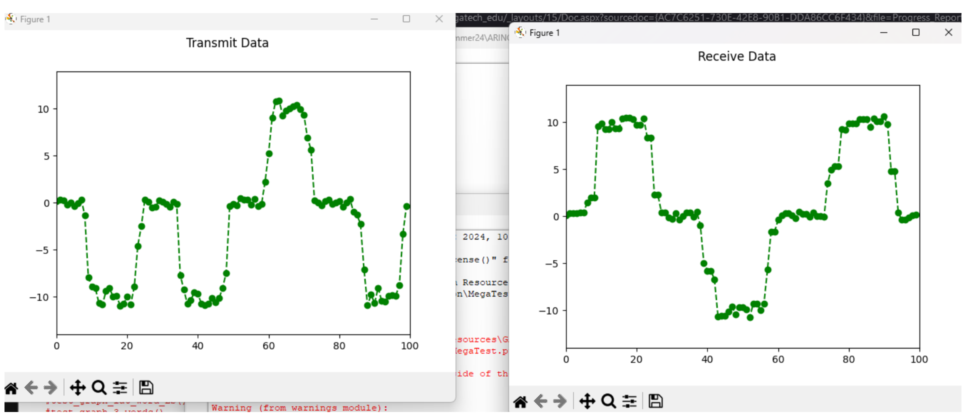

Figure 13: A snapshot of a transmitting (left) and receiving LRU (right) communicating using `BusQueue_Simulator.py` class.

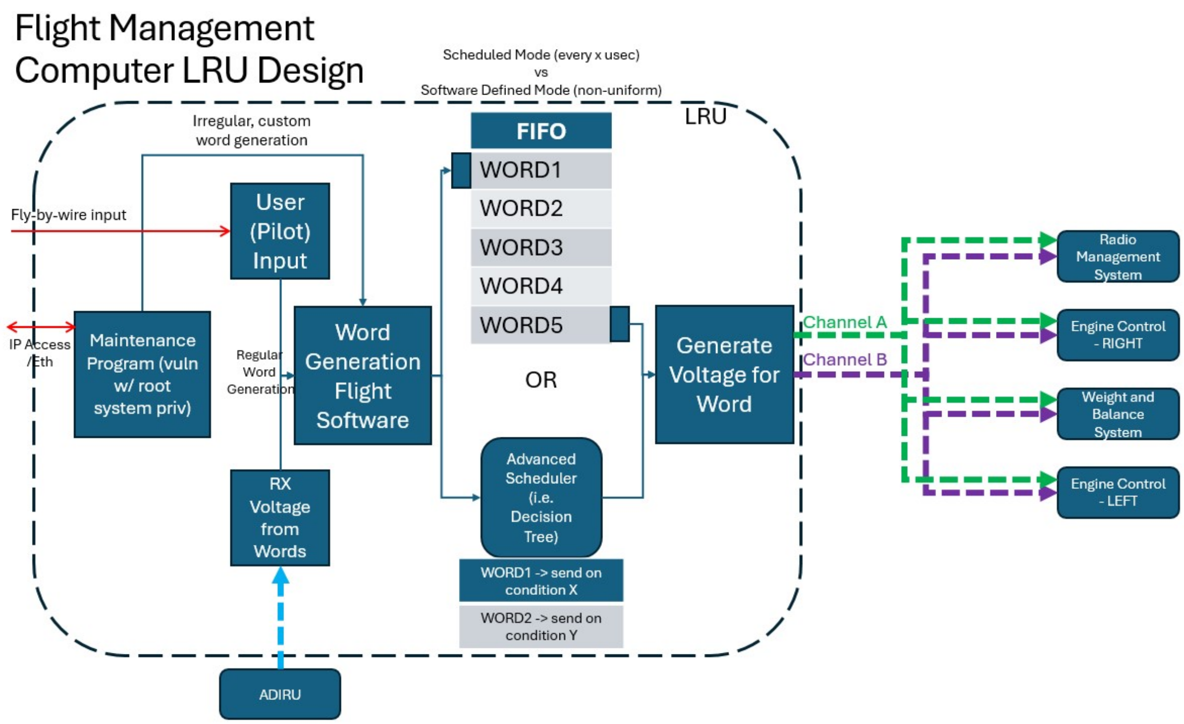

Figure 14: Graphical design blueprint for the FMC

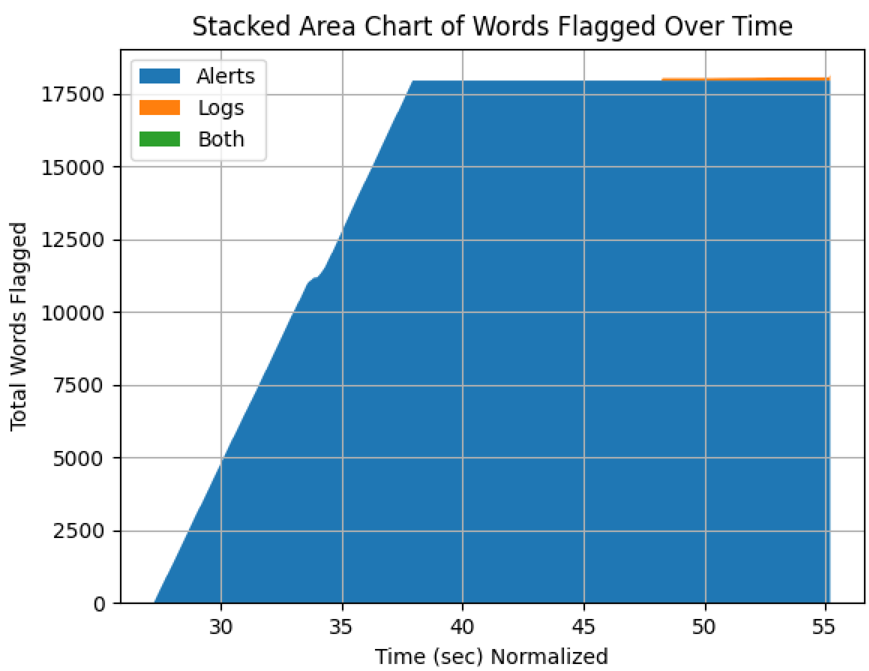

Figure 15: A graph of the alerts (blue) and logs (orange) generated from the IDS showing the attack present.

```python
if(attack_flag and (lats[gps_index] == 35.741 and lons[gps_index] == 50.578) ):
    # Simulate the attack:
    print(f"{Colors.RED}Executing Attack.{Colors.RESET}")
    # Uncomment this for demo.
    #cont = input("")
    downword = '01101100110000111100000000000000'
    for x in range(100):
        print(f"{Colors.GREEN}Sending word from FMC to RMS, FAEC-1, FAEC-2, and W&BS:\t\t\t\t\t0b{fmc_word1}{Colors.RESET}")
        print(f"{Colors.MAGENTA}Sending word from FMC to RMS, FAEC-1, FAEC-2, and W&BS:\t\t\t\t\t0b{fmc_word1}{Colors.RESET}")
        RMS_LRU.decode_word(downword)
        FAEC_1_LRU.decode_word(downword)
        FAEC_2_LRU.decode_word(downword)
        WnBS_LRU.decode_word(downword)
        RMS_LRU.decode_word(downword)
        FAEC_1_LRU.decode_word(downword)
        FAEC_2_LRU.decode_word(downword)
        WnBS_LRU.decode_word(downword)
        if(ids_flag):
            hit, _alert_log_ = IDS_main.alert_or_log(downword)
            if(hit):
                alert_logs_hits_time.append(time())
                als.append([time(),_alert_log_])
            IDS_main.n += 1
```

Figure 16: The code for the attack in `main_noThreads.py`

```
!Outfiles
#Default path is C:/ARINC_IDS/ or ./home/ARINC_IDS/
alerts = C:/ARINC_IDS/Alerts/Alerts.txt
logs = C:/ARINC_IDS/Logs/Logs.txt

!Channels
Orange: GPS -> ADIRU
Blue: ADIRU -> FMC
Purple: FMC -> RMS
Purple: FMC -> FAEC1
Purple: FMC -> FAEC2
Purple: FMC -> WnBS
Green: FMC -> RMS
Green: FMC -> FAEC1
Green: FMC -> FAEC2
Green: FMC -> RWnBS

!SDI
# Do not identify any TX LRUs SDI here.
Orange: ADIRU -> 11
Blue: FMC -> 11
Purple: RMS -> 00
Purple: FAEC1 -> 01
Purple: FAEC2 -> 10
Purple: WnBS -> 11
Green: RMS -> 00
Green: FAEC1 -> 01
Green: FAEC2 -> 10
Green: WnBS -> 11
```

Figure 17: Part 1 of default rules template file.

```
!Rules
# Example formats:
# <alert/log>* <channel>* <label> <SDI> data:<data> <SSM> <P> <Time> "<message (if alert)>"
# <log>* <channel>* <label>/<bits>* -> logs the decoded data for this channel & label.
# <alert/log>* <channel>* <bit[index1:index2)="01..10"> "<message (if alert)>"
# <alert/log>* <channel>* <label> <BCD/BNR/DISC> "<message (if alert)>"

# Some information:
# * = required field

# Labels must be in octal format 0oXXX since some of the data is the same for different words!
# E.G. ACMS Information is for both 0o062 and 0o063

# bit[index1:index2)="10..." option must have indices be integers in [1,33] (length of a word)
# AND the difference between must match the length of the given string
# E.G. bit[5:10)="11111" gets bits 5, 6, 7, 8, and 9.
# bit[20:33)="1011010010110" gets bits 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, and 32

# <SDI> = human-readable name for that channel.
# E.G. in examples above, Orange: ADIRU -> 11 would be ADIRU and not 11

# SSM must be one of the following options: 00, 01, 10 or 11

# Parity bit <P> is either C for correct or I for incorrect

# Time is an option that prepends the UTC time to the front of the log/alert
# E.G. <UTC Time recording>:"<message (if any)>"
# OR
# <UTC Time recording>:0000...1001 / <UTC Time recording>:<human-readable data point>
# Either the line will have 'time' in it before the message or it will not have it.

# If alerting / logging at the same time, log can only log the bits version and not the human-readable
# When alert/logging on BCD it's impossible to know without the hex ID if it's BCD/BNR/DISC/SAL. And when given the
# SDI, you know if it's already BCD/BNR/DISC/SAL anyway. So it's redundant.
# So if you specify BCD/BNR/DISC/SAL option, it gives you the percent change of being encoded to that option.
# and adds that to the message.

# Examples (REPLACE THIS SECTION HERE WITH YOUR OWN RULES):
# Alert on if there is a word sent through the Blue channel
alert Blue "Blue bus has a word."
# Alert on any latitude words sent.
alert Blue 0o010 "Latitude word sent"
# Log all the longitudinal data in human-readable format
log Blue 0o011
# Log the word-bits that were sent on the Blue channel
log Blue bits
# Alert and log on this particular word in the Orange Channel
# So logs the full bit-string representation of this word into Logs.txt
# Alerts on 6000 knots for ground speed, with correct parity as message of "Plane's speed is 6000 Knots"
# SSM = 00
alert Blue log 0o012 ADIRU data:6000 00 C "Plane's speed is 6000 Knots"
# Alerts on any word containing alternate bits as the data section in the purple bus.
alert Purple bits[11:29)=010101010101010 "Funny Pattern!"
# Alerts on any word with that particular label that is BCD as opposed to BNR
# For this label there are 6 different things it could be with 5 encoded as BNR and 1 as BCD
# So the message will be appended with: ". Percent Chance of being BCD: 16.667%."
alert Purple 0o062 BCD "Tire Loading (Left Wing Main) Word!"
# Same as above, but just prepends time to the logged alert
alert Purple 0o062 BCD time "Tire Loading (Left Wing Main) Word!"
# etc
```

Figure 18: Part 2 of default rules template file.

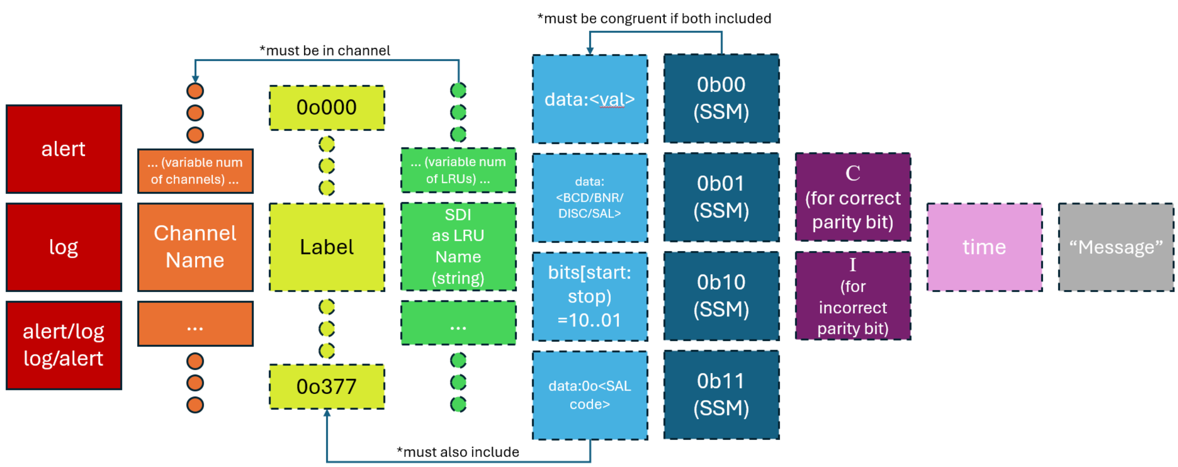

Figure 19: Visual rolodex of rules syntax options.

```python
def alert_or_log(self, word: str):
    flag_this_tuple = False

    this_word_alerted_or_logged = False

    #self.rules.add(
    # 0 (alert_log,
    # 1 channel,
    # 2 bitmask,
    # 3 parity_check,
    # 4 time_notate,
    # 5 message)
    # )
    a_o_r = "None"
    with open(self.log_filepath, "a") as log_fd:
        with open(self.alert_filepath, "a") as alert_fd:
            for _tuple_ in self.rules:
                parity = _tuple_[3]
                time_notate = _tuple_[4]
                flag_this_tuple = False
                # Part 1 Check if you should flag this word.
                if(_tuple_[2] == "0"*31 and parity == None):
                    flag_this_tuple = True
                if (_tuple_[0].__contains__("alert") or _tuple_[0].__contains__("log")):
                    #Check channel? -> done by caller
                    psuedo_word = word[:-1]
                    partity_calc = lru_txr()
                    correct_parity_bit = partity_calc.calc_parity(psuedo_word)
                    #p_bitmask = _tuple_[2] + correct_parity_bit
                    bitmask = _tuple_[2]  #+ lru_txr.calc_parity(_tuple_[2])
                    #if (bitmask == 31 * "0"):
                    #    flag_this_tuple = True
                    #word_check = word[:-1]
                    #if (int(bitmask, 2) & int(psuedo_word, 2) == int(bitmask, 2)):
                    if (bitmask[0:10] == psuedo_word[0:10] and (int(bitmask[10:],2) == 0 or (int(bitmask[10:],2) ^ int(psuedo_word[10:],2) == 0)) ):
                        if ((parity == True and word[-1] == correct_parity_bit)
                                or (parity == False and word[-1] != correct_parity_bit)):
                            # alert
                            flag_this_tuple = True
                        elif (parity == None):
                            # alert
                            flag_this_tuple = True
                # Part 2: If the word is flagged, the log it appropriately.
                if (flag_this_tuple and time_notate):
                    a_o_r = _tuple_[0]
                    this_word_alerted_or_logged = True
                    if (_tuple_[0].__contains__("alert")):
                        alert_fd.write(f"{ctime()}: Alert! {_tuple_[5]}\n")
                    if (_tuple_[0].__contains__("log")):
                        #input(_tuple_)
                        log_fd.write(f"{ctime()}: {word} {_tuple_[5]}\n")
                elif (flag_this_tuple and time_notate == False):
                    a_o_r = _tuple_[0]
                    this_word_alerted_or_logged = True
                    if (_tuple_[0].__contains__("alert")):
                        alert_fd.write(f"Alert! {_tuple_[5]}\n")
                    if (_tuple_[0].__contains__("log")):
                        #input(_tuple_)
                        log_fd.write(f"Logged word #{self.n}: {word} {_tuple_[5]}\n")
        alert_fd.close()
    log_fd.close()
    return((this_word_alerted_or_logged, a_o_r))
```

p

Figure 20: Code of the function that alerts/logs on a word based on its bitmask.

```
def test_rules_AllDataTypes():
    current_dir = getcwd()
    filename = current_dir + "\\IDS_Rules_test_files\\" + "rules_data_stress_test.txt"
    IDS_test_default = IDS(rules_file=filename)
    sdi = IDS_test_default.sdis
    print(sdi)
    rulez = IDS_test_default.rules
    assert(rulez == [('alert/log','Channel1','00010000110000000000010000000000',True,False,'Longitude is -40.0 Degrees'),
            ('alert/log','Channel1','01010000110000000000000011000000',True,False,"Plane's speed is 6000 Knots"),
            ('alert/log','Channel1','11010000111001100110100100000000',None,False,"Plane's Track Angle is 259.9 Degrees"),
            ('alert/log','Channel1','00110000111100000110010000000000',None,False,"Plane's Magnetic Heading is 98.3 Degrees"),
            ('alert/log','Channel1','10110000111100010010000000000000',True,False,'Wind Speed is 98.3 Knots'),
            ('alert/log','Channel1','11111001111100010010001010010000',True,False,'Baro Correction (ins. Hg) '),
            ('alert','Channel2','00000010000000101111110000000000',None,False,'Selected Course '),
            ('alert','Channel2','00100010000101101000000000000000',None,False,'Selected Vertical Speed is 1432 Ft/Min upwards'),
            ('log','Channel1','01010110110000000101011011000000',None,False,'Cabin Pressure is 1748 mB'),
            ('log','Channel1','11100101011000100110111110000000',False,False,'Horizontal Figure of Merit just under 2 nautical miles.'),
            ('log/alert','Channel1','10001001111001111010100000000011',None,False,'Indicated Angle of Attack is -70 WARNING!!'),
            ('log','Channel2','10000011000000111100001111000000',None,False,'Application Dependant Data 1 for FMC'),
            ('log','Channel2','11111111000000000000000000000100',None,False,'Identification required for FMC'),
            ('log','Channel2','01100011001100110110110010110000',None,False,'Application Dependant Data 2 for FMC'),
            ('alert','Channel1','00011101111001001010110110010000',None,False,'ADIRU Discrete Data '),
            ('alert/log','Channel1','00010111111001001010110110010000',None,False,'ADIRU MX Data '),
            ('alert','Channel2','00001111000000011110000000000000',None,False,'Aircraft Condition and Event Surveillance System (ACESS)'),
            ('alert','Channel2','10111110000101111110000000000000',None,False,'HGA/IGA HPA'),
            ('alert','Channel2','01011111000010111111000000000000',None,False,'CABIN TERMINAL 3'),
            ('alert', 'Channel2', '00010011000000100110000000000000', None, False, 'GPWS'),
            ('alert','Channel2','10001111000100011110000000000000',None,False,'Electronic Flight Instrument System (EFIS)')])
```

Figure 21: Example of a typical function to test simulation correctness.

Bus Speed (% slowed down)	Voltage Sample Rate	Words Correct (of 5) averaged over 0-10 rules	Bits Correct per word averaged over 0-10 rules
-1,000,000%	0.5 sec (½ second)	5/5	32, 32, 32, 32, 32
-100,000%	0.05 sec (1/20th of a second)	5/5	32, 32, 32, 32, 32
-10,000%	0.005 sec (5 milliseconds)	5/5	32, 32, 32, 32, 32
-1,000%	0.0005 sec (½ millisecond)	5/5	32, 32, 32, 32, 32
-100%	0.00005 sec (1/20th of a millisecond)	5/5	32, 32, 32, 32, 32
-10%	0.000005 sec (5 microseconds)	4/5	32, 32, 32, 32, 31
-0%	0.0000005 sec (½ microsecond)	1/5	32, 31, 30, 29, 28

Figure 22: Results from `IDS_Eval1.py`.

```
"C:\Users\mspre\Desktop\Practicum Resources\GATech_MS_Cybersecurity_Practicum_InfoSec_Summer24\venv\Scripts\python.exe" "C:\Users\mspre\Desktop\Practicum
 Resources\GATech_MS_Cybersecurity_Practicum_InfoSec_Summer24\ARINC429 Simulation\IDS_EVAL2.py"
Opening and analyzing flight data...
Finished opening and analyzing flight data in 13.999 seconds.
Press enter to start the test.
```

Figure 23: Setup for test `IDS_Eval2.py`.

```
Sending words:
Airspeed BCD:                  0b00011001001100000101000000000001,
Airspeed BNR:                  0b00010001000010110110001000000000,
Corrected Angle of Attack:     0b10000101000111001000000000000111,
Indicated Angle of Attack:     0b10001001000111001000000000000111,
Latitude BNR:                  0b00010011001110110001111001110001,
Longitude BNR:                 0b10010011001110110001111001110000
```

Figure 24: Sample output from `IDS_Eval2.py`.

```
Sending words:
Airspeed BCD:                  0b00011001000000000000000000000001,
Airspeed BNR:                  0b00010001000000000000000000000000,
Corrected Angle of Attack:     0b10000101000000000000000000000001,
Indicated Angle of Attack:     0b10001001000011100100000000000111,



This concludes Eval 2. It took 10164.366 seconds.
Number of alerts: 12726074
Number of logs: 9623

Process finished with exit code 0
```

Figure 25: Results from `IDS_Eval2.py`.

```
Sending words:
Airspeed BCD:                   0b00011001000000000000000000000001,
Airspeed BNR:                   0b00010001000000000000000000000000,
Corrected Angle of Attack:      0b10000101000000000000000000000001,
Indicated Angle of Attack:      0b10001001001011100100000000000110,



Sending words:
Airspeed BCD:                   0b00011001000000000000000000000001,
Airspeed BNR:                   0b00010001000000000000000000000000,
Corrected Angle of Attack:      0b10000101000000000000000000000001,
Indicated Angle of Attack:      0b10001001000011100100000000000111,



This concludes Eval 2. It took 9794.079 seconds.
Number of alerts: 12726074
Number of logs: 9623

Process finished with exit code 0
```

Figure 26: Another run with results from `IDS_Eval2.py`.

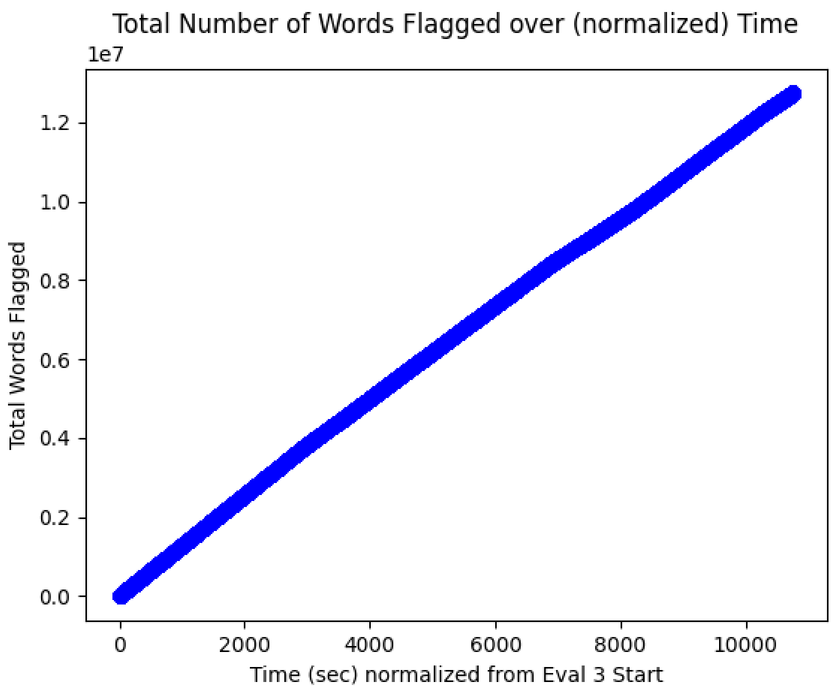

Figure 27: Number of alerted/logged words until that point over time for `IDS_Eval2.py`.

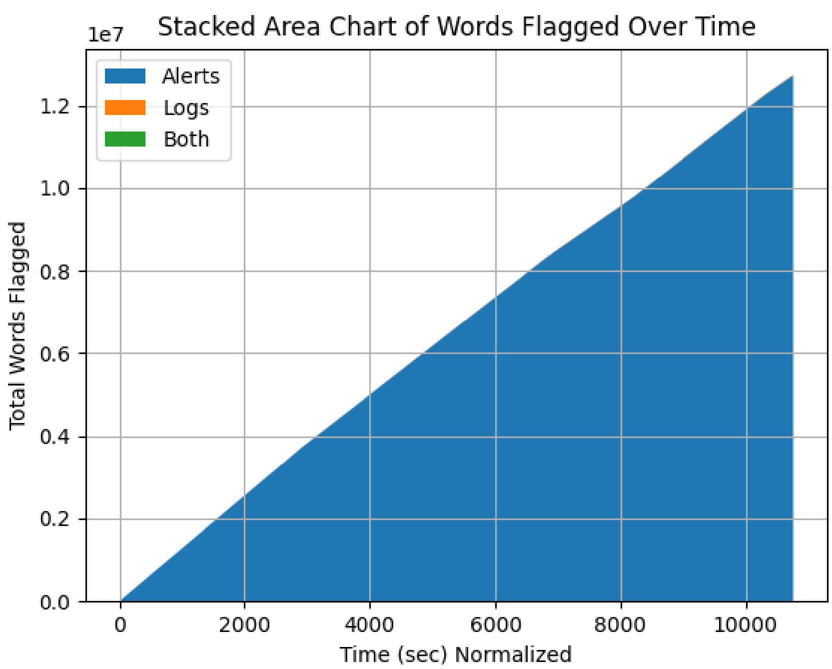

Figure 28: Cumulative graph of number of alerted/logged words over time for `IDS_Eval2.py`. Blue is alerts and Orange is Logs.

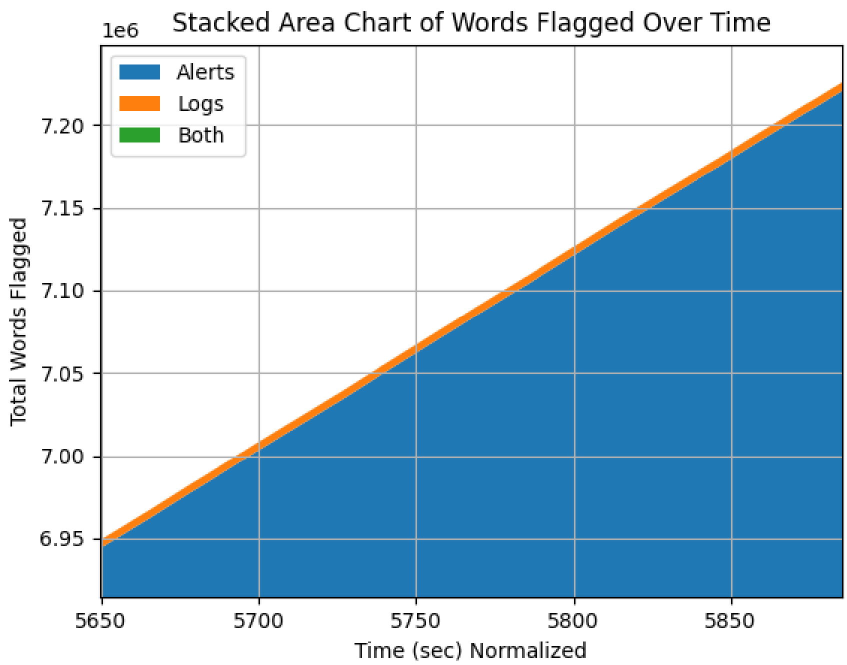

Figure 29: Zoomed in view to see figure 27 delineation more clearly.

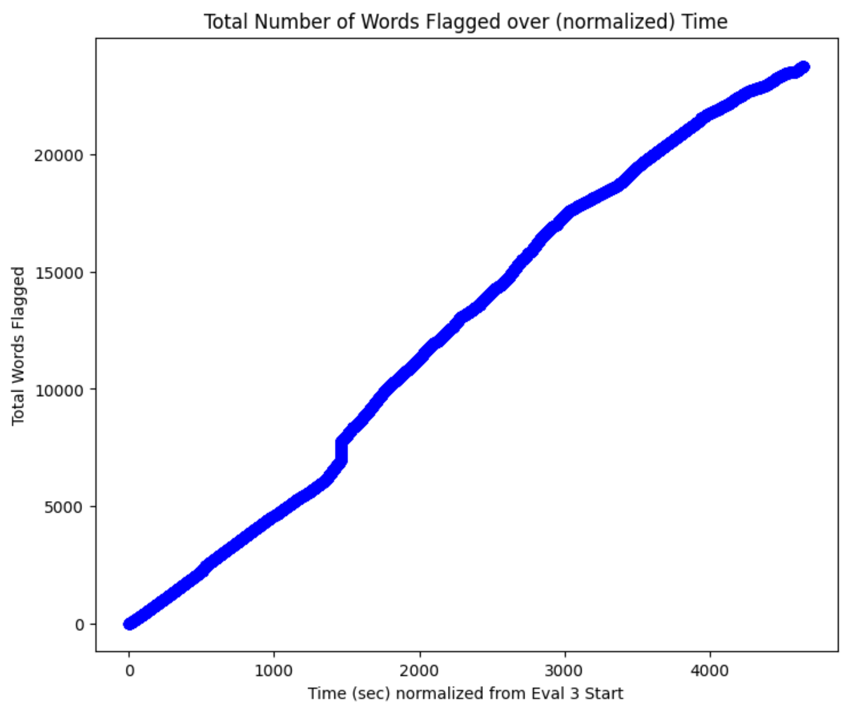

Figure 30: Number of alerted/logged words until that point over time for `IDS_Eval3.py`

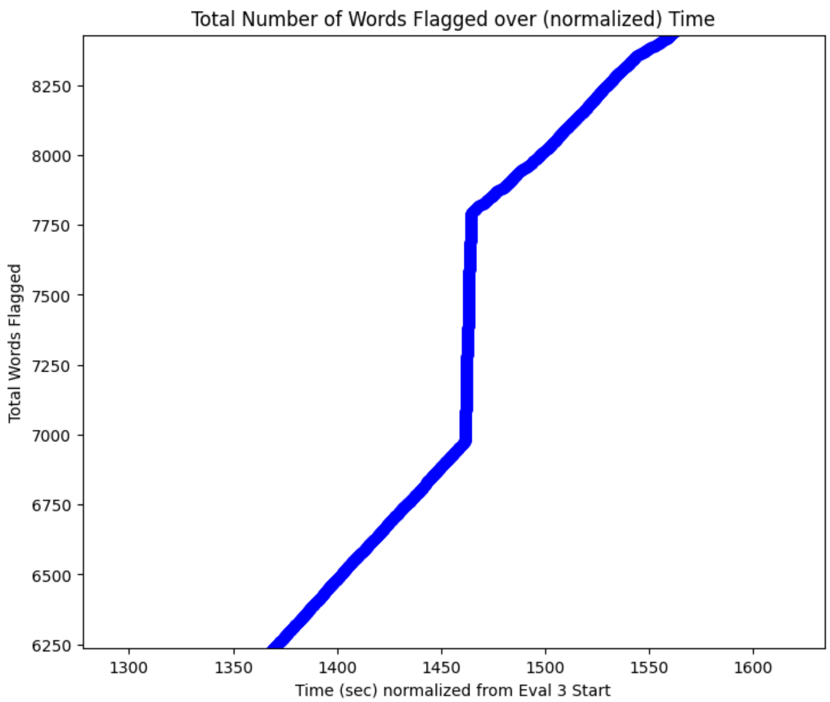

Figure 31: Zoomed in graph for `IDS_Eval3.py` showing the huge jump in alerts because of the attack words.

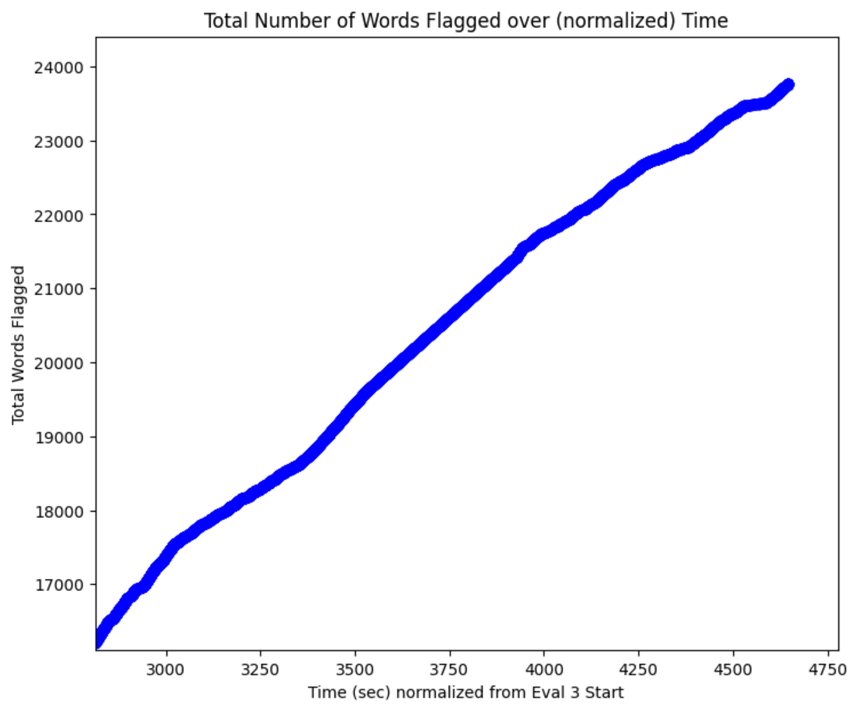

Figure 32: Zoomed in graph for `IDS_Eval3.py` showing the end of the flight as words are alerting sporadically on the plane's descent.

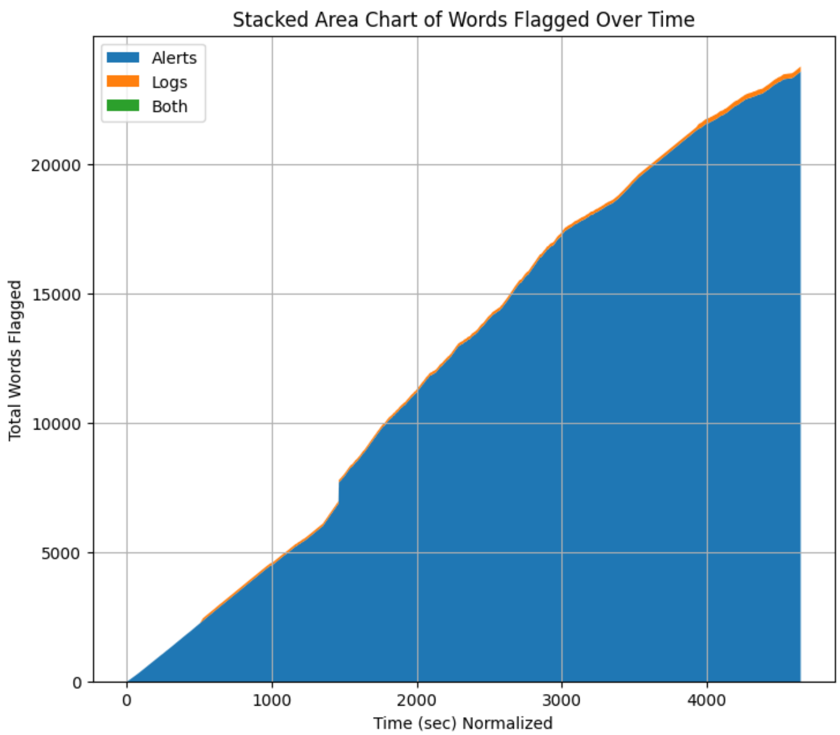

Figure 33: Cumulative graph of number of alerted/logged words over time for `IDS_Eval3.py`. Blue is alerts and Orange is Logs.

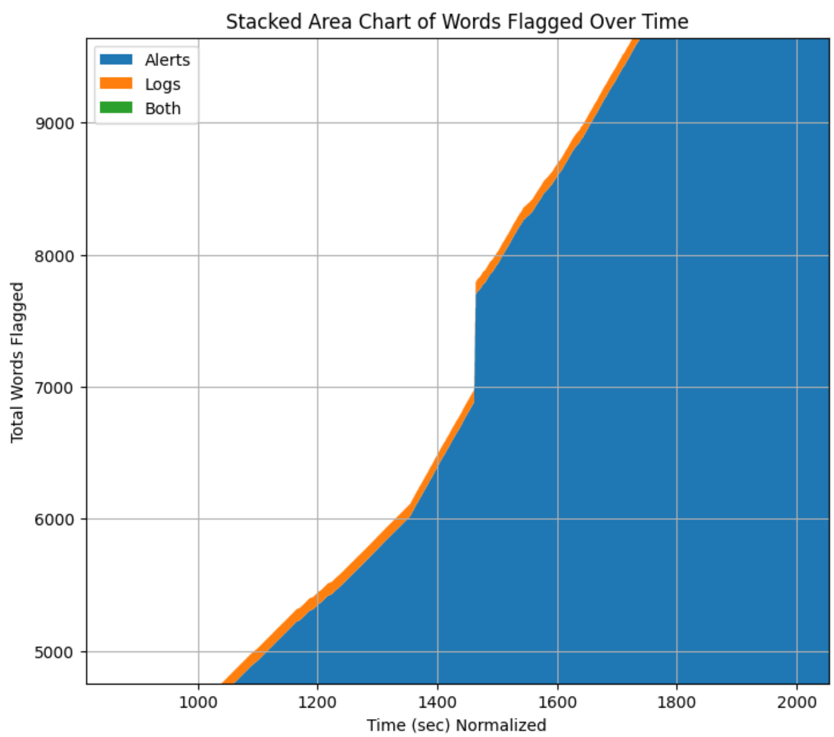

Figure 34: Zoomed in cumulative graph for `IDS_Eval3.py`. Note that the jump purely in blue (alerts on the attack) while no logs are generated.

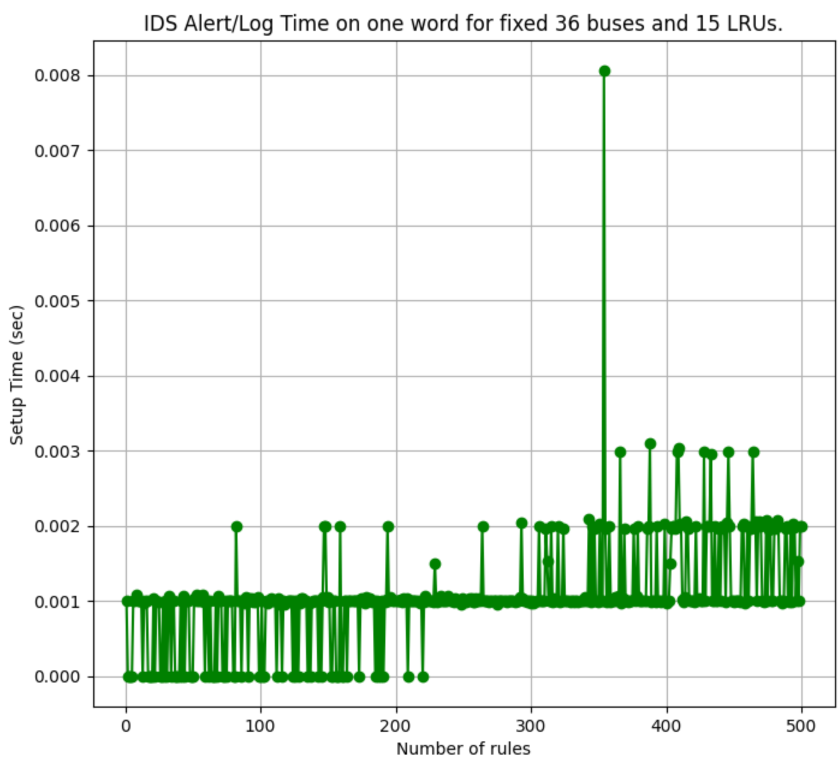

Figure 35: `IDS_Eval4.py`: Time to process 1 word for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

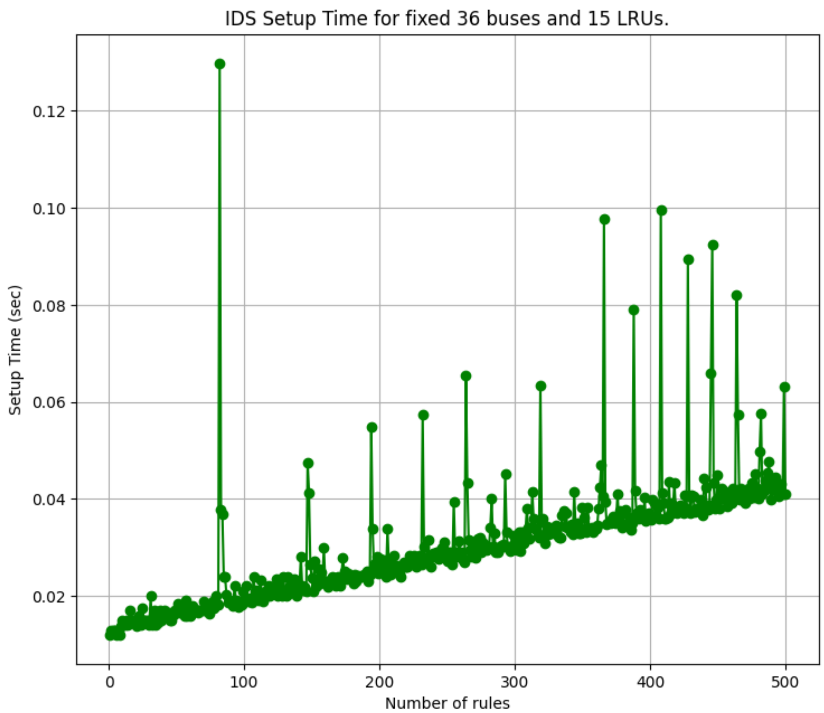

Figure 36: `IDS_Eval4.py`: Time to boot-up the IDS for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

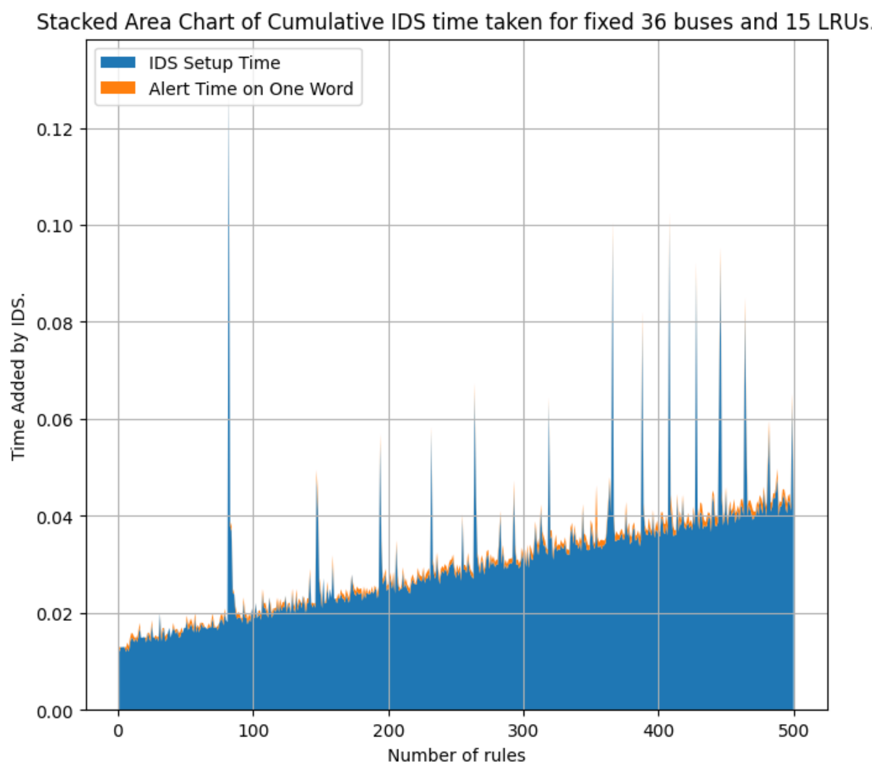

Figure 37: `IDS_Eval4.py`: Cumulative time added for figures 35 and 36.

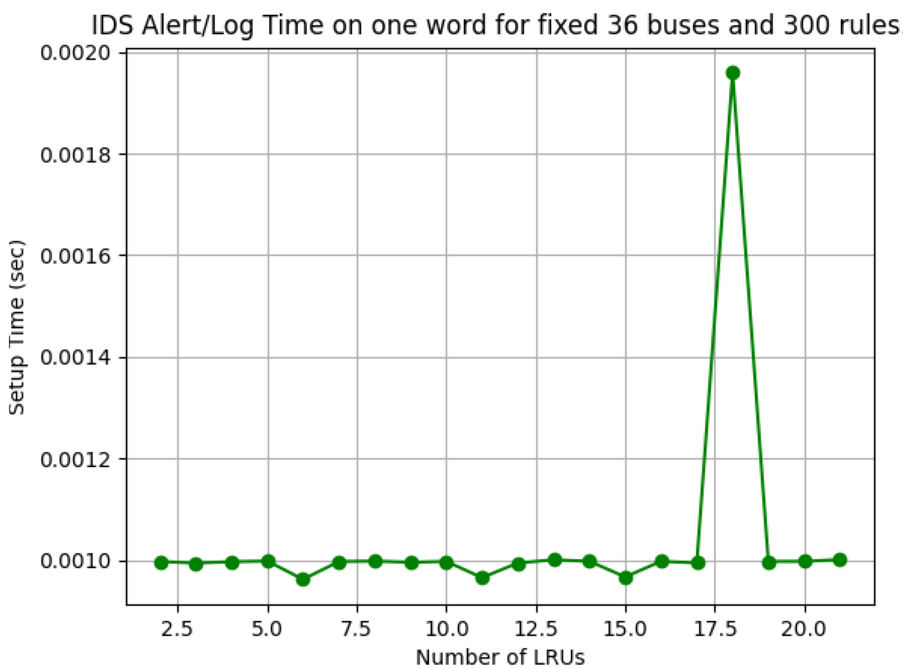

Figure 38: `IDS_Eval4.py`: Time to process 1 word for 2 to 21 LRUs while fixing the number of channels to 36 and rules to 300.

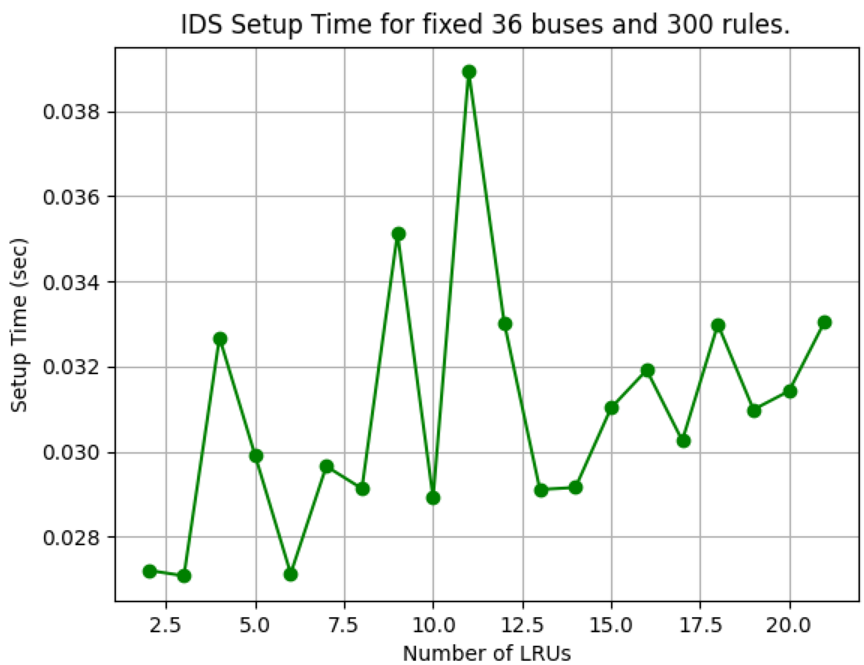

Figure 39: `IDS_Eval4.py`: Time to boot-up the IDS for 2 to 21 LRUs while fixing the number of channels to 36 and rules to 300.

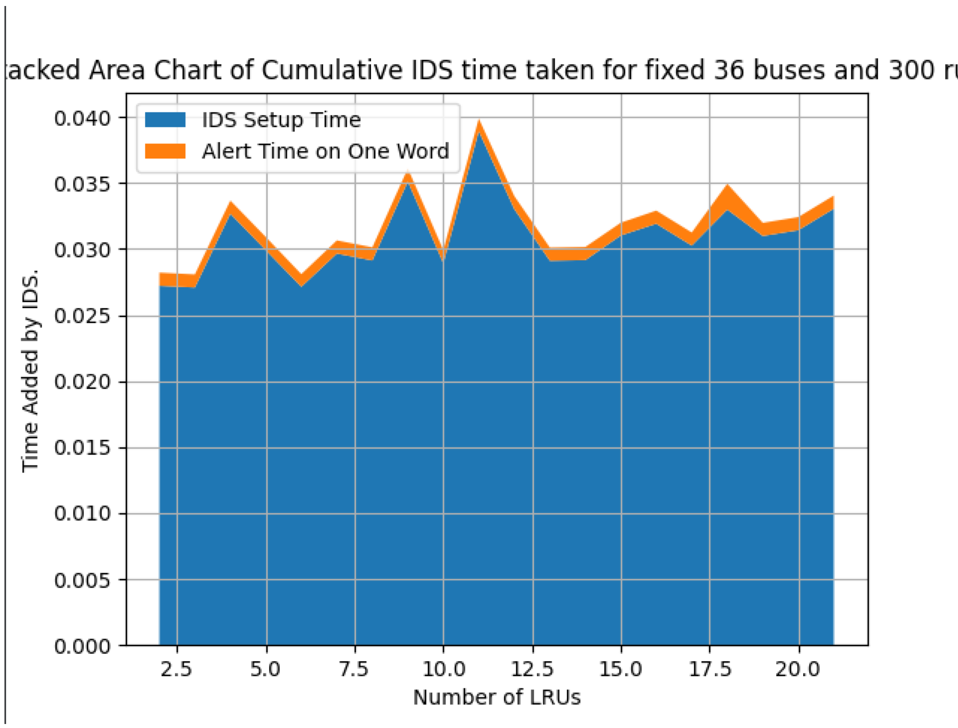

Figure 40: `IDS_Eval4.py`: Cumulative time added for figures 38 and 39.

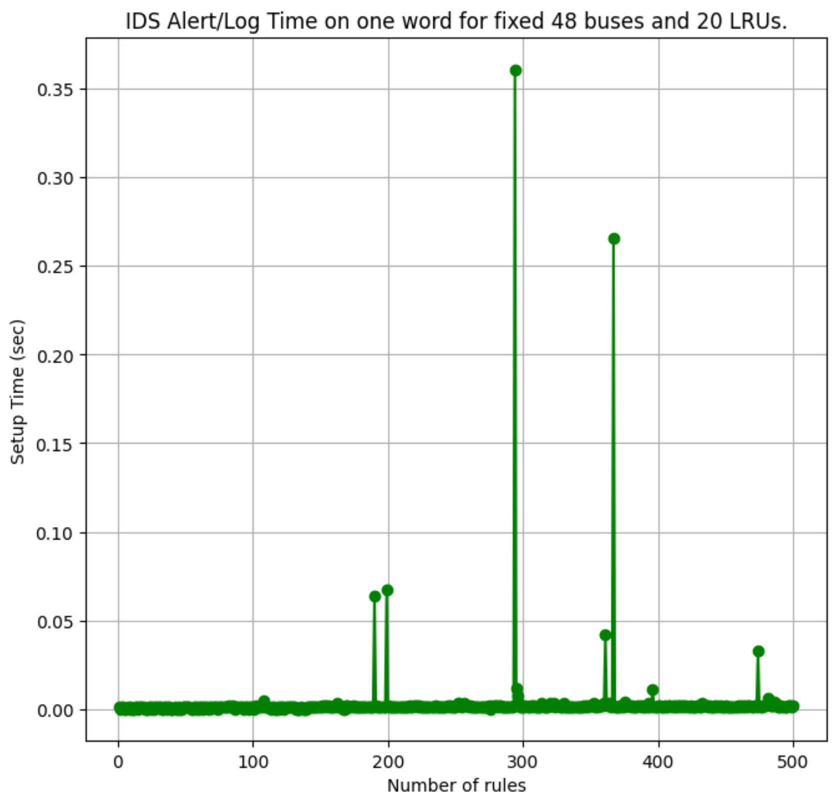

Figure 41: `IDS_Eval4.py`: Time to process 1 word for 1 to 500 rules while fixing the number of channels to 48 and LRUs to 20.

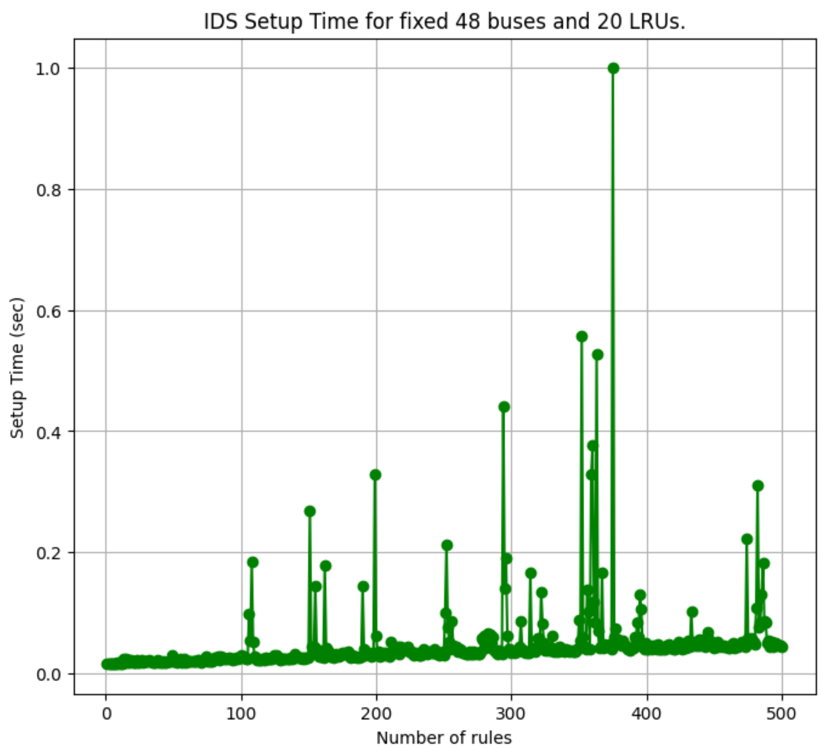

Figure 42: `IDS_Eval4.py`: Time to boot-up the IDS for 1 to 500 rules while fixing the number of channels to 48 and LRUs to 20.

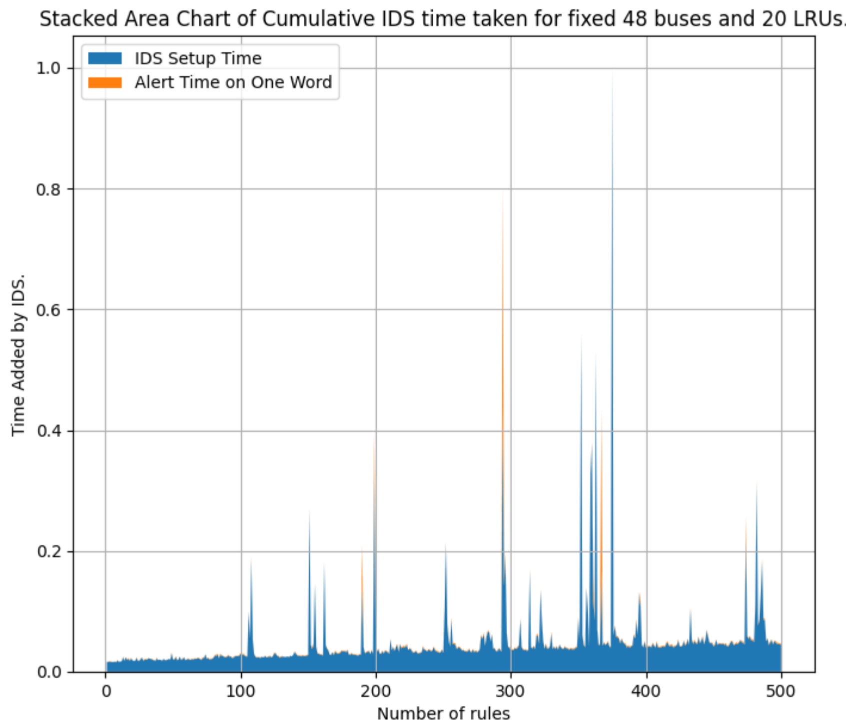

Figure 43: `IDS_Eval4.py`: Cumulative time added for figures 41 and 42.

Figure 44: How to change the IDS into an IPS.