Lowering Avionics Bus Trust: Moving ARINC 429 Bus Architecture Towards Zero Trust

Matthew Preston
Georgia Institute of Technology
https://github.com/PrestonMatt/GATech_MS_Cybersecurity_Practicum_InfoSec_Summer24
mpreston9@gatech.edu
mspreston98@gmail.com

Abstract

In avionics, bus architectures such as ARINC 429, MIL-STD-1553 and ARINC 629, play critical roles for aircraft computer communication. Designed in the 1960s and 70s, these systems are robust and reliable, but are no longer resilient to modern cybersecurity threats. With respect to the project of ARINC 429, the problem with these bus architectures is that they are inherently the opposite of zero-trust. If an adversary gains access to the bus, they can potentially issue unchecked commands to line replaceable units (LRUs) in cyber-physical systems, leading to catastrophic kinetic effects. This practicum aims to explore and develop solutions to enhance the security of the ARINC 429 bus architecture by engineering and enforcing zero trust security measures, thereby mitigating the risk of unauthorized access and malicious commands.

Keywords – ARINC 429 (a.k.a. Aeronautical Radio Incorporated Standard 429, or Mark 33 Digital Information Transfer System (DITS) Standard), Cybersecurity, Avionics, Zero Trust, Zero Trust Architecture(s) (ZTA), Cyber-Physical Systems, Bus Architectures, Internal Defense System (IDS)

1 Introduction

Looking at a baseline analysis of ARINC 429 with the C.I.A. triad, ARINC 429 is like most other bus communication architectures. While it is robust and provides high availability in nearly all weather conditions, it is lacking in integrity - only having a parity bit at the end of a word - and has no confidentiality. In the event of a cyber attack where an attacker can ingress onto the bus, there are no defenses to keep availability up, and as long as the parity check is performed by the hacker, there is no way to tell integrity between line replaceable units (LRUs). Moving on to NIST's 7 tenets of Zero Trust, Zero Trust, does not meet tenet 2 onwards.[1]

Why try to convert/upgrade these legacy systems to Zero-Trust? Executive Order 14028 directs governmental agencies to implement Zero-Trust Architectures for US Government systems. For civilian systems where lives are on the line, like a Boeing 737 which is one of the most common civilian aircraft, similar security measures must be attempted to be met for the safety of the public and crews flying on these planes every day. It should be noted that, as part of its design and function, the B737 uses ARINC 429 communication bus which makes ARINC 429 an attractive target for both hackers to attack and defenders to secure.

This project completed the following 3 requirements: a simulator for the ARINC 429 bus, a proof of concept for and ARINC 429 attack, and an Intrusion Defense System implementation for the ARINC 429 bus. On its own, this work will not solve the ZTA problem for ARINC nor for bus architectures. However, it will help move ARINC 429 towards being a ZTA. From NIST Special Publication 800-207, there are 7 basic tenets of zero trust. I will show in the course of this report the following movement (or lack thereof) on each of the tenets:

1. "All data sources and computing services are considered resources."[1]
   In my model, I am considering LRUs, external connections and the bus itself as resources (logically speaking) to fly a plane. By definition of simulation framework this tenet is met.

[1] NIST Special Publication 800-207 https://doi.org/10.6028/NIST.SP.800-207

2. “All communication is secured regardless of network location.”[1]
   The IDS will support this effort.

3. “Access to individual enterprise resources is granted on a per-session basis.”[1]
   This is not addressed by my defense or models as per the specification for ARINC 429, there is no such thing as a session. Additionally, access to the IDS reports will be only available to operators of the plane. There is no plan to limit access to any (machine-to-machine perspective) LRU to the bus as per necessary, since availability is critical in the model.

4. “Access to resources is determined by dynamic policy—including the observable state of client identity, application/service, and the requesting asset—and may include other behavioral and environmental attributes.”[1]
   The IDS addresses this tenet similarly as tenet 2, with the rules-based system that can change based on needs of the operators.

5. “The enterprise monitors and measures the integrity and security posture of all owned and associated assets.”[1]
   As part of the IDS, monitoring of the bus is required. However, it will not continuously monitor LRUs for vulnerabilities in their software.

6. “All resource authentication and authorization are dynamic and strictly enforced before access is allowed.”[1]
   This will not be addressed.

7. “The enterprise collects as much information as possible about the current state of assets, network infrastructure and communications and uses it to improve its security posture.”[1]
   Logging bus traffic as part of the defense functionality has been implemented.

From the responses to each tenet above, I want to clarify that the aim of this project was not to completely match up defense with each one holistically, but rather to initiate movement toward ZTA. For example, it’s unrealistic and unsafe to give authentication protocols to LRUs on a bus where a cyber-physical system needs constant access to fly.

While there exist a few tutorials to create your own ARINC 429 interfaces for real-world LRUs[2] and previous coding projects that implement parts of ARINC 429[3], generally speaking real ARINC 429 modules and LRUs cost in the orders of thousands of dollars, and to the author’s knowledge there are no publicly available ARINC 429 bus simulators that account for interaction between LRUs nor any that try to implement any security measures[4].

2 Project Models

2.1 System Model

Part of the motivation for this project is the US Government’s interest in securing its technologies by applying a Zero-Trust framework[5]. In 2022 The White House commanded government agencies to move towards zero trust in Executive Order 14028. The intent was to move even sub-components of larger systems (such as ARINC 429) to ZTA. ARINC 429 has specified up to 833 equipment ID codes, of which 243 are currently designated with specific LRUs. The simulation this project ultimately created was designed around a specific simplified model of an airplane with the following LRUs and 4 bus channels[6]:

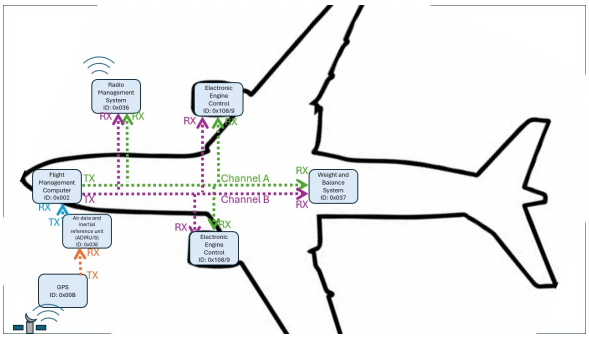


Figure 1: Project system model.

In this system model, the Flight Management Computer is both a receiver and transceiver, as it is a computer with the appropriate ARINC 429 RX and TX chips. Therefore, it is receiving on one

[2] Interfacing Electronic Circuits to Arduinos: https://www.instructables.com/Interfacing-Electronic-Circuits-to-Arduinos/

[3] PyARINC429: https://github.com/aeroneous/PyARINC429

[4] https://github.com/topics/arinc429

[5] https://whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf

[6] Also see appendix for copies of each graphic in larger resolution.

bus to get positional, altitude, heading, etc. data to present to the pilots in the cockpit, while broadcasting/transmitting on 2 other lines. These 2 lines are redundant for safety reasons. They allow the pilot to control the plane's motion from the cockpit consoles and controls which are physically interfacing with the Flight Management Computer.

2.2 Threat Model

The most likely attacker against systems with ARINC 429 would be a nation-state actor looking to gather intelligence or to physically damage critical infrastructure. For example, nation-state actors are motivated to exploit and affect the critical infrastructure of their adversaries. One simple motivation would be intelligence gathering on ARINC 429 bus data to try to reverse engineer and steal the technology behind the LRUs to artificially prop up their nation's economy. More sinister motivations could include targeted assassinations with plausible deniability.

The threat model that this project was designed around is the case of an attacker compromising a traditional IT asset, i.e. a maintenance laptop from the airline company. Because of the inherent trust of ARINC 429, the attack model is short and simple. The control of the laptop propagates to control of the Flight Management Computer where the hacker inserts a logic bomb to compromise and control engine controls. Using the bus is the easiest part of the attack since the attacker can use ARINC 429 design documents to send valid commands on the buses to the engines.

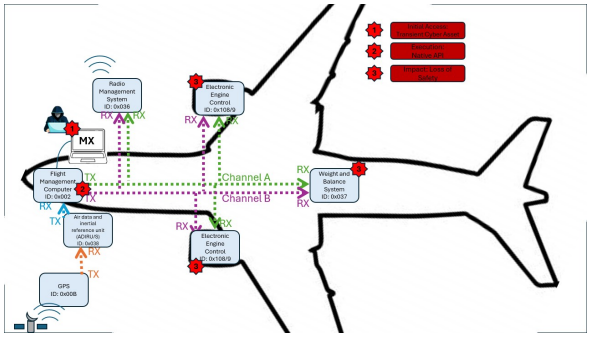


Figure 2: Project threat model.

The specific attack that I created in this project was that of a person-of-interest capture. The attack would result in the physical capture of a political prisoner who sought asylum status in another country. In my attack, this example person would be outspoken against the Iranian government and is a refugee now living in Germany. They are taking a flight from Frankfurt to India. This flight is based on the real-world flight on July 2nd, 2024[7]. This flight passes over Iran. In this case, the logic bomb would go off inside Iranian airspace. It would cause the plan to dip twice, forcing the captain and first officer to call a mayday, and land at the nearest airport, which would be inside Iran. From there, the authorities can apprehend and reestablish oppresive control over the refugee.

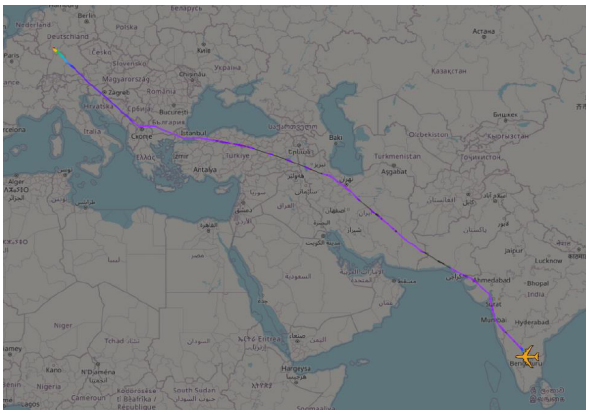


Figure 3: The flight path of Airplane DLH745 Leg 2 on 2 July 2024.[7]

The logic bomb would be a simple algorithm. If the plane is within a tolerance of a certain latitude and longitude (to identify that it entered Iranian airspace), then the bomb would send dive words forcing the plane into a downward dive only just briefly. This emulates Quantas Flight 72, where the plane dove because of a faulty Air Data and Inertial Reference System (ADIRS), and the first officer declared a mayday, resulting in an early landing[8]. The attack in this project would not force a plane crash.

3 Simulation

3.1 Emulating Specifications of ARINC 429 / Mark 33 DITS

The entire simulator was built from the ground up emulating the specification document, "ARINC Specification 429 Part 1-17: Mark 33

[7] Airplane DLH754, Leg 2, https://globe.adsbexchange.com/?icao=3c4b35&lat=40.621&lon=52.094&zoom=4.7&showTrace=2024-07-02&leg=2

[8] https://www.atsb.gov.au/publications/investigation_reports/2008/aair/ao-2008-070

– Digital Information Transfer System (DITS)". Each class helps build upon the previous one that came after it. This starts first by going all the way down to simulating the voltage on the wire.

ARINC 429 Bus Architecture is a bus communication standard primarily used in commercial aircraft.[9] One 'channel' has one and only one transmitting device and up to 20 listening devices for a max of 21 devices allowable on the bus. The transmitting device (often an LRU) sends 32-bit words as high and low voltages across the bus, which are then received by every other LRU on the bus. A '1' is designated by +10 volts on the bus for a unit of time and a '0' is respectively -10 volts. Zero volts on the wire indicates either the bus is in between bits or words, or there is no transmission of any words. The length of a bit is determined by the designation of a bus as either high or low speed. The high speed bus has transmission rates of 100kHz while a low speed bus has a transmission rate of 12.5kHz. This translates to one bit taking on the order of 10 μsec (give or take) for the high speed bus while a low speed bus is around 40 to 44 μsec. There exist devices that speed up or slow down bus speeds by 33% as well[10].

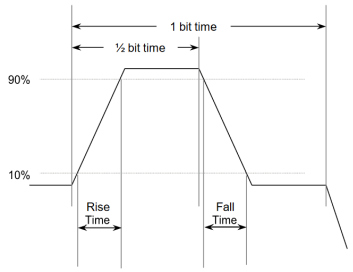


Figure 4: A graph specification for the '1' bit.[9]

In my simulation, the voltages are handled by the class `arinc429_voltage_sim.py`. This class creates voltages for high or low speed buses at every $\frac{1}{2}$ μsec, thus simulating the LRU sampling the voltage on the wire at each half microsecond. Also in this class is the tolerance that the transmitting LRU can sent voltages within tolerance. For example, a 1 is +10.0 V $\pm$ 1.0 V. Similarly, 0.0 V $\pm$ 0.5 V and -10.0 V $\pm$ 1.0 V for the null and low state of the bit respectively. Receiving bits has higher, but similar tolerances for voltages.

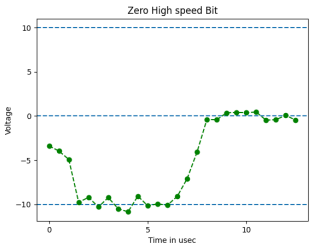


Figure 5: Voltage output samples for a '0' bit at high speed.

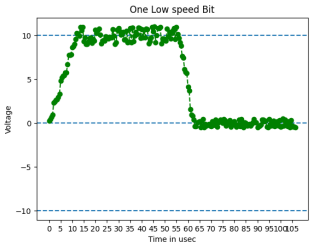


Figure 6: Voltage output samples for a '1' bit at low speed.

As show in figure 4, half the time of a bit is spent rising (or falling) to its respective state 1 or 0 and the other half is spent returning to and staying in the null state in preparation for the next bit. Each quarter stage in the standard is represented in the outputs shown in figures 2 and 3. From there, the aforementioned voltage simulator class can string together the desired 32-bits to form a typical word.

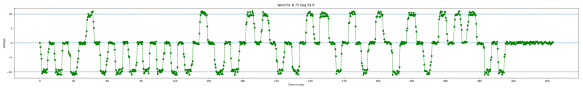


Figure 7: A word representing the latitude N 75 Deg 59.9', high speed.

Additionally, to complete a word, the voltage simulator adds 4 bit-times respective to the bus speed. As per the standard, 4 bit-times of null state must be in between words. This can be seen in both generated words in figures 7 (above) and 8 (in appendix).

Creation of word bits is handled by each respective transmitting LRU simu-

[9]The source for the information in section 3 is here: `https://www.aim-online.com/wp-content/uploads/2019/07/aim-tutorial-oview429-190712-u.pdf`, unless noted otherwise.

[10]See the product video here: `https://www.ueidaq.com/arinc-429-tutorial-reference-guide`

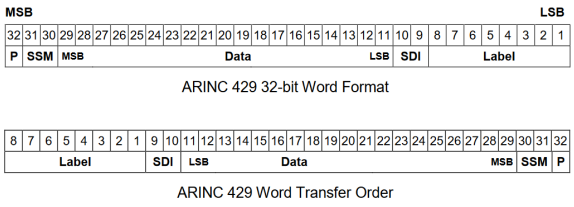


Figure 9: ARINC 429 word format and bit transfer order[9].

lation class (in this project that would be the `LRU_FMC_Simulator.py`, `LRU_GPS_Simulator.py`, and `LRU_ADIRU_Simulator.py` classes). Generally speaking, a word is broken into the following components:

- Label: this denotes the words function. For example, a label of 010_8, often written as $0o010$[11] is always for "Present Position - Latitude". However, in most cases labels have more than one use depending on the receiving LRU.
- Source/Destination Identifier (SDI): denotes which source is transmitting the data (in the event that an LRU is receiving on multiple buses) or which receiver the data is meant for (in the event that multiple LRUs are on the bus).
- Data: self - explanatory.
- Sign/Status Matrix (SSM): This depends on the label, and can indicate the sign (positive: $+$ or negative: $-$) of the data or statuses of the bus, etc.
- Parity bit: this bit is an integrity check (the only of the standard) where it's set to 1 if there are an odd number of 1s in the word and 0 if there are an even number of 1s in the word. The parity bit does not count itself.

Finally, moving onto the last part of the specification necessary to understand the project is the types of data encoding. Each LRU simulator class deals with this as they are either encoding the data when sending it, or decoding the data when receiving it. There are 4 types:

- Binary (BNR): Fraction binary conversion data.
- Binary Coded Decimal (BCD): Bit chunked place-by-place representation.
- Discrete Data (DISC): Combination of the previous 2 and/or individual bit-by-bit flag representation.
- System Address Label (SAL): Specific Subsystem Labelling on words.

In BNR, up to bits 11 to 28 (sometimes 29) of the word can be used, and the SSM often represents if the encoded data is positive or negative. There is also a resolution of the data that acts as a fractional multiplier, which helps to encode floating point numbers. This resolution is nowhere in the word itself, only in the spec and implimentation. BNR is somewhat similar to the IEEE standard for floating point but not exactly the same. BNR has a range, number of significant bits, and resolution. The range is the values that the data can fall within, the significant bits is the number of bits between 11 to 28 that are used to represent the data, and the resolution is the lowest place value that the data can represent. In the example of label 0o052 for "Body Pitch Acceleration in Deg/Sec^2: range = $[-64.0, +64.0]$, with 15 significant bits, and a resolution of 0.002. So, this data can only be within ± 64, only uses bits 28 to 13 for the data and can only represent down to the nearest 500^{th} Deg/Sec^2. To encode an example 32.0 Deg/Sec^2, you divide it by the fractional resolution normalized to the 1's place (i.e. 2), and then encode the resulting digits, including places down to the resolution to binary:

$$32.0 \div 2 = 16.0$$

$$\text{bin}(16000) = 0\text{bPPP}11111010000000$$

The 'P' is the unused sig bits. Data is also sent reverse order, and the SSM must also show the value is positive in this case:

$$0\text{b}00011111010000000 \rightarrow 0\text{b}00000001011111000$$

$$\text{SSM} = 000$$

$$\text{SDI} = 00 \text{ (example)}$$

$$\text{Label} = 0\text{x}052 = 0\text{b}00101010$$

The resulting word would be:
$0\text{b}\|01010100\|00\|00000001011111000\|000\|1\|$,

[11] Henceforth, like in ARINC 429's specification, this paper will also use 0oXXX as standard octal notation.

(splitting the word up into each section). Recall from figure 8 that labels are sent in reverse order (bit 8 to 1). A general BNR encode function was necessary to create as part of the IDS in figure 10 in the appendix. Additionally, each LRU in the simulation that handles BNR data uses a similar function to encode their data. It may be slightly tweaked depending on any exceptions.

In BCD, each digit/place of the word is designated to every few bits. The decimal point is decided by the range for the data, designated by the label. Ignoring exceptions where the placement of data-to-digits is slightly different, the least significant digit is bits 11 to 14, the next digit is bits 15 to 18, similarly onward with chunks of bits 19 to 22, 23 to 26 and finally bits 27 to 29 as the most significant place. As an example:

- Label: 0o232
- Representing: Altitude Rate in Ft/Min.
- Range: $[-20000, +20000]$
- Resolution: 0.1

In Example: -16342 ft:

- Bits 11-14: $2 \rightarrow$ 0b0010
- Bits 15-18: $4 \rightarrow$ 0b0100
- Bits 19-22: $3 \rightarrow$ 0b0011
- Bits 23-26: $6 \rightarrow$ 0b0110
- Bits 27-29: $1 \rightarrow$ 0b001

Reversing each digit (as per spec) and adding the SSM=0b11 to signify the negative value nets a data section of 0b...$\|$0100$\|$0010$\|$1100$\|$0110$\|$100$\|$11.
Finishing the word:
0b$\|$01011001$\|$00$\|$01000010110001101 00$\|$11$\|$1.
Similarly, I implemented a general BCD encode function, in figure 11 in the appendix.

Discrete data is handled on a case-by-case basis in the sim since every encoding is so diverse from each other. SAL data is given a separate subsection for the SAL code immediately following the label, and then the remaining data is left to the user of the sim to handle and implement since it is proprietary and not included in the specification.

In addition to the functionality above, `arinc429_voltage_sim.py` has:

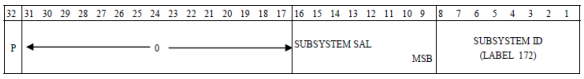


Figure 12: General SAL word format.

- functions that can create random words.
- a function that turns an integer in the range of $[0, 4294967295]$ into a word, where 4294967295 is the upper limit on what a word can be as it's $2^{32} - 1$ or 32 1s in binary.
- a functions to turn a bitstring into a word, similarly it must be between "0b00...00" to "0b11...11" and of length 32.
- a function to graph words.

3.2 The Bus, and creating an LRU

The bus in the simulation is its own class, `BusQueue_Simulator.py`. This bus class uses the previous `arinc429_voltage_sim.py` class. It creates a shared queue object, which represents the bus, for all the LRUs that use it. Therefore, this object must be declared and used by all the LRUs on it. The queue has a length of 100 and has the 100 most recent voltage samples generated by the voltage simulator portion. They get added to the bus as index 0 in the queue and then pushed further down as another voltage is added, hence why a queue is used. This, as done in the `add_voltage` function, is how it simulates the voltage running down a wire. Meanwhile, receiver LRUs use the `get_voltage` function to 'sample' a voltage from the wire, which grabs a voltage at the end of the queue, which is always index 100. If the queue doesn't have a voltage sample in the queue at index 100, it generates a voltage in the null state, i.e. ≈ 0.0 V. This class is speed agnostic, so transceivers and receivers can send and receive voltage on this simulated bus as fast or as slow as python lets them. Therefore when designing a specific bus the whole sim needs to be calibrated to one speed. Also, to help simulate noise on the bus, there is a function, `simulate_EMI_noise_interference` that gets a few random voltages in the queue and adds or subtracts a small amount from those voltages. This function helps replicate real-world conditions to make the simulator more accurate but is optional. Finally, the bus has a `queue_visual`

function that graphs the bus voltage in real time for testing and debugging purposes.

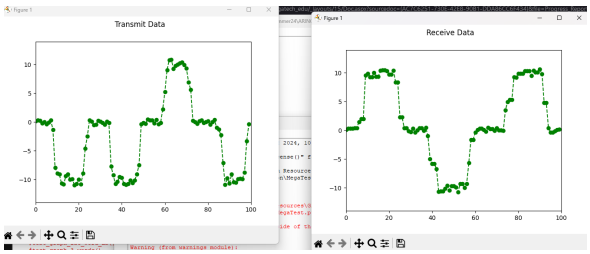


Figure 13: A snapshot of a transmitting (left) and receiving LRU (right) communicating using `BusQueue_Simulator.py` class.

This leads us to the `LRU_RX_Helper.py` and `LRU_TX_Helper.py` classes. These 2 sibling classes can be thought of as the TX/RX chips as part of the LRUs that attach to the ARINC 429 buses and interface between the voltage and the code to handle words[12]. These classes contain common functions that every LRU will use for example functions that help LRUs communicate with the bus. Beginning with the `LRU_RX_Helper.py` class, it has a `bus_speed` attribute which is either high or low given as a string, and a BusQueue object list for the bus channels that the LRU is on. It has a function, `receive_given_word` that can decode the sent voltages on the wire (i.e. BusQueue) into a 32-bit word, at a specified rate. Additionally, it validates words based on their parity and can also plot the sampled voltages in a matplotlib graph in real time. Importantly, it also has a function that gets a label from a given word, i.e., given some word 0b10110100...100, it returns the label 0o055 as 0b00 0b101 0b101 is 0, 5, and 5.

Its sister class, `LRU_TX_Helper.py` helps an LRU transmit to a BusQueue object. It can transmit random voltages as part of testing (`transmit_random_voltages`), given a word, transmit it (`transmit_given_word`), validate words before they are sent (`validate_word`), similarly visual the bus voltages (`visualize_LRU_transmissions`), and given the label as an int, encode that label to a binary int and bitstring (`make_label_for_word`).

[12]E.G. the type of ARINC 429 communicator chips sold commercially; `https://content.instructables.com/FP6/557Q/GMJT3SUG/FP6557QGMJT3SUG.pdf`

Finally, with all those previous classes as building blocks I moved on to creating LRUs themselves. These are the meat of the simulator and can be as diverse as desired. I started by designing each LRUs specific functionality. The `LRU_GPS_Simulator.py` is the class that runs the Global Positioning System LRU. For its attributes, it has a bus_speed, channel, a starting latitude and longitude, and a communicator_chip that is the `LRU_TX_Helper.py` class. It's a simple LRU that determines the plane's position from input flight data (i.e. text files) or barring no flight data determines the next latitude and longitude from where the plane would be if just flying straight forward from the previous one. It then encodes the latitude and longitude to words and transmits those to the Orange Bus in figure one above.

The next LRU is the ADIRU, which is `LRU_ADIRU_Simulator.py`. This is also sometimes referred to as ADIRS. It has similar attributes as the GPS except in addition to a receiver chip instance it also has a TX Chip (`LRU_TX_Helper.py` object instance), and a data dictionary that includes a variety of air/inertial data such as Latitude and Longitude, Altitude (of which there are many different types), ground speed, heading, Air Speed, Roll Angle, Pitch Angle, etc. It decodes the words from the GPS (`decode_GPS_word`) and sets those as latitude and longitude. For the rest of the data, it uses the input text file flight data to set values in in the data dictionary, encode that data (`encode_word`) as words and send those words over the Blue Bus. The amount of data types it has to encode is the most numerous of all the classes.

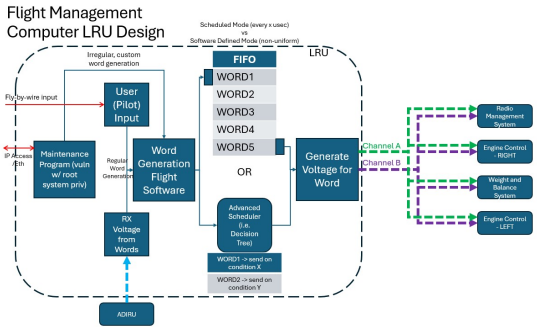


Figure 14: Graphical design blueprint for the FMC

The Flight Management Computer (FMC) handles actually flying the plane in the simulator. Figure 14 is the initial design for it. It can take

in data from the maintenance program, user input, and words from the ADIRU. It decodes the words from the ADIRU and uses those data points to keep the plane stable in the `decodeADIRUword` function. Using the previous word send by the ADIRU, if the flight data that the words contain show that the plane is ascending, for example, then it creates and sends a word to the engines and weight and balance system to move the plane accordingly. For example, if it decodes the altitude word as having data larger than the previous altitude, it makes and generates an ascending word. Additionally, for the simulation user to mimic the for pilot's input, they can use the arrow keys to generate and send words that move the plane their respect directions. For example, the right arrow key will make a word that commands the plane to turn to the right. The 'w' and 's' keys are used to push the plane forward and backwards respectively, covering all 6 directions of freedom in flight that a plane can go. This project is only an ARINC 429 simulator and includes a separate flight simulator class, but it only generates words to move the plane and does not interface between the words and flight simulation besides just print statements saying the direction of the plane. Separate from the code of the FMC, but symbolically connected, is the maintenance program, as detailed in `maintenance_program.c` and `mxProgram.exe` which take input from the maintainer (or attacker) and output words. Together they compose the FMC part of the sim. The FMC sends words periodically after a set amount of time when in scheduled mode, or if set to FIFO mode sends words as it gets them from a short queue of 8.

The Weight and Balance System (W&BS); `LRU_WnBS_Simulator.py`, and Full Authority Engine Control (FAEC); `LRU_FAEC_Simulator.py`, take words from the FMC and print the direction that the word told it go. They also handle a few other decoding mechanisms from words such as the serial number for the engines. They receive the words over the Green and Purple buses for redundancy.

The final LRU I created in this project is the Radio Management System (RMS); `LRU_RMS_Simulator.py` class. It is the LRU that chirps all the radio signals, such as ADS-B, for example. It can set the frequency of each radio (i.e. ADF, ILS, DME, etc.), and 'sends' (prints) a dictionary of its radio values. Airplanes must turn on their ADS-B radios and send various data points for air traffic control to be able to discern where airplanes are and to land them safely. The ADS-B message has the flight number, latitude, longitude, altitude, ground speed, vertical speed, track angle, magnetic heading, status, Ident switch on or off, ICAO airplane address, and aircraft type (default to civilian in this simulation). It also receives information and decodes words from the FMC on the Green and Purple buses.

3.3 Completing the Model

Finally, in the main class, `main.py`, I put all the LRUs and voltage, TX/RX and BusQueue simulator classes together to create the model shown in figure one. The first thing the main function does is load all the flight data from the flight in figure 3. Then it runs through all the flight data with the ADIRU, GPS classes, and the FMC classes which all send words to each other, encoding and decoding words, and then send words to the RMS, FAECs, and W&BS. This main class uses python multi-threading to send and receive each word with the BusQueue class. This completes the simulation.

I created `main_noThreads.py` as well where words are passes directly (and thus robustness is assumed) because of python limitations I will explain further in section 7. Otherwise, this class does all the same as `main.py` except, it adds the IDS from section 5, and adds the attack proof of concept in, making it the more complete demonstration of the project. Each word is printed to screen in its bus channel color (orange, blue, green, purple) and writes alerts/logs from the IDS to the same sub-folder structure. Running the class gives you the option of whether or not to use the IDS, set the word send rate, and whether or not to execute the attack. It also then plots the amount of alerts and logs that the IDS alerted, if applicable. The total amount of words sent in these main classes is 432256 with the attack and 430656 without. Note that even with just 7 LRUs, 4 channels, and a medium/long length flight of approximately 8 and a half hours, there will be over 400,000 words generated. A typical flight may well generate millions of words. Obviously, (thinking about the next sections), the IDS will have to be able to handle a large amount of words.

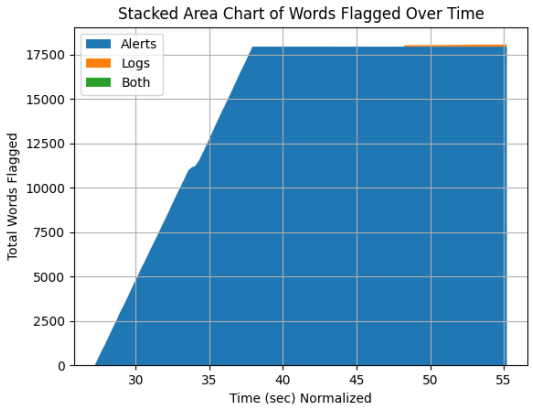
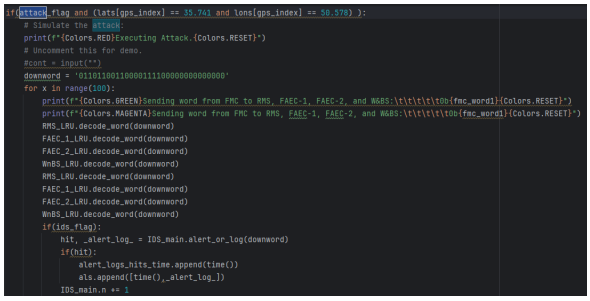


Figure 15: A graph of the alerts (blue) and logs (orange) generated from the IDS showing the attack present, from `main_noThreads.py`.

3.4 Attack

The attack proof of concept (`flight_attack_poc.py`) uses the maintenance program (recall symbolically part of the FMC) to send 100 downward dive words. It's extremely simple since the attack itself is simple. Further, this is implemented in the `main_noThreads.py` class in figure 16. Note that because of the inherent trust of the ARINC 429 standard, it is exceedingly simple and short to implement. This is the implementation for the attack in section 2.2 as the latitude and longitude are roughly within where the airplane enters Iranian airspace.



```
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 = '01101100110000111100000000000'
    for x in range(100):
        print(f"{Colors.GREEN}Sending word from FMC to RMS, FAEC-1, FAEC-2, and WGBS:\t\t\t\t\t0b{fmc_word1}{Colors.RESET}")
        print(f"{Colors.MAGENTA}Sending word from FMC to RMS, FAEC-1, FAEC-2, and WGBS:\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`

From here, the aircraft would need to call a mayday and then force the airplane to land.

4 IDS Explanation

4.1 IDS Setup

The IDS solution developed uses a rules file to initially set up the IDS and tell it what words, data, etc. to alert and log on. Figures 17 and 18 in the appendix show the default rule file template that would accompany the IDS out of the box. In the rules file, the user can add comments to the file similar to python with the # symbol, and anything after the symbol is a comment. Comments can be on their own line or inline after valid syntax. There are 4 sections:

1. `!Outfiles`: specifies where in your file structure you want your alerts and logs files to be written to:
   `alerts = <given filepath>`
   `logs = <given filepath>`
   When the IDS boots up, it will only append these files and not overwrite them. The user would have to write a small wrapper to reset the contents as desired (this functionality is seen in `main_noThreads.py` and the evaluation python files).

2. `!Channel`: specifies the connections between LRUs on each channel. Each LRU must match one of the equipment names listed in the specification[13]. Each connection is in the form of `<Channel>: <LRU TX> -> <LRU RX>`. Each connection is in the form of `<Channel>: <LRU TX> -> <LRU RX>`. For the example of the model from figure one:
   `Orange: Global_Positioning_System -> ADIRS`
   `Blue: ADIRS -> Flight_Management_Computer`
   `Green: Flight_Management_Computer -> FADEC_Channel_A`
   and so fourth.

3. `!SDI`: specifies the SDI bits that represent each LRU in the form of `<Channel>: <LRU> -> BB` where 'B' is a 1 or 0. E.G., `Orange: ADIRS -> 00`

4. `!Rules`: constitute their own subsection regarding syntax. Generally, they tell the IDS to alert, log or both on specified data/bits,

[13] ARINC Specification 429 Part 1-17: Mark 33 – Digital Information Transfer System (DITS) pages 43-46

channels (in the case the IDS is receiving on multiple channels), a message to transcribe upon alerting, etc.

4.2 IDS Rules Syntax

Rules are split into multiple subsections split by spaces. First, the user must define if the rule will generate an alert or a log or both: as `alert`, `log`, or both: `alert\log or log\alert`. Alerts should be accompanied by a message and are for alerting an operator about a specifically important word. Logs on the other hand are meant for more routine types of words and traffic and just log the word into a file when it comes across the wire. Logs can be accompanied with a message as well.

The next section of the rule and other strict requirement is the channel to listen on for the word. It must match channels defined in the `!Channel` section. Those are the only 2 strictly mandatory values. Obviously defining an alert or log on just the channel will alert/log on every word in that channel if that's what the operator desires. Other values are weakly mandatory as well, in that they are necessary for another section.

A good example of a weakly mandatory section is the label section, written in its octal value from 0o000 to 0o377, which is only mandatory if the data section has a value to alert or log on. This is because data must be a certain unit (degrees, feet, knots, etc.) and certain encoding (BNR, BCD, DISC) which can only be known from the label. For example, label 0o010 encodes latitude data as BCD while label 0o310 encodes latitude data as BNR. So even when the data is the same value, the bits to alert on will be drastically different. If the syntax were designed to just specify units or the type of data to look for that would not be enough detail to tell specifically which words to alert and log on.

The SDI section then follows as the human readable name of the LRU you want to alert on the word going to. The IDS translates this into the bits defined in the SDI section, e.g. putting ADIRS would be translated into "00". This section must be an LRU defined in the `!Channel` or `!SDI` sections

Moving onto the data section, the user has the most variety of options. As noted above regarding labels, the user can alert on specific values for the data, granted it matches the label and LRU that can receeive that type of data. This would just be the simple form of `data:<value>`. One caveat for value searching; if the data is encoded DISC, then the user must specify the value as a binary since there is no standard on DISC encoding, that would be something like `data:10101...11` for 19 bits (length of data section). They can also display the percent chance that word has of being encoded as BNR, BCD, DISC, or SAL if the label is known but the SDI is unknown: `data:<BNR\BCD\DISC\SAL>`. They can specify bits from index to index; `bits[start:stop)=10...01`. And they can alert on specific sub-SAL codes: `data:0o<SAL>`.

The next section is the SSM which is designated with the values 00, 01, 10, 11 as the SSM in the word. If the data section is specified as well, the SSM must match it. For example, negative data values would need to be either 01, 10, or 11 while positive values would need a SSM of 00.

The parity section then follows which is either "C" for words containing correct parity bit, or "I" for words containing an incorrect parity bit. If words start going on the bus with incorrect parity, that is a cause of concern either from a cyber defense perspective or LRU maintenance perspective.

The last 2 sections are "time" which, if included, add a timestamp to the alert/log, and message which is added to the alert/log.

The IDS will throw errors if either not formatted correctly or if necessary information is missing in the rules file, and will tell you what you need to fix. For example, if trying to search for specific data (such as for example latitude as 100 degrees) without the label, it will raise a value error, "Label needed in order to properly search word or given data:100". Handling all the rules from the rules is contained the `handle_ruleline` function. This function is run line by line on each line in the section under `!Rules` as part of the IDS boot-up.

The output from the function `handle_ruleline` is a 5-tuple that contains: alert or log or both to instruct the IDS what to do, bitmask generated to compare the word against from the Label, SDI, Data, and SSM sections, a parity boolean True if checking for correct parity, False if checking for incorrect parity, and None type object if not checking for parity, a time boolean True if to include a timestamp, False if not, and finally the message, if

any. These rules are all stored in the IDS object's `rules` list attribute, and referenced later when checking against each word.

Figure 19 below shows a rolodex of all the rule syntax options. If the option box has dashed outlines, that indicates that it is not strictly mandatory. The arrows show a reliance or a relationships from one section to which the section that the arrow points.

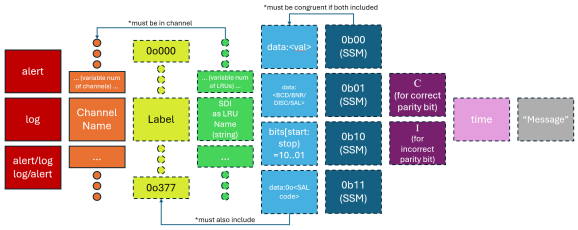


Figure 19: Visual rolodex of rules syntax options.

The user can use this visual rolodex to mentally plan out their rule. Starting from left to right, the user must include an option from the hard outlined boxes, and then can include or skip options from any of the dashed boxes rightward, as long as they maintain relative order. The rule can end any time after the channel section. Looking at an example from one of the evaluations' rules that was crafted to catch the attack in section 2.2 and 3.4:

```
alert Channel1 0o066
Flight_Control_Computer
bits[15:20)=1111 time ``FMC
Sending Spurious downward dive
words!!"
```

- `alert`→ tells the IDS to write the message "FMC Sending Spurious downward dive words!!" when the word is detected.
- `Channel1`→ tells the IDS to focus on Channel1.
- `0o066`→ tells the IDS to add 10110100 to the bitmask.
- `Flight_Control_Computer`→ tells the IDS to add 11 to the bitmask as the Flight_Control_Computer has SDI label 11.
- `bits[15:20)=1111`→ tells the IDS to add 1111 to the bitmask at position 15 to 19.
- SSM section is skipped.
- Parity section is skipped.
- `time`→ tells the IDS to add timestamps to the output.
- The message is covered above in the first item of this list.

The output tuple from this is: `(alert, 01101100110000111100000000000000, None, True, "FMC Sending Spurious downward dive words!!")`

4.3 IDS Workings

All the code for the IDS is in the single-class python files `ARINC429_IDS.py`. When the IDS starts, it opens the rules files and starts going through each section. It has 4 functions dedicated to parsing each of the sections above. Those functions modify the IDS' attributes so that it has a list of the LRUs and channels, as well as the aforementioned rules list. Also encoded into the IDS are 3 important dictionaries. The first is all of the SAL codes to their human readable name, so that the IDS can notate when it comes across those labels in a word. The next is a dictionary of all possible equipment IDs to their human readable names so that the rules can only contain those predefined LRUs. Finally, the most important pre-coded dictionary is the generic label dictionary. It has the label mapped to each possible encoding type and then the range and resolution for those encodings. Label 0o157 is a good example since it has all of the available encoding types:

- `0x01c: ["DISC", 0.0],`
- `0x027: ["DISC", 0.0],`
- `0x033: ["DISC", 0.0],`
- `0x04d: ["DISC", 0.0],`
- `0x055: ["BNR", 1.0, (0.0, 20000.0), 15],`
- `0x0bb: ["DISC", 0.0],`
- `0x10a: ["DISC", 0.0],`
- `0x10b: ["DISC", 0.0],`
- `0x114: ["BCD", 1.0, (0.0, 400.0)],`
- `0o157: ["SAL", 0.0]`

The responsibility of this sub-dictionary is help encode a bitmask for each rule. When a rule specifies a label and data section, the information above is enough to generally encode that data to bits, from bits 11 to 19, which is generally the range of the data section. In the case of DISC data, the encoding is supplied by the rule, hence why there is a placeholder 0.0. Similarly, for SAL data, the encoding is just the next 8 bits following the label in the word, which is the system address label itself. For BCD data, the value (as an int or float) is then converted by the BCD encode function described in section 3.1. The list above, `["BCD", 1.0, (0.0, 400.0)]` denotes that for equipment ID 0x114, the data is encoded as BCD with resolution 1.0 and a range of 0.0 to 400.0. Similarly for BNR data, the value is encoded using the general BNR encode function described in section 3.1. In `["BNR", 1.0, (0.0, 20000.0), 15]`, the 1.0 is the resolution of the data, the range again is 0.0 to 20000.0, and the significant digits are 15. From those 2 functions, and a similar `SAL_encode` function. As shown in section 4.3, The `handle_ruleline` function makes heavy use of these dictionaries and encoding functions. It turns the rule in the rules file into a queue by splitting it up into sections, and then handles each section.

To alert or log on a file, the IDS first receives the voltages from the BusQueue object, and then translates that into a word, using the `LRU_RX_Helper.py` class. From there the word is passed to the `alert_or_log` function (see figure 20 in appendix). This function goes through the list of 5-tuples, rules, in the IDS and checks if the word alerts or logs on each rule. The first element obviously tells the function to write to the alert or log file or both. The bitmask is then XOR'ed section by section with the word and checking to see if XOR'd together they equal zero. Any integer XOR itself equals zero. If it does, then it checks for parity, timestamp, and message if applicable and writes to the alert/log-file. The XOR operation is $\mathcal{O}(1)$ complexity as opposed to checking bit-by-bit in a bitstring which is $\mathcal{O}(31)$.

5 Project and IDS Evaluation

The evaluation plan for this project has 5 components: Simulation Correctness, IDS Robustness, IDS Correctness, IDS Detection (of the attack), and the time added by the IDS. To start, the Simulation Correctness was all built in the file `MegaTest.py` where each function of the LRU and their sub-classes had a few test functions to check functionality. It is a large code base that was purely made for testing function after function. This file was also used to test the IDS as well. An example of a testing function for the IDS in `MegaTest.py` can be seen in figure 21 in the appendix.

I tested test IDS Robustness by seeing how well and fast it could handle getting words from voltages on the wire. The test `IDS_Eval1.py` receives 5 predefined words from the BusQueue object. This test runs starting at speeds starting from 0.5 sec sampling rate up to $\frac{1}{2}\mu$sec on zero to 10 rules per sampling rate, showing which words it was able to correctly receive from the bus. Figure 22 is a table of the results. Because this test was varying the speeds of the bus, it effectively tests how robust the IDS is. Unfortunately, because of a limitation in python multi-threading, it was not able to achieve true sampling speeds of sending and receiving voltages every $\frac{1}{2}\mu$sec. While it got close, when testing at $\frac{1}{2}\mu$sec, and 10% slower, the bus had a tendency to carry over a bit to the next word because it couldn't keep up, and thus getting each subsequent word wrong.

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`.

To test IDS Correctness, I passed words directly to the IDS (which assumes robustness) and then barraged it with 100s of millions of words based off of more real-world flight data[14]. By sending a lot of words, I wanted to make sure the IDS would be able to correctly alert and log on words specified in its rules file over a prolonged period of time. The rule file (`Eval2_Rules.txt` in the IDS_EVAL2_RULES_FILES folder) alerted and

[14]To get enough data I had to use the open dataset hosted by NASA here: `https://c3.ndc.nasa.gov/dashlink/resources/664/`

logged on both specific and general criteria. For example, the IDS was set to alert on 41.88 degrees latitude and on an Indicated Angle of Attack of 1.01 degrees. A more general criterion was alerting on latitude words (ADIRS LRU with a label of 0o311). Calculating the expected amount of words from the dataset for alerts I got:

$$50 \text{ latitude words for } 41.88 \text{ degrees} +$$

$$11,977,472 \text{ BCD encoding words} +$$

$$748,552 \text{ GPS Longitude words} +$$

$$0 \text{ Angle of Attack } 70.45 \text{ degrees words}$$

$$= 12726074 \text{ alerts}$$

And for logs, it should get 9623 logs for the Indicated Angle of Attack.

```
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`.

Additionally, figures 27 and 28 show the number of alerts and logs collected over time. Since it was a pretty even distribution a nearly flat makes sense. The test for IDS Correctness was a success.

To test the IDS's ability to detect attacks, I took the flight data from the attack (section 2.2), ran the attack and then saw if the IDS was able to show an operator something out of the ordinary. This is back to the first set of real-world flight data as described in the attack sections above. Figures 30 through 34 in the appendix show the results. In particular figure 34 is a good example of how this test was a success. Since the blue spiked while the orange didn't the graph shows a large number of

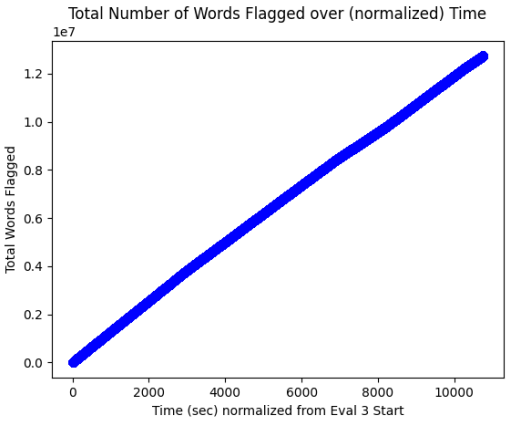


Figure 27: Number of alerted/logged words until that point over time for `IDS_Eval2.py`.

words alerting the IDS for a cyber-induced downward dive. In this case, because the IDS is alerting the operators of the aircraft that they are under a cyber-attack they can make a more informed decision about the situation and decide not to land in Iran, but rather a more friendly country for help.

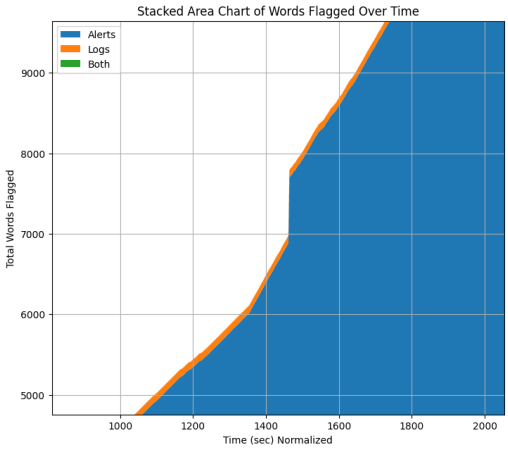


Figure 34: Zoomed in cumulative graph for `IDS_Eval3.py`. Note that the jump is purely blue which means it's only alerts–for the attack. No logs are generated.

Finally, for the last IDS test, I wanted to see how much time the IDS would add to the airplane's operations. Specifically, I wanted to see how long it took to boot-up and how much time the IDS took to evaluate a single word. For the first metric, if the IDS took a long time to set up, then that would slow down an airplane's ability to take off from the runway and cause

an overall slowdown of operations and loss of money. For the time it took to alert on a single word, if the IDS took too long to decide on whether or not to alert/log on a word then it would quickly get overwhelmed by a backlog of words to check, and be in degraded operation. The test code, `IDS_EVAL4.py` generates a random rules file based on varying the number of channels, LRUs and rules. From each of those rules files, distinct from each other in at least on category (channel number, LRU number, or rule number), it then checked the amount of time that the IDS took to set itself up and it took to alert on one word: 01010101010101010101010101010101. From there it gathered the data together (see eval4_data.csv in the IDS_EVAL4_RULES_FILES folder).

The specific numbers by which the test varied were:

1. the number of channels; either 36 or 48 - the most common for civilian airliners,

2. the number of LRUs on a channel from the minimum of 2 to the maximum of 21,

3. the number of rules; from 1 to 500 rules.

This rounds out to an even 20,000 test cases. I could have gone higher on the cap for the number of rules, but at 500, the test was starting to take a few hours on my hardware. What I found was that the IDS does generally take longer to boot up as well as to decide whether to alert or log on a word, but at negligible margins, and sometimes increasing a parameter wouldn't make a difference. By running the data visual tool for the .csv file, `Eval4_Data_Visual.py`, the user can generate many different graphs showing how much the time went up by as more rules, channels, and LRUs were added to the IDS's rules file. In the example of 36 buses adn 15 LRUs as the fixed parameters and 1 to 500 rules, the IDS set up time goes up from 0.01 seconds to about 0.04 seconds, and the time it takes to alert on one word goes up from 0.0 seconds to 0.002 seconds. If we fix the channel count at 36 and the number of rules at 300 and vary the number of LRUs from 2 to 21, with the exception of some outliers, there is no discernible increase in alert/log time one word, and a very slight increase on the time it takes to set up the IDS from 0.028 seconds to about 0.031 seconds. All in all, this test was a success because all of these times are negligible for airplane operation. For more examples, see figures 35 to 43.

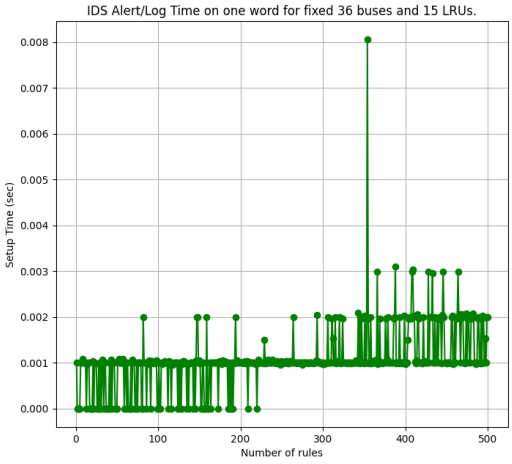


Figure 34: `IDS_Eval4.py`: Time to process 1 word for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

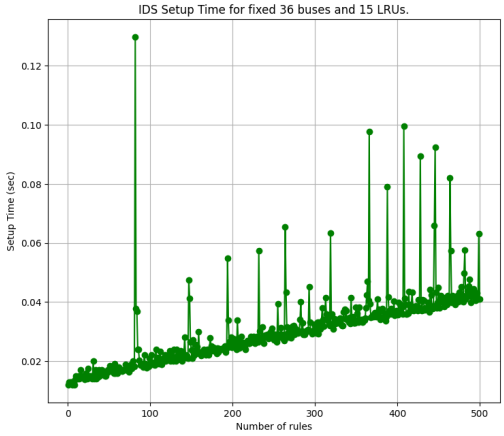


Figure 35: `IDS_Eval4.py`: Time to boot-up the IDS for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

6 Limitations

The solution I developed does have some limitations. First, for a typical B737, the model I made is simple compared to a real airplane. The model for the simulation only has 4 channels and 7 LRUs where as a typical B373 will have many more channels (around a couple dozen)

and LRUs, as well as having different types of avionics bus communications. Additionally, for the majority of the project, my simulation was struggling with matching real-world timing, and still does because most of the project is coded in python, which is very limited and slow for multi-threading processes; this kneecapped me deep into the project such that I couldn't run the BusQueue object at real-world sampling rate speeds. To course correct, for many of the files, tests, and in `main_noThreads.py` (as a demonstration of the difference) I had to 'assume' robustness and send words directly as they were generated by LRUs to each other.

Furthermore, the IDS is useless without the right know-how on its syntax and usage. If the plane operator doesn't know how to set rules correctly, they won't be able to start the IDS, and even if they do, if they don't make substantial rules that are able to detect abnormalities and instead create benign rules then the IDS loses functionality as well. Training operators (maintenance technicians, flight attendants, and pilots) on how to use the IDS and what to do with generated alerts and logs may also be too high a monetary cost for many airlines.

The IDS itself also cannot, as defined in scope of the model, stop attacks. For some circumstances where pilots lose control due to a cyber attack and the intent is not as lenient as described for this project's threat model, the IDS has no way to stop the cyber attack, only alert that it's happening. Additionally, this project does not move ARINC 429 to 100% ZTA. As stated in the introduction, the IDS helps secure communication by alerting on intrusion, and the rules act as a dynamic policy for operators for channel traffic, but to keep robustness of the plan, the IDS cannot help with tenet 6, for example. Finally, because of the high financial cost of ARINC 429 computers and interface chips, I was unable to test the IDS on real hardware.

7 Lessons Learned

The number one lesson learned was to do more research into the tools you'll select for the project. Because I chose to write everything in Python and not a better language for multi-threading, like C, I stunted the overall performance of the BusQueue class when trying to reach parity with real-world bus speeds. Additionally, I sunk a lot of time into hard coding many edge cases for word encoding and word generation for each LRU. The IDS itself has every variation of how any word can be encoded. If I had started there instead of taking up that task midway to near the end of the project, I would have saved a lot of time developing the LRUs more succinctly when they generate their own words.

Finally, another lesson learned was that for some technologies it's not possible to fully move them to ZTA, nor does it really make sense. In the example of ARINC 429, and tenet 6 of NIST's Zero Trust Framework, "All resource authentication and authorization are dynamic and strictly enforced before access is allowed."[1], this tenet doesn't fully make sense. LRUs don't need to authenticate with other LRUs because they are only talking unidirectionally on a co-physical location. They are not talking across planes. Additionally, requiring LRUs to authenticate to each other would slow down the bus to the point of uselessness.

8 Future Work

There are many more things that can be done with this project. The first would be expanding the model and simulation with LRUs and channels to be a more accurate representation of a real-world plane like the B737. Re-coding the project in a better language as well could be a good endeavor to help the BusQueue object instances. Additionally, expanding the logic within the IDS such that it has condition chains between alert words could help in further nailing down cyber attacks. In the case of `IDS_Eval3.py`, the cyber attack was suspicious because the FMC was sending downward dive words but the ADIRU was not changing anything about the indicated angle of attack. This is visible when reading the alerts, but would have to be interpreted by the operator of the IDS mid-air.

Also, the IDS could be converted into an IPS in the future if tweaked such that it alerted more mid word and then flipped voltages of the parity bit causing the rest of the LRUs on the bus to drop the word. This would have to necessitate more training for the operators of the IDS such that they don't accidentally craft a word that takes away positive control from the pilots. As stated in the limitations, testing and interfacing the IDS with real hardware would also be a good next step.

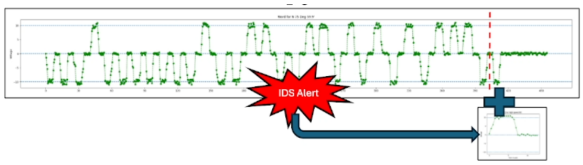


Figure 44: How to change the IDS into an IPS.

Additionally, airplanes don't collect word data, only some inertial and reference data from the ADIRUs/ADIRS. Collecting real-world data from real planes and then generating attacks on that data would be a good proof of concept to create. With that, one could theoretically implement a machine learning system that detects malicious words.

Finally, diversifying the threat model would also be a good next step. In the research for this project, I found one other ARINC 429 security research paper in which their threat model was a rogue LRU vampire attack[15]. There are many other avenues an attacker could take than shoving a random LRU onto the bus, or in the case of my model, propagating from maintenance. For example, testing to see if one could navigate from in flight media Wi-Fi or entertainment systems to an ARINC 429 bus, or if from radio signals as through the Radio Management System. Some of the SAL codes are indeed for in-flight entertainment systems hinting at some proprietary connection to ARINC 529 buses; i.e. The Audio Entertainment System (AES) as SAL 0o364. Exploring ways onto the bus would be a good research endeavor in it of itself.

9 Conclusion

In this project, I developed a comprehensive simulation for the ARINC 429 bus architecture to try to enhance its security by integrating zero trust principles. The core components of this project included a detailed simulation of the ARINC 429 bus, a proof of concept for a potential cyber attack, and the implementation of an IDS.

My findings highlighted several key insights:

1. Simulation: The ARINC 429 bus simulation provided a framework for testing and replicating bus architecture upon which I could develop an attack and IDS.
2. Attack PoC: The attack demonstrated the ease and feasibility of a cyber attack on the ARINC 429 bus, and how since the bus does not record word data would leave no record if not for the IDS.
3. IDS: While the IDS cannot prevent attacks, it can serve as a crucial first line of defense by providing alerts to in-flight operators, allowing them to take necessary actions to mitigate potential risks.

While this project does not achieve complete ZTA for the ARINC 429 bus, it makes the first strides toward enhancing its security. By addressing the identified limitations and pursuing the recommended future work, users in the aviation industry can move closer to ensuring the safety and security of one of its critical communication technologies: ARINC 429.

[15] Hardware Fingerprinting for the ARINC 429 Avionic Bus

References

R. Vincent. 28 Nov. 2023, *Arinc-429 RX Implementation in LabVIEW FPGA.*, NI Community. https://forums.ni.com/t5/Example-Code/Arinc-429-Rx-Implementation-in-LabVIEW-FPGA/ta-p/3507624

aeroneous. 17 Jul. 2018, *PyARINC429.*, Discover PyARINC429, a simple Python module for encoding and decoding ARINC 429 digital information. https://github.com/aeroneous/PyARINC429

Peña, Lisa and Shipman, Maggie. Feb. 2024, *Episode 64: Zero-Trust Cybersecurity for Vehicles.*, Technology Today Podcast, Southwest Research Institute, https://www.swri.org/podcast/ep64

Sital Technology. Accessed May 2024, *ARINC-429 with Cyber and Wirefault Protection.*, ARINC-429 Solutions. Sital Technology, https://sitaltech.com/arinc-429/

Sital Technology. Accessed May 2024, *Understanding Cyber Attacks on MIL-STD-1553 Buses*, Sital Technology, https://sitaltech.com/understanding-cyber-attacks-on-mil-std-1553-buses/

Alta Data Technologies LLC. 19 Jan. 2021, *1553 Network and Cybersecurity Testing*, Alta Data Technologies LLC, https://www.altadt.com/wp-content/uploads/dlm_uploads/2020/10/1553-Network-and-Cybersecurity-Testing.pdf

Tilman, Bill. 14 Dec. 2021, *Why You Need to Secure Your 1553 MIL-STD Bus and the Five Things You Must Have in Your Solution.*, Abaco Systems, https://abaco.com/blog/why-you-need-secure-your-1553-mil-std-bus-and-five-things-you-must-have-your-solution

Waldmann, B. Jul. 2019, *ARINC 429 Specification Tutorial.*, Avionics Databus Solutions, Version 2.2, AIM Worldwide, https://www.aim-online.com/wp-content/uploads/2019/07/aim-tutorial-oview429-190712-u.pdf, https://www.aim-online.com/products-overview/tutorials/arinc-429-tutorial/

KIMDU. 26 Jun. 2023, *ARINC-429 tutorial: A Step-by-Step Guide.*, KIMDU Technologies, https://kimdu.com/arinc-429-tutorial-a-step-by-step-guide/

United Electronic Industries/AMETEK. 26 Jun. 2023, *ARINC-429 Tutorial & Reference.*, Understanding ARINC-429, United Electronic Industries/AMETEK, https://www.ueidaq.com/arinc-429-tutorial-reference-guide

Biden, Joesph R. Jr. 12 May 2021, *Executive Order on Improving the Nation's Cybersecurity.*, Briefing Room, Presidential Actions, The White House, https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

Rose, Scott; Borchert, Oliver; Mitchell, Stu; and Connelly, Sean. Aug. 2020, *Zero Trust Architecture.*, NIST Special Publication 800-207, National Institue of Standards and Technology, U.S. Department of Commerce, https://doi.org/10.6028/NIST.SP.800-207

Young, Shalanda D. 26 Jan. 2022, *Moving the U.S. Government Toward Zero Trust Cybersecurity Principles.*, MEMORANDUM FOR THE HEADS OF EXECUTIVE DEPARTMENTS AND AGENCIES, Version M-22-09, Executive Office of the President; Office of Management and Budget, https://whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf

Ballard Technology. 26 Jan. 2022, *Avionics Databus Tutorials.*, Astronics AES, https://www.astronics.com/avionics-databus-tutorials

maewert. Accessed May 2024, *Interfacing Electronic Circuits to Arduinos.*, Circuits; Arduino, Autodesk Instructables, https://www.instructables.com/Interfacing-Electronic-Circuits-to-Arduinos/

Airlines Electronic Engineering Committee. 17 May 2004, *ARINC Specification 429 Part 1-17: Mark 33 – Digital Information Transfer System (DITS).*, ARINC Document, Aeronautical Radio Inc., https://read.pudn.com/downloads111/ebook/462196/429P1-17_Errata1.pdf, https://web.archive.org/web/20201013031536/https://read.pudn.com/downloads111/ebook/462196/429P1-17_Errata1.pdf

D. De Santo, C.S. Malavenda, S.P. Romano, C. Vecchio. 2021, *Exploiting the MIL-STD-1553 avionic data bus with an active cyber device.*, Computers & Security, Volume 100, 2021, 102097, ISSN 0167-4048, https://doi.org/10.1016/j.cose.2020.102097, https://www.sciencedirect.

com/science/article/pii/S0167404820303709

Gilboa-Markevich, N., Wool, A. 2020, *Hardware Fingerprinting for the ARINC 429 Avionic Bus.*, In: Chen, L., Li, N., Liang, K., Schneider, S. (eds) Computer Security – ESORICS 2020. ESORICS 2020. Lecture Notes in Computer Science(), vol 12309. Springer, Cham. https://doi.org/10.1007/978-3-030-59013-0_3

Kiley, Patrick. 30 Jul. 2019, *Investigating CAN Bus Network Integrity in Avionics Systems.*, Rapid7, https://www.rapid7.com/research/report/investigating-can-bus-network-integrity-in-avionics-systems/

Matthews, Bryan 4 Dec. 2012, *Flight Data for Tail 687.*, DASHlink, National Aeronautics and Space Administration, https://c3.ndc.nasa.gov/dashlink/resources/664/

Airplane DLH754 15 Jul. 2024, *DLH754, Leg 2, 2024-07-02.*, ADS-B Exchange, adsbexchange.com, https://globe.adsbexchange.com/?icao=3c4b35&lat=40.621&lon=52.094&zoom=4.7&showTrace=2024-07-02&leg=2

Appendix

July 24, 2024

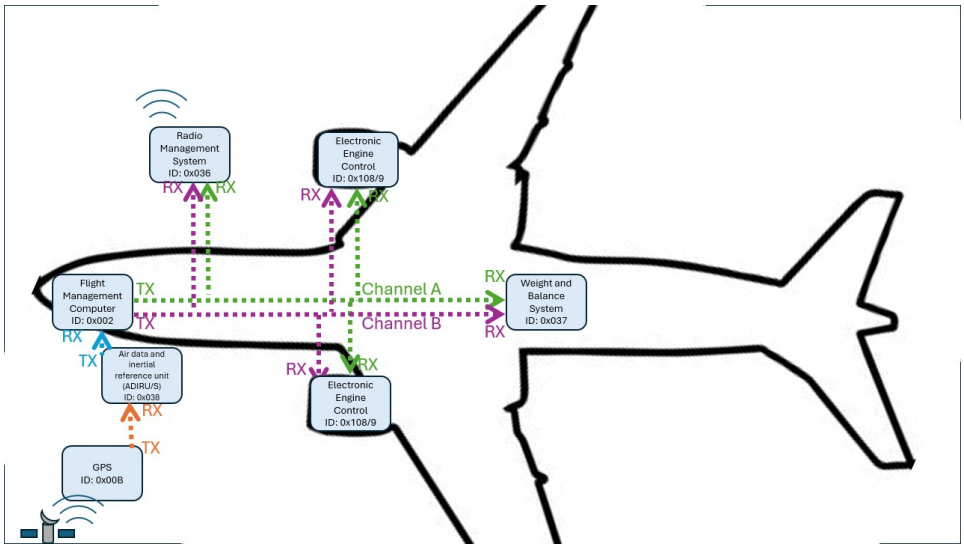


Figure 1: Project system model.

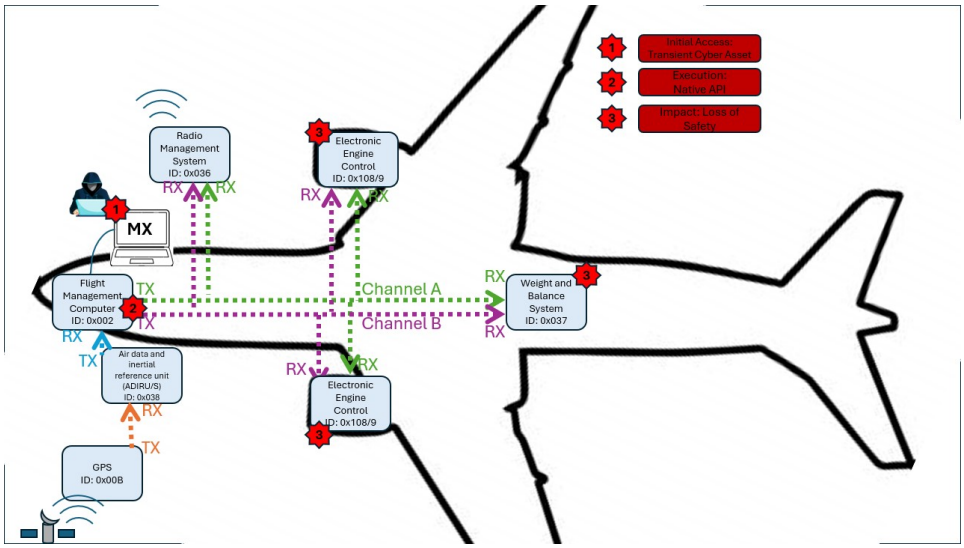


Figure 2: Project threat model.

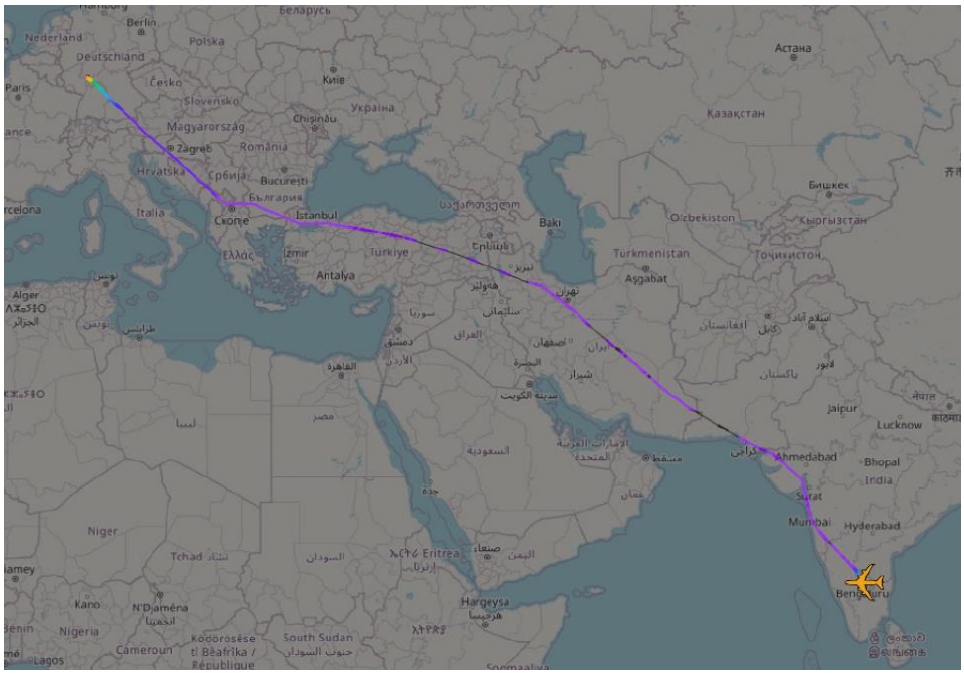


Figure 3: Project threat model.

b

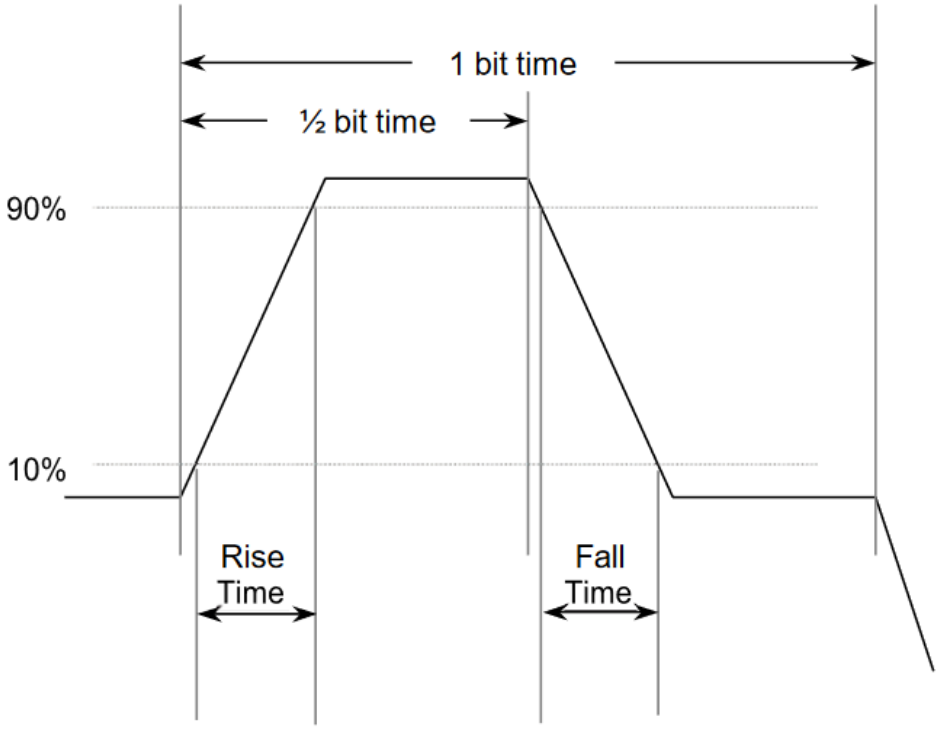


Figure 4: A graph specification for the ‘1’ bit[9].

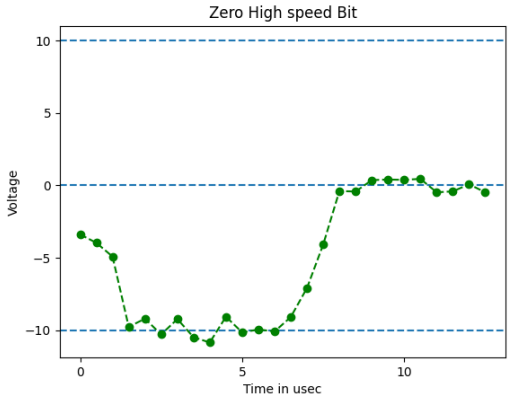


Figure 5: Voltage output samples for a ‘0’ bit at high speed.

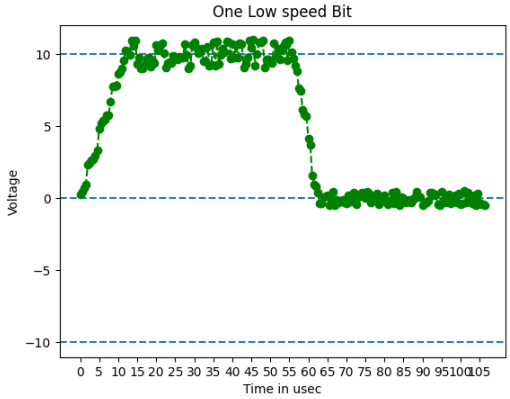


Figure 6: Voltage output samples for a ‘1’ bit at low speed.

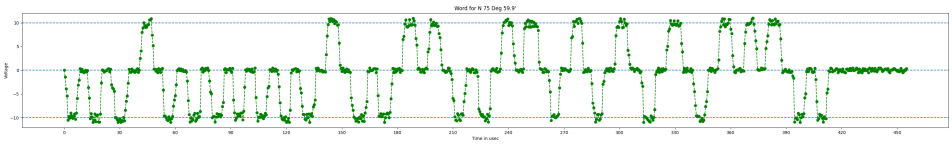


Figure 7: A word representing the latitude N 75 Deg 59.9’, high speed.

d

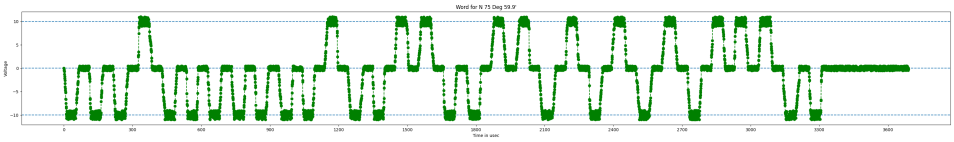


Figure 8: A word representing the latitude N 75 Deg 59.9', low speed.

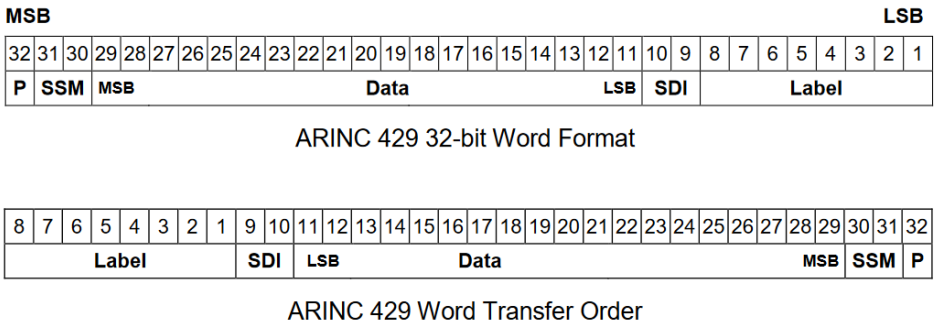


Figure 9: ARINC 429 word format and bit transfer order[9].

```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.

```
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)
```

g

Figure 11: The general BNR Encoding function.

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	← 0 →	SUBSYSTEM SAL MSB	SUBSYSTEM ID (LABEL 172)

Figure 12: General SAL word format.

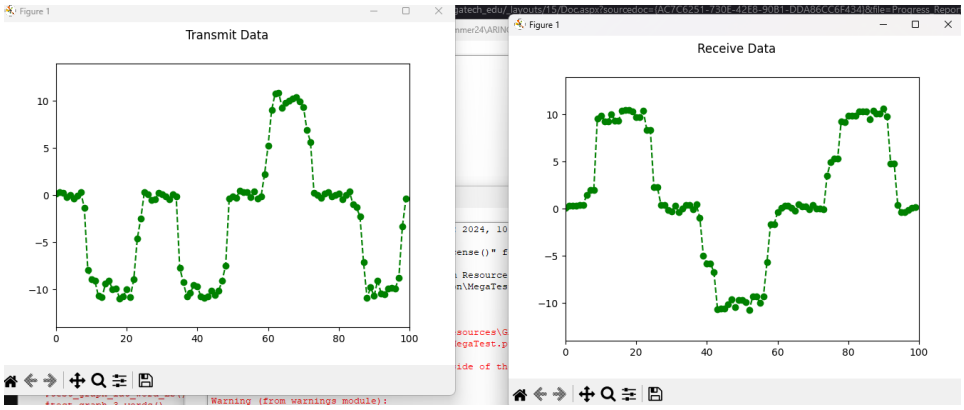


Figure 13: A snapshot of a transmitting (left) and receiving LRU (right) communicating using `BusQueue_Simulator.py` class.

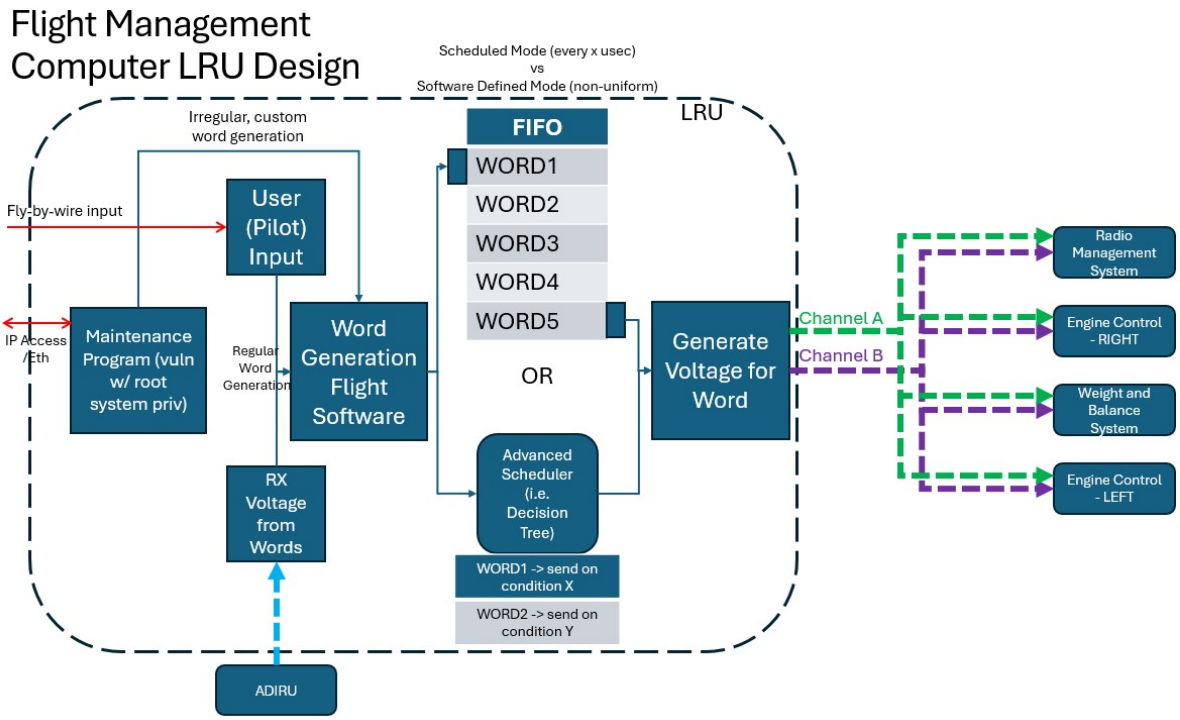


Figure 14: Graphical design blueprint for the FMC

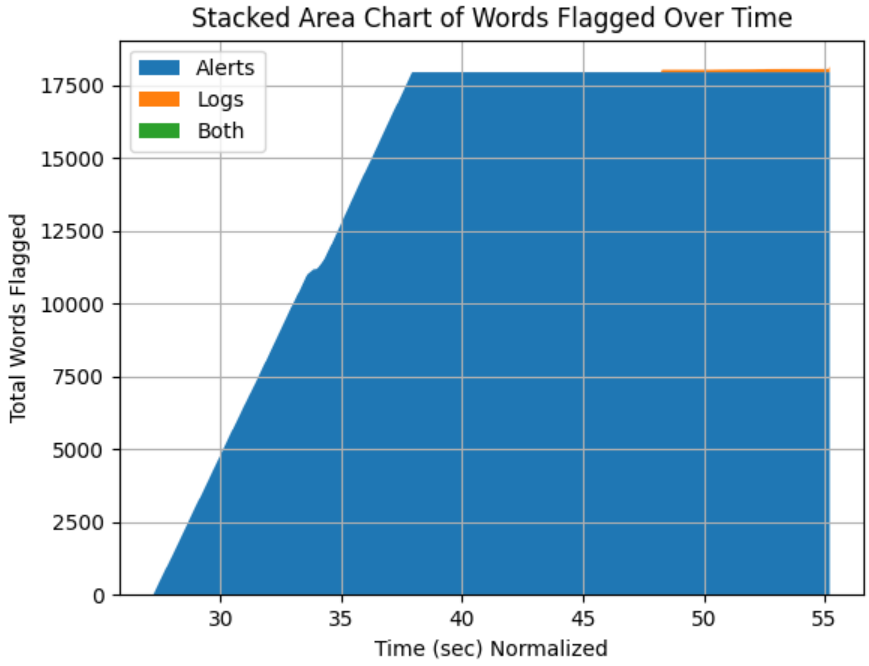
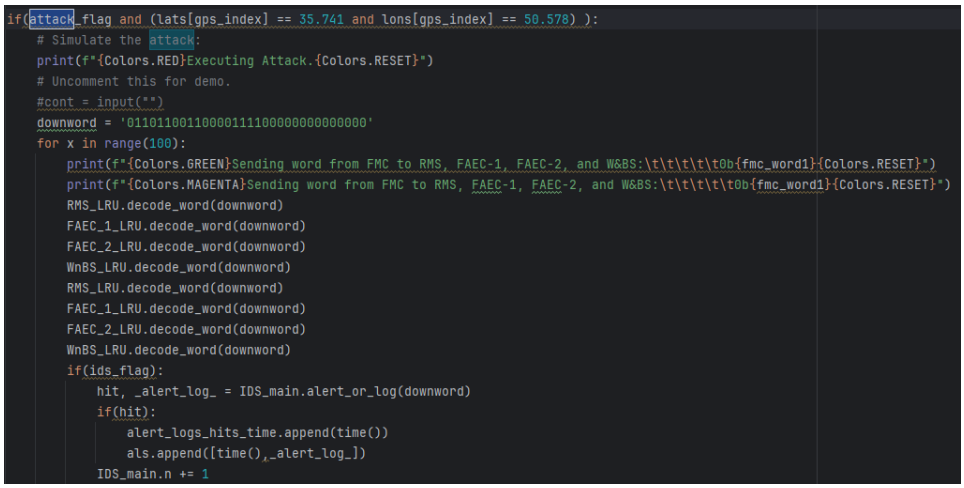


Figure 15: A graph of the alerts (blue) and logs (orange) generated from the IDS showing the attack present, from `main_noThreads.py`.



```
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)=01010101010101010 "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.

m

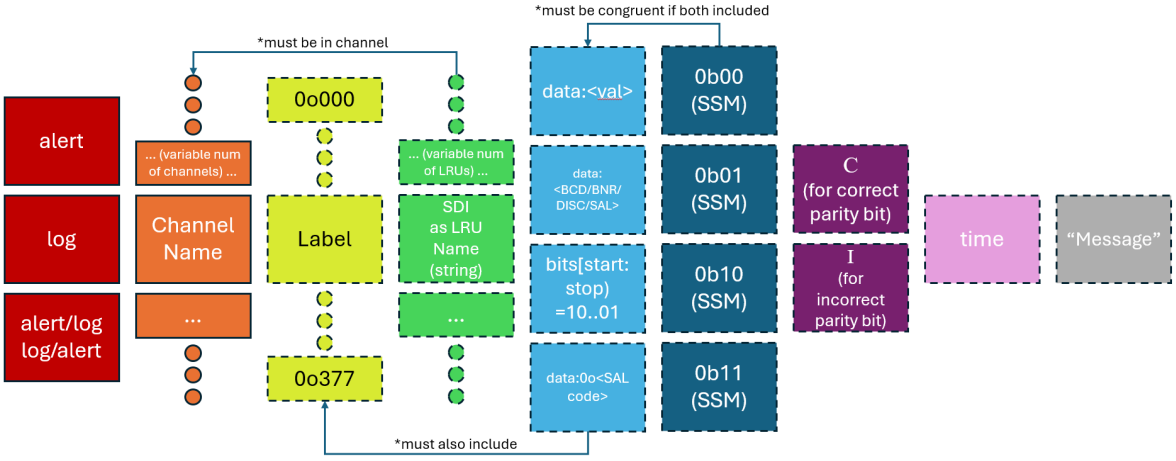


Figure 19: Visual rolodex of rules syntax options.

```
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))
```

o

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','11010000111001100110010000000',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','11111001111100010010001001000',True,False,'Baro Correction (ins. Hg) '),
            ('alert','Channel2','00000010000000101111100000000000',None,False,'Selected Course '),
            ('alert','Channel2','00100010000101101000000000000000',None,False,'Selected Vertical Speed is 1432 Ft/Min upwards'),
            ('log','Channel1','01010110110000000101011011100000',None,False,'Cabin Pressure is 1748 mB'),
            ('log','Channel1','11100101011000100110111111000000',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','10000011000000111100001111100000',None,False,'Application Dependant Data 1 for FMC'),
            ('log','Channel2','11111111000000000000000000000100',None,False,'Identification required for FMC'),
            ('log','Channel2','01100011001100110110110011011000',None,False,'Application Dependant Data 2 for FMC'),
            ('alert','Channel1','00011101111001001010110110011001000',None,False,'ADIRU Discrete Data '),
            ('alert/log','Channel1','00010111111001001010110110011001000',None,False,'ADIRU MX Data '),
            ('alert','Channel2','00001111000000011110000000000000',None,False,'Aircraft Condition and Event Surveillance System (ACESS)'),
            ('alert','Channel2','10111110000101111100000000000000',None,False,'HGA/IGA HPA'),
            ('alert','Channel2','01011111000010111110000000000000',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`.

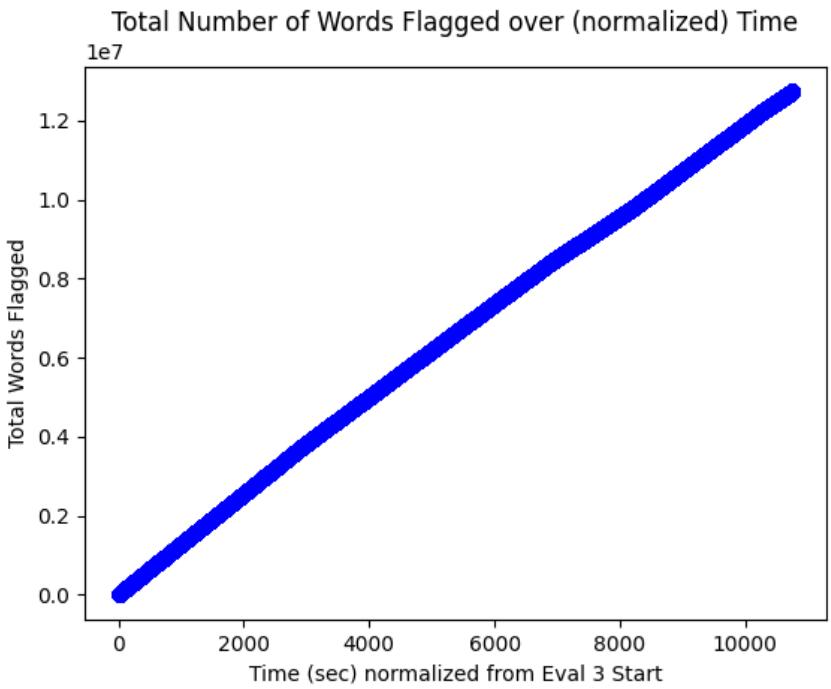


Figure 27: Number of alerted/logged words until that point over time for `IDS_Eval2.py`.

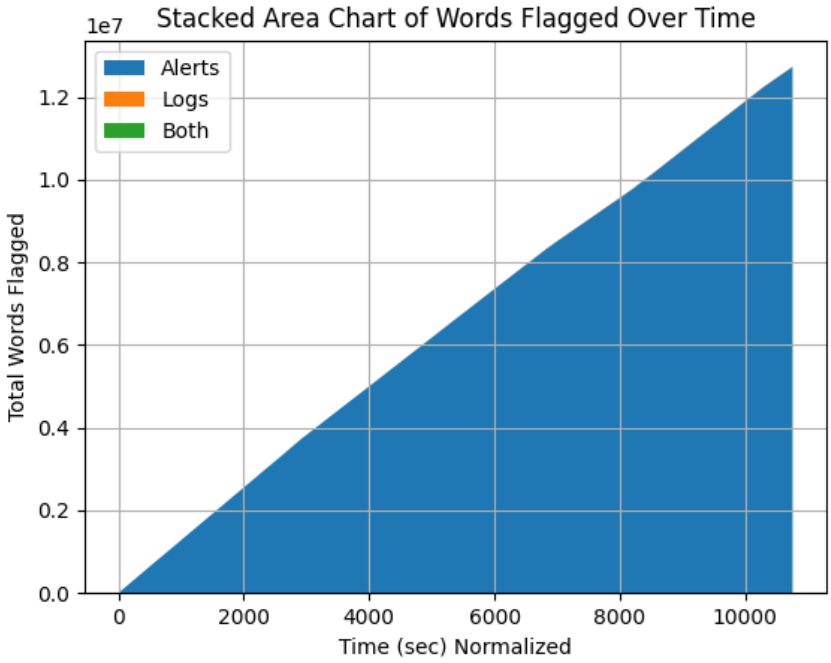


Figure 28: Cumulative graph of number of alerted/logged words over time for `IDS_Eval2.py`. Blue is alerts and Orange is Logs.

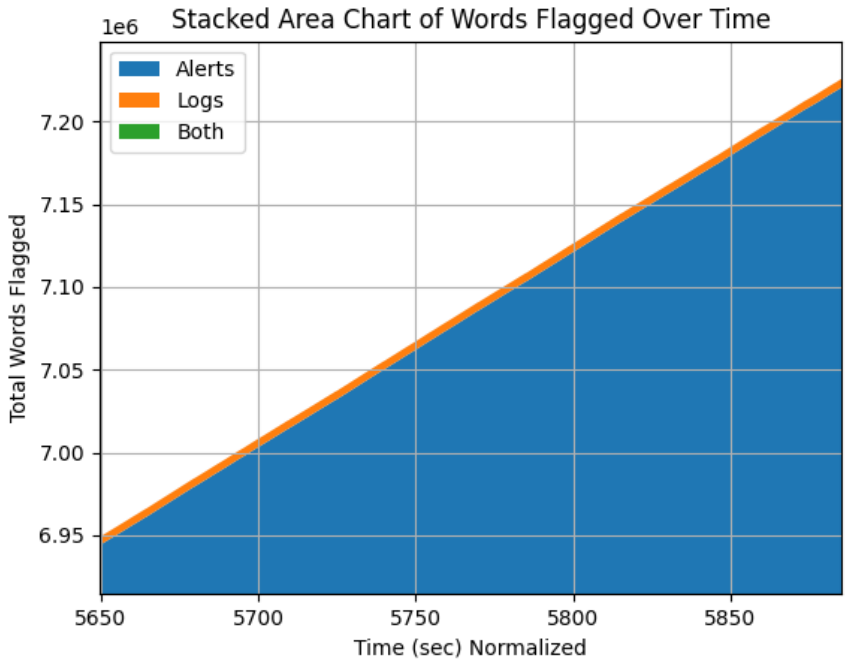


Figure 29: Zoomed in view to see figure 27 delineation more clearly.

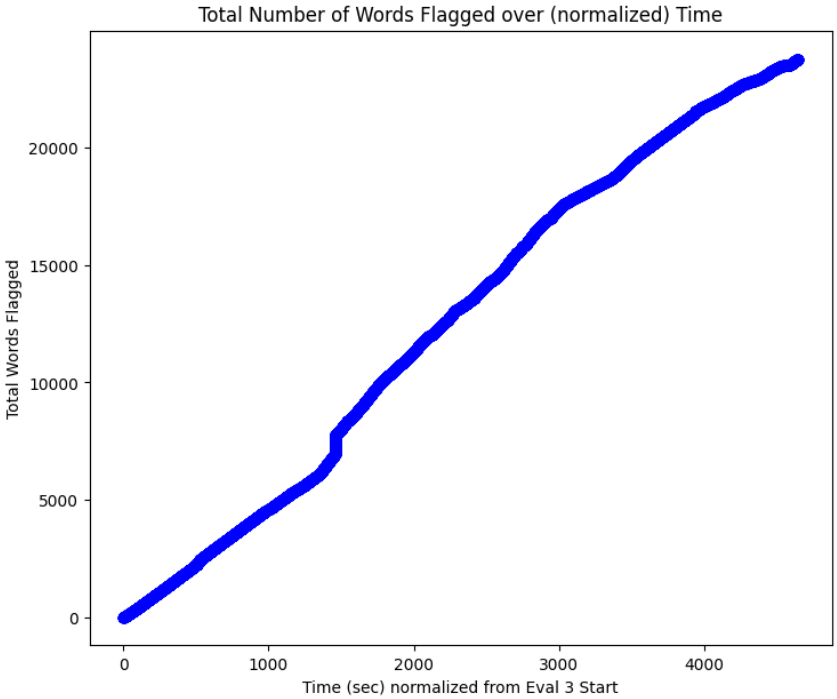


Figure 30: Number of alerted/logged words until that point over time for `IDS_Eval3.py`

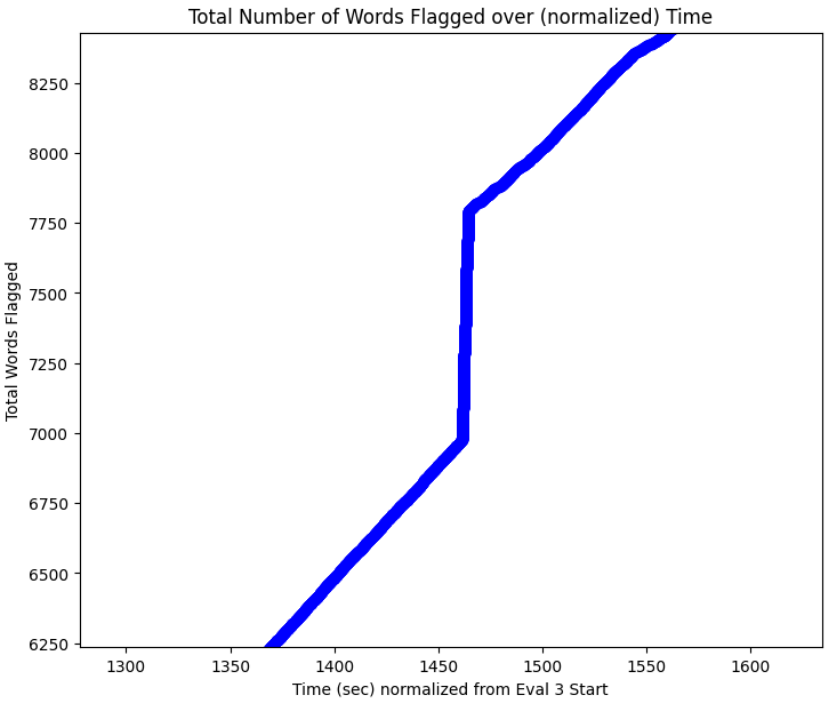


Figure 31: Zoomed in graph for `IDS_Eval3.py` showing the huge jump in alerts because of the attack words.

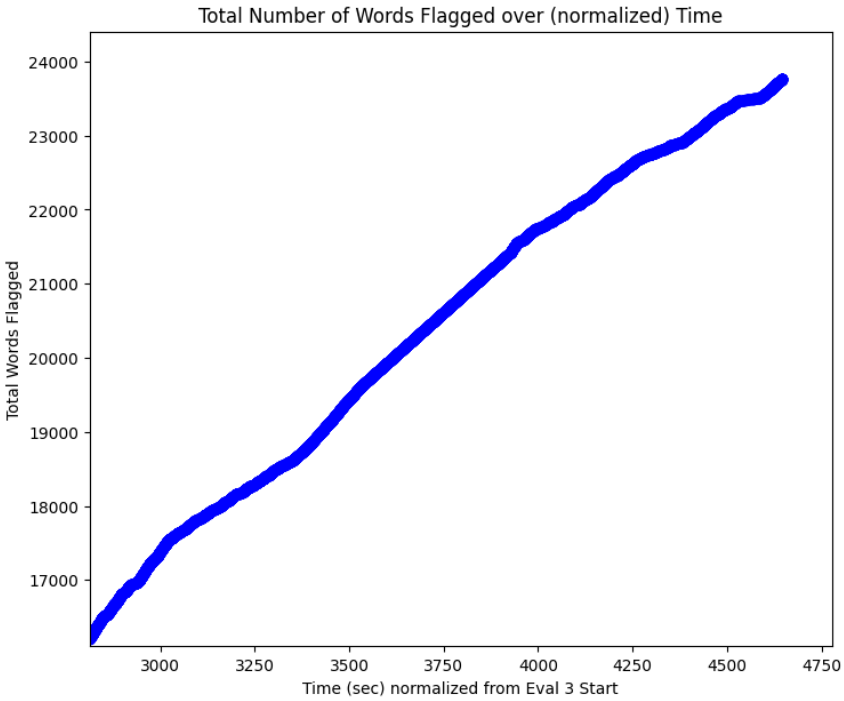


Figure 32: Zoomed in graph for `IDS_Eval3.py` showing the end of the flight as words are alerting sporadically on the plane's descent.

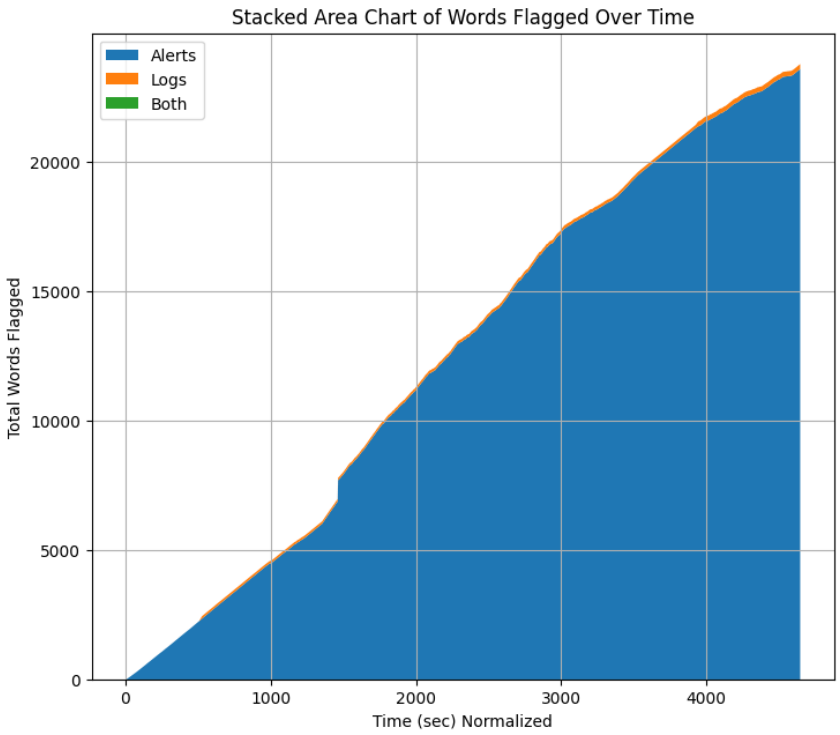


Figure 33: Cumulative graph of number of alerted/logged words over time for `IDS_Eval3.py`. Blue is alerts and Orange is Logs.

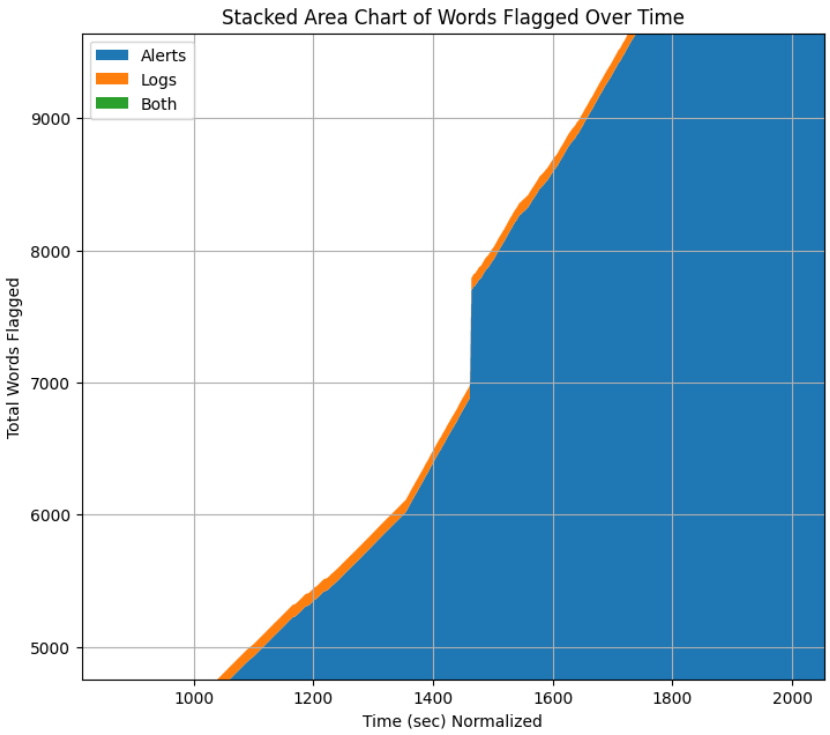


Figure 34: Zoomed in cumulative graph for `IDS_Eval3.py`. Note that the jump is purely blue which means it's only alerts–for the attack. No logs are generated.

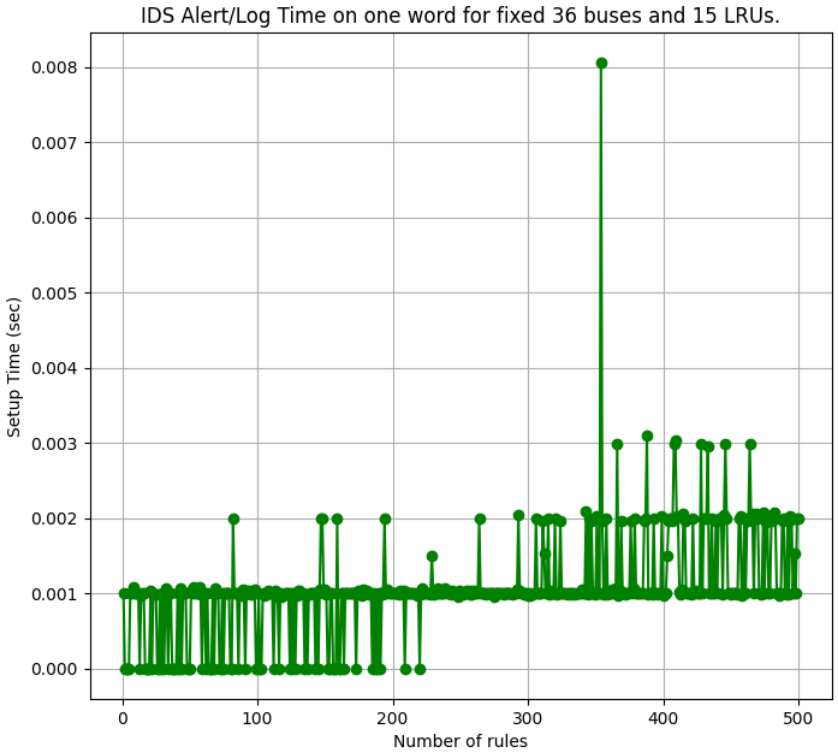


Figure 35: `IDS_Eval4.py`: Time to process 1 word for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

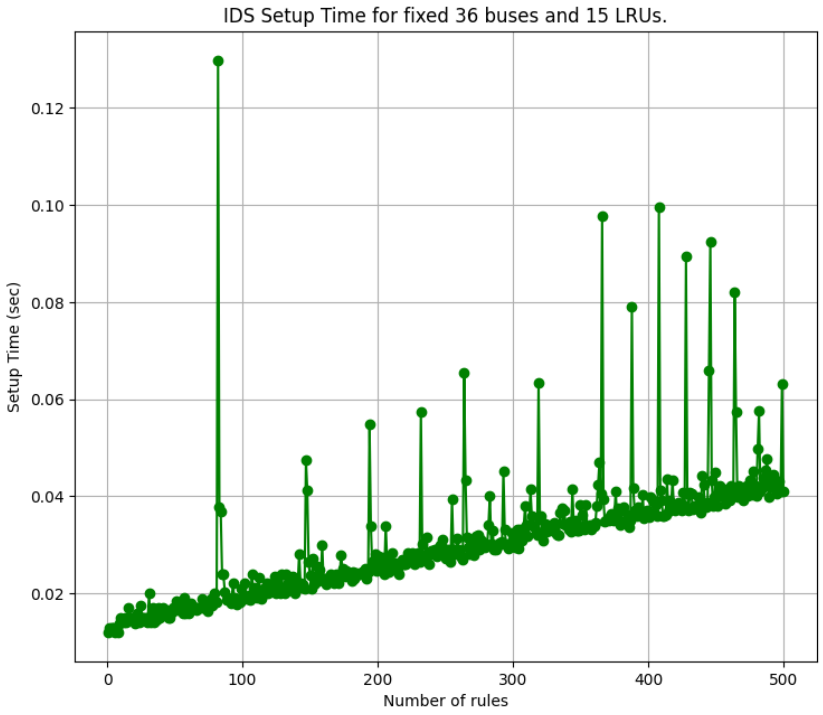


Figure 36: `IDS_Eval4.py`: Time to boot-up the IDS for 1 to 500 rules while fixing the number of channels to 36 and LRUs to 15.

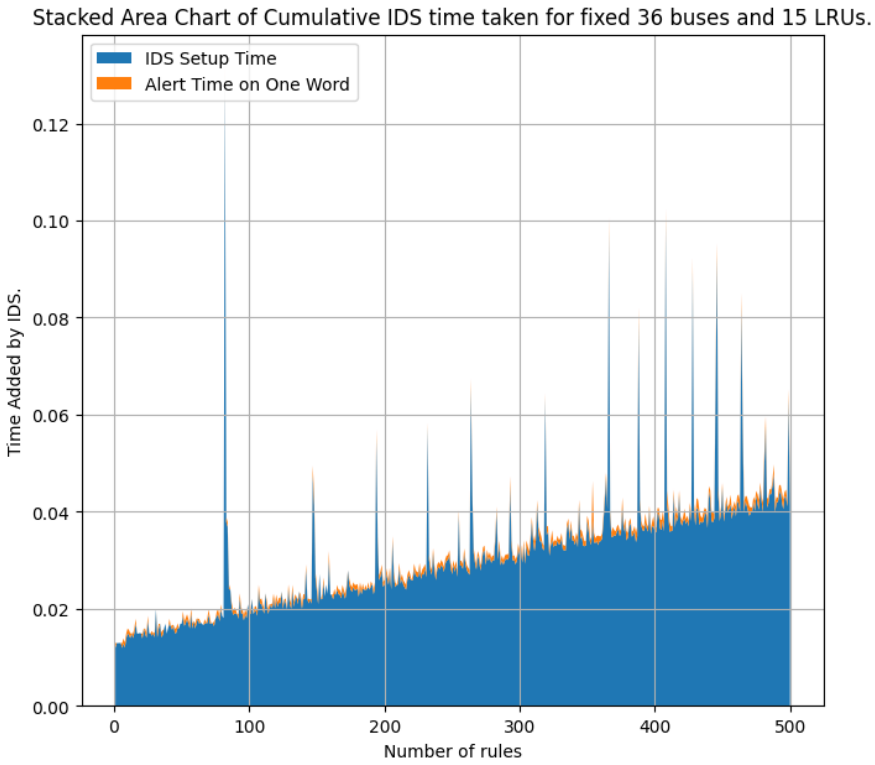


Figure 37: `IDS_Eval4.py`: Cumulative time added for figures 35 and 36.

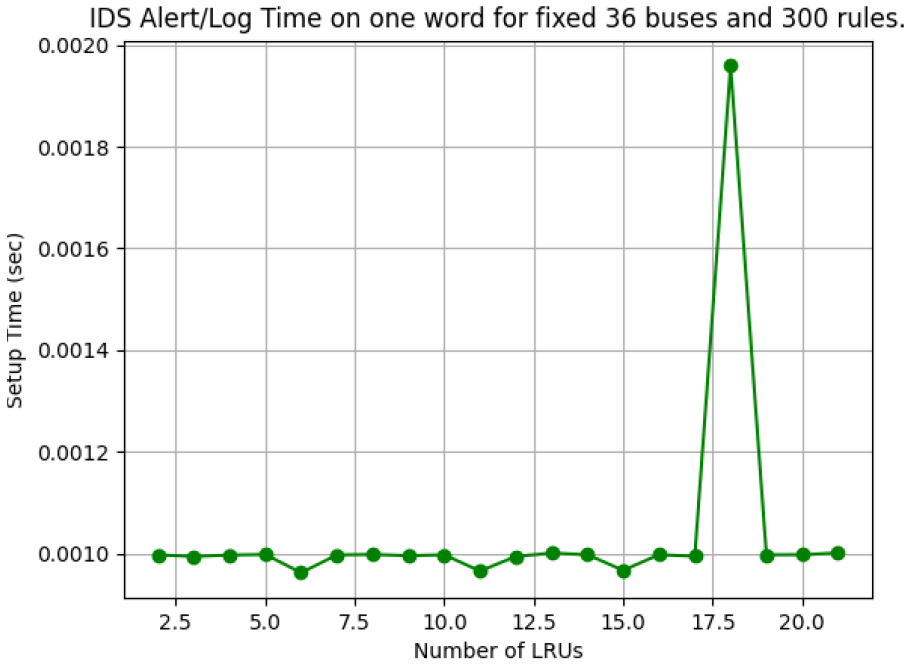


Figure 38: `IDS_Eval4.py`: Time to process 1 word for 2 to 21 LRUs while fixing the number of channels to 36 and rules to 300.

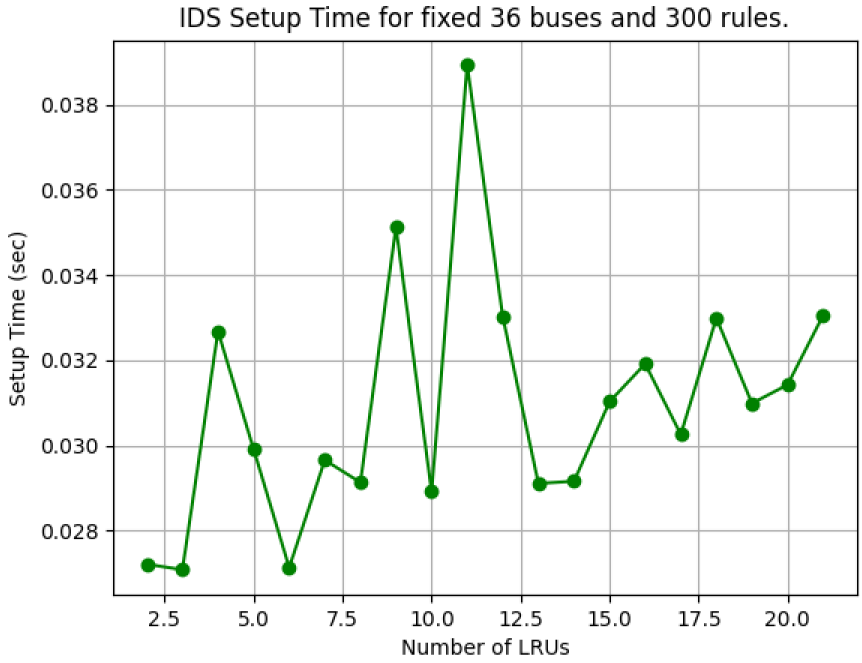


Figure 39: `IDS_Eval4.py`: Time to boot-up the IDS for 2 to 21 LRUs while fixing the number of channels to 36 and rules to 300.

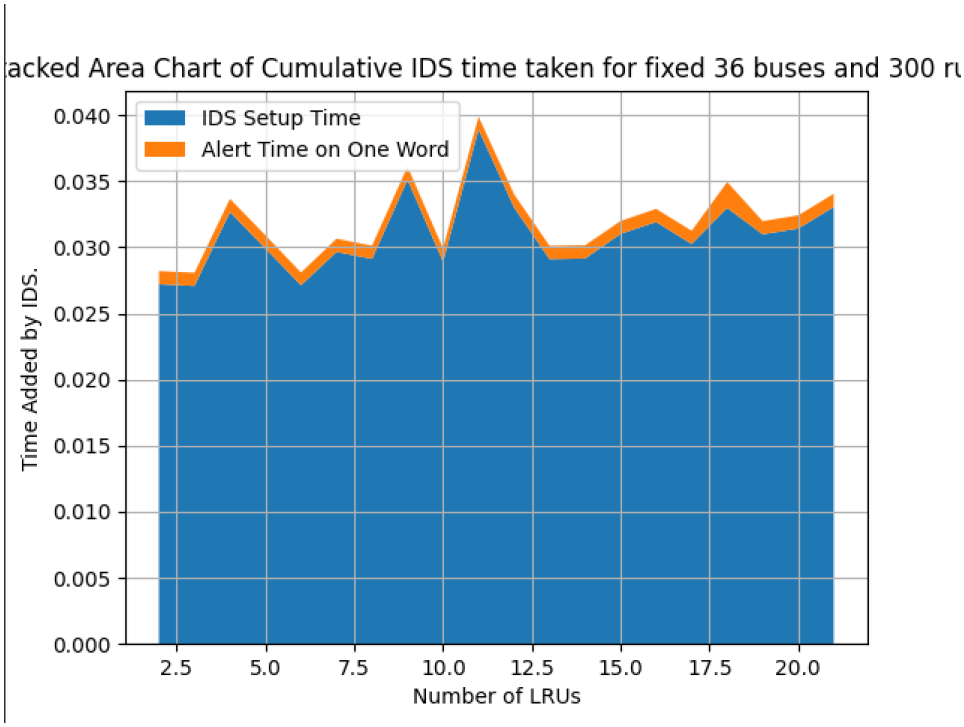


Figure 40: `IDS_Eval4.py`: Cumulative time added for figures 38 and 39.

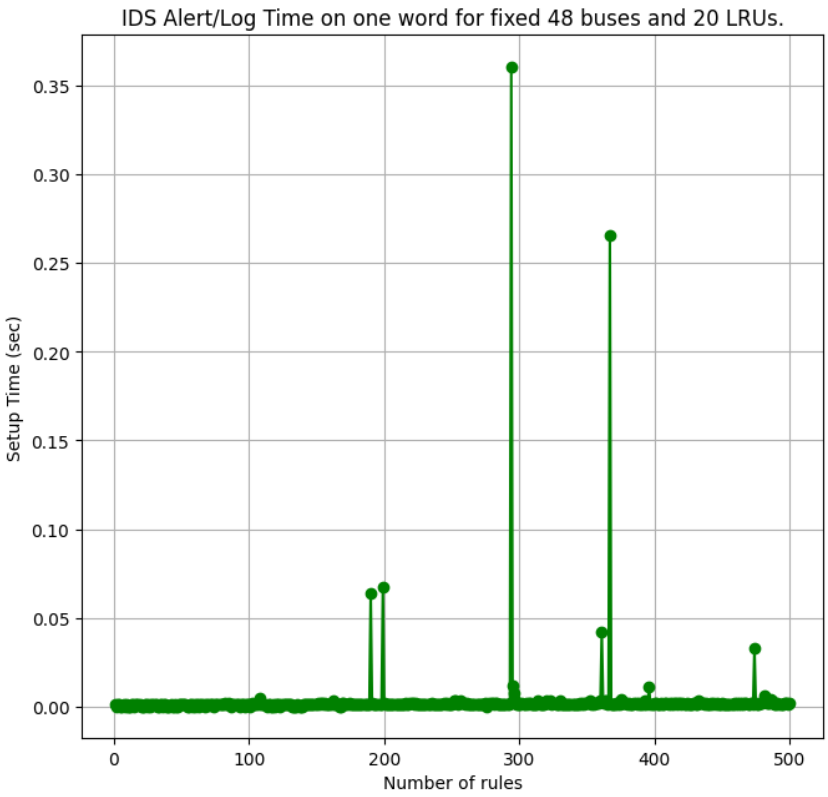


Figure 41: `IDS_Eval4.py`: Time to process 1 word for 1 to 500 rules while fixing the number of channels to 48 and LRUs to 20.

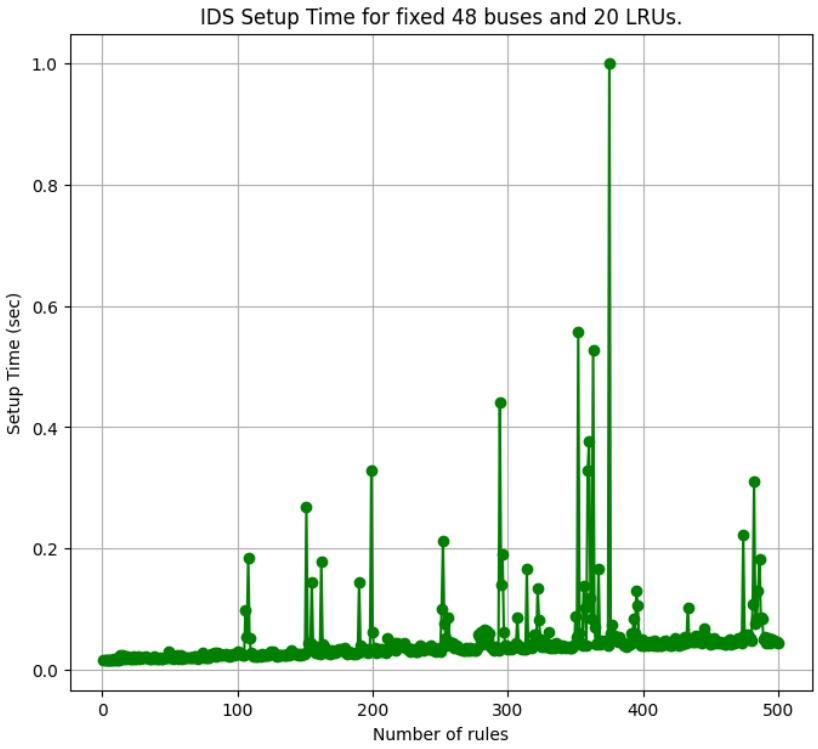


Figure 42: `IDS_Eval4.py`: Time to boot-up the IDS for 1 to 500 rules while fixing the number of channels to 48 and LRUs to 20.

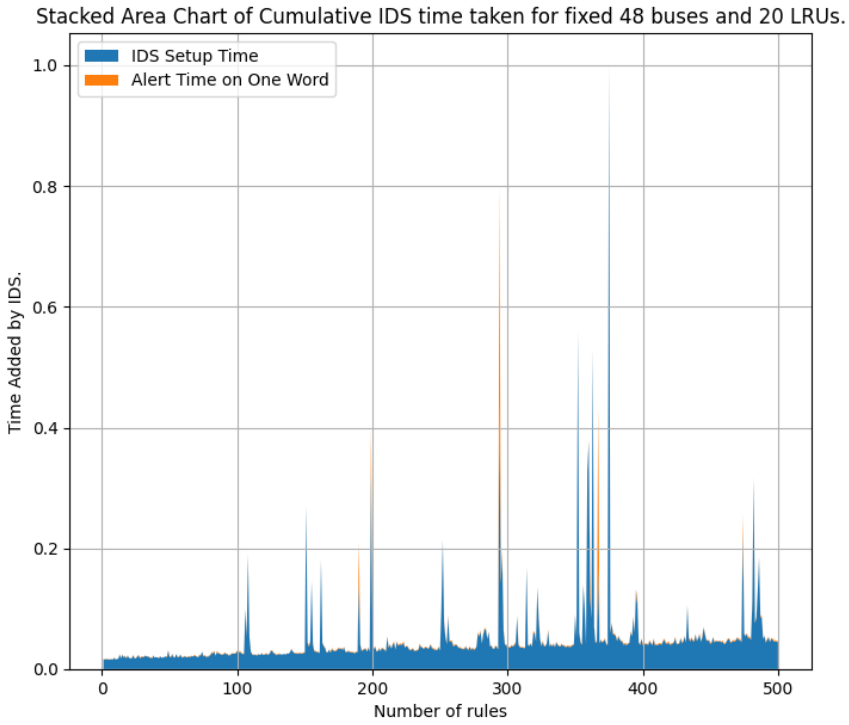


Figure 43: `IDS_Eval4.py`: Cumulative time added for figures 41 and 42.

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