الجامعه المصريه للتعلم الالكترونى الأهليه

The Egyptian E-Learning University

Deep Learning System For Discovering Self Driving Intrusions

By:-

1. **Antonius Fekry Abdo** **20-01015**
2. **Yassa Adli Fawzy** **20-00562**
3. **Viola Mamdouh Mounir** **20-00063**
4. **Youstena Tharwat Gergis** **20-00618**
5. **Engy Emad Gamil** **20-00621**
6. **Alaa Nasser Badr Mahdi** **20-00989**
7. **Mohamed Mokhtar Saad El-Deen** **20-00885**

Under Supervised of

Dr. Mohamed Zedaan

Lecturer, Department of Information Technology
Egyptian E-Learning University (EELU)

Eng. Dina Mohamed

Teaching Assistant, Department of Information Technology at Egyptian E-Learning University (EELU)

This thesis is submitted as a partial fulfillment of the requirements for thedegree of Bachelor of Science in Computer & Information Technology.

Assiut 2023-2024

Acknowledgment and Dedication

First and foremost, we express our sincere gratitude and praise to the Almighty God for His blessings throughout our years in college and the successful completion of our graduation project, marking a significant stage in our lives. We have exerted efforts in this project; however, it would not have been possible without the kind support and assistance of numerous individuals and organizations. We extend our heartfelt thanks to all of them. We are grateful to the Egyptian E-Learning University, especially the Assuit Center, for helping us attain this level of awareness.

We would like to convey our deep and sincere gratitude to our project supervisor, Professor Dr. Mohamed Zedaan, and Engineer Dina Mohamed, for granting us the opportunity to work under their guidance and providing invaluable direction throughout this project. It was a great privilege and honor to study and work under their tutelage. We are extremely grateful for their contributions. We are especially indebted to express our profound thanks to Engineer Dina Mohamed for her guidance, constant supervision, patience, friendship, empathy, and great sense of humor. Her dynamism, vision, sincerity, and motivation have deeply inspired us.

Finally, we would like to express our heartfelt gratitude to everyone who assisted us during the completion of our graduation project

Table of Contents

Table of Figure

List of acronyms or abbreviation.

CAVs: Connected and Autonomous Vehicles

ECUs: Electronic Control Units

IVNs: In-Vehicle Networks

CAN: Controller Area Network

KNN: k-Nearest Neighbor

LSTM: Long Short-Term Memory

R^2: Determination Coefficient

ADAS: Advanced Driver-Assistance Systems

SAE: Society of Automotive Engineers

LIDAR: Light Detection and Ranging

GPS: Global Positioning System

ICT: Information and Communication Technology

AI: Artificial Intelligence

U.S. Bureau of Labor Statistics: United States Bureau of Labor Statistics

CA: California (State abbreviation)

FL: Florida (State abbreviation)

CNNs: Convolutional Neural Networks

RNNs: Recurrent Neural Networks

YOaaLO: You Only Look Once

DRL: Deep Reinforcement Learning

IDS: Intrusion Detection Systems

CRC: Cyclic Redundancy Check

AMP: Arbitration on Message Priority

EMI: Electromagnetic Interference

DoS: Denial-of-Service

PCB: Printed Circuit Board

EMC: Electromagnetic Compatibility

V2X: Vehicle-to-Everything

OTA: Over-The-Air

ABS: Anti-lock Braking System

OSI: Open Systems Interconnection (model)

LIN: Local Interconnect Network

DNN: Deep Neural Network

OBD: On-Board Diagnostic

RTR: Remote Transmission Request

CRC: Cyclic Redundancy Check

ACK: Acknowledgement

MSB: Most Significant Bit

DLC: Data Length Code

SOF: Start of Frame

EOF: End of Frame

SJW - Synchronization Jump Width

TCU - Transmission Control Unit

ABS - Anti-lock Braking System

FD - Flexible Data-rate

ISO - International Organization for Standardization

V: Voltage

Ω: Ohms (Unit of impedance)

bps: Bits per second

ESD: Electrostatic Discharge

DB9: D-Sub 9-pin connector

M12: M12 circular connector

CISPR: International Special Committee on Radio Interference

IEC: International Electrotechnical Commission

SAE: Society of Automotive Engineers

CiA: CAN in Automation (organization)

NMEA: National Marine Electronics Association

OBD-II: On-Board Diagnostics II

DTC: Diagnostic Trouble Codes

TEC: Transmit Error Counter

REC: Receive Error Counter

ID: Identifier

RPM: Revolutions Per Minute

GRU: Gated Recurrent Unit

BRNN: Bidirectional Recurrent Neural Network

Abstract:

Connected and autonomous vehicles (CAVs) present exciting opportunities for improving both the mobility of people and the efficiency of transportation systems. The small computers in autonomous vehicles (CAVs) are referred to as electronic control units (ECUs) and are often perceived as being components of a broader cyber-physical system. Subsystems of ECUs are often networked together via various in-vehicle networks (IVNs), such as the Controller Area Network (CAN), so that data can be exchanged, and the vehicle can operate more efficiently. The purpose of this work is to explore the use of machine learning and deep learning methods in defense against cyber threats to autonomous cars, particularly to protect the CAN bus from malicious attacks. Our primary emphasis is on identifying erroneous or malicious information implanted in the data buses of various automobiles.

To categorize this type of erroneous data, the gradient boosting method, which is a powerful machine learning technique that combines multiple weak models to create a strong predictive model, is used, providing a productive illustration of machine learning application. To examine the performance of the proposed model, two real datasets, namely the Car-Hacking and UNSE-NB15 datasets, were used. These real automated vehicle network datasets were used in the verification process of the proposed security solution. These datasets included simulated spoofing, flooding, and replay attacks, as well as benign packets representing normal CAN bus traffic.

The categorical data were transformed into numerical form via pre-processing techniques. Machine learning algorithms, namely k-nearest neighbor (KNN) and decision trees, as well as deep learning algorithms such as long short-term memory (LSTM) and deep autoencoders, were employed to detect CAN bus attacks. According to the findings of the experiments, using the decision tree and KNN algorithms as machine learning approaches resulted in accuracy levels of 98.80% and 99%, respectively, in detecting malicious CAN bus traffic. On the other hand, the use of LSTM and deep autoencoder algorithms as deep learning approaches resulted in accuracy levels of 96% and 99.98%, respectively, in the same task.

The maximum accuracy was achieved when using the decision tree and deep autoencoder algorithms. Statistical analysis methods were used to analyze the results of the classification algorithms, and the determination coefficient measurement (R^2), which represents the proportion of variance in the dependent variable that is predictable

from the independent variable(s), for the deep autoencoder was found to reach a value of R^2 = 95%. This indicates that the deep autoencoder model can explain 95% of the variance in the data, which is a very high level of predictive power.

The performance of all the models built in this way surpassed that of existing methods, with almost perfect levels of accuracy being achieved. The system developed can effectively overcome security issues in IVNs by detecting and mitigating various types of CAN bus attacks, thereby enhancing the security and reliability of autonomous vehicles.

Chapter 1

Introduction

1.1. Introduction

The concept of self-driving cars has been discussed since the 1920s; however, the first semi-automated car was developed by Tsukuba Mechanical Engineering Laboratory in Japan in 1977. A major landmark in self-driving car development was achieved in the 1980s with the Carnegie Mellon University's Navlab and ALV projects, which created autonomous vehicles capable of reaching speeds up to 20 mph (32 km/h) on urban roads.

After further developments in the field, in 2013, Tesla Motors began their autonomous car project. In 2014, they released the Tesla Model S with a semi-autonomous "Autopilot" mode, which provided advanced driver-assistance features such as lane centering, adaptive cruise control, and self-parking. Further improvements in the Autopilot software were released in subsequent models like the Tesla Model X and Model 3.

A survey of public opinion about self-driving cars, conducted by Brandon Schoettle and Michael Sivak in the top three major English-speaking countries (United States, United Kingdom, and Australia), revealed an initially positive response from the public towards this upcoming technology in the automobile industry.[1]

However, in many other countries, including ours, there was no significant development in the field of autonomous vehicles. Therefore, this project aimed to analyze and build upon the existing technology to develop an autonomous car at a low cost that could be affordable for many citizens. In addition to the existing systems, we have included a Traffic Signal Response feature, which is not present in Tesla or many other companies' offerings. This feature can improve transportation quality and safety by reducing human errors and increasing road safety.

Furthermore, the development of autonomous vehicles can bring about a revolution in aiding differently-abled and blind individuals, enabling them to travel independently without relying on others. By incorporating advanced technologies and addressing the unique needs of different user groups, this project aims to make autonomous vehicles more accessible and inclusive.

[1] Schoettle, B., & Sivak, M. (2014). A survey of public opinion about autonomous and self-driving vehicles in the US, the UK, and Australia. University of Michigan Transportation Research Institute.

1.2. Definition of Self-Driving Cars

A self-driving car (also known as an autonomous car, driverless car, or robotic car) is a vehicle that is capable of sensing its environment and navigating without human input by using a combination of sensors, cameras, radar, and artificial intelligence (AI) systems. To be considered a fully autonomous vehicle, it must be able to operate without any human intervention or supervision, and navigate safely to a predetermined destination over roads that have not been specifically adapted or designed for its use.

Several major automotive manufacturers and technology companies are actively developing and testing autonomous car technologies, including Audi, BMW, Ford, General Motors, Tesla, Volkswagen, Volvo, Toyota, Waymo (formerly Google's self-driving car project), Uber, Baidu, and others. These companies are investing heavily in research and development efforts to overcome the challenges associated with autonomous driving, such as perception, decision-making, and control systems.

Google's self-driving car project, now known as Waymo, has been one of the pioneers in this field. Their testing involved a fleet of self-driving cars, including modified Toyota Prius and Audi TT vehicles, which have collectively navigated over 20 million miles (as of 2020) on public roads across various cities and environments, primarily in the United States.

It's important to note that while significant progress has been made, fully autonomous vehicles that can operate safely under all conditions without any human supervision are not yet available for commercial use. Most current systems are considered advanced driver-assistance systems (ADAS) or partially automated systems that still require a human driver to monitor the vehicle and take control when necessary.

Regulatory frameworks and infrastructure changes may also be required to accommodate the widespread adoption of fully autonomous vehicles, ensuring their safe and efficient integration into existing transportation systems.

1.3. How Autonomous Vehicles Operate in Practice

The development of autonomous vehicles has reached advanced stages. The artificial intelligence systems employed in these cars have made it possible to use computers and modern equipment to distinguish between different types of road obstacles and their varying conditions, and to interact with them according to predetermined criteria.

Computers in self-driving cars exchange information with each other through special communication systems, enabling them to learn from unexpected situations encountered by other vehicles. This shared learning process helps autonomous cars adapt and improve their ability to deal with similar circumstances in the future. Therefore, self-driving cars require central computer systems that are more powerful than the electronic control units currently used, and they also need specialized diagnostic systems.

Most autonomous driving systems create an autonomous map of their surroundings based on information obtained from a wide range of sensors, such as radar, lidar (light detection and ranging), and high-resolution cameras. Some autonomous vehicles use a combination of lasers and other sensors to create their map, while others rely on radar, high-power cameras, acoustic detection, and pre-loaded high-definition maps.

Special software processes the information obtained in real-time, tracks the path, and sends instructions to the car's actuators, which control acceleration, braking, and steering. These software systems incorporate special rules and algorithms to follow traffic rules, navigate obstacles, avoid collisions, make predictions, and differentiate between objects (such as distinguishing between a bicycle and a motorcycle).

The level of autonomy in vehicles can be categorized based on the Society of Automotive Engineers (SAE) levels of driving automation:

- Levels 0, 1, and 2 (partially automated): These vehicles require driver intervention and monitoring at all times.

- Level 3 (conditionally automated): The vehicle can handle some driving situations without human intervention, but the driver must be ready to take control when requested.

- Level 4 (highly automated): The vehicle can operate without human intervention in certain conditions and environments, but may require a human driver for some circumstances.

- Level 5 (fully automated): The vehicle can operate without human intervention in all conditions and environments, without any need for a driver or controls.

Autonomous vehicles can be further distinguished by whether they are connected or unconnected, referring to their ability to communicate with other vehicles and with city infrastructure, such as next-generation traffic lights and urban traffic management systems. Connected autonomous vehicles can potentially improve traffic flow, safety, and efficiency by sharing real-time data and coordinating their movements.

It's important to note that while significant progress has been made, fully autonomous vehicles (Level 5) that can operate safely under all conditions without any human supervision are not yet available for widespread commercial use. Ongoing research and development efforts, as well as regulatory frameworks and infrastructure changes, are required to enable the safe and efficient integration of autonomous vehicles into existing transportation systems.

1.4. Six Levels of Autonomous Control in Autonomous Vehicles

Self-driving cars are divided into different levels according to a sliding scale from 0 to 5. Understanding these levels is necessary before talking about the practical operation of this type of car.

The more technology a car has - and therefore the number of sensors used in it - the greater the degree of automation. So the Society of Automotive Engineers (SAE) created a classification to differentiate vehicles according to their degree of automation, to make it easier for consumers and maintenance professionals to identify a vehicle model. The following six levels have been identified:

Level 0 (No Automation): The driver is in full control of the vehicle at all times.

Level 1 (Driver Assistance): The vehicle has one automated system, such as electronic stability control or adaptive cruise control, to assist the driver.

Level 2 (Partial Automation): The vehicle can control both steering and acceleration/deceleration under certain conditions, but the driver must remain engaged and ready to take over at all times.

Level 3 (Conditional Automation): The vehicle can handle all aspects of driving, but the driver must be prepared to intervene when the system requests. This is sometimes referred to as "eyes off, but not mind off" driving.

Level 4 (High Automation): The vehicle can perform all driving tasks within a defined operational design domain (ODD), such as a specific geographic area or under certain weather conditions. The driver may have the option to control the vehicle, but is not required to do so.

Level 5 (Full Automation): The vehicle can perform all driving tasks in all conditions that a human driver could handle, without any human intervention required.

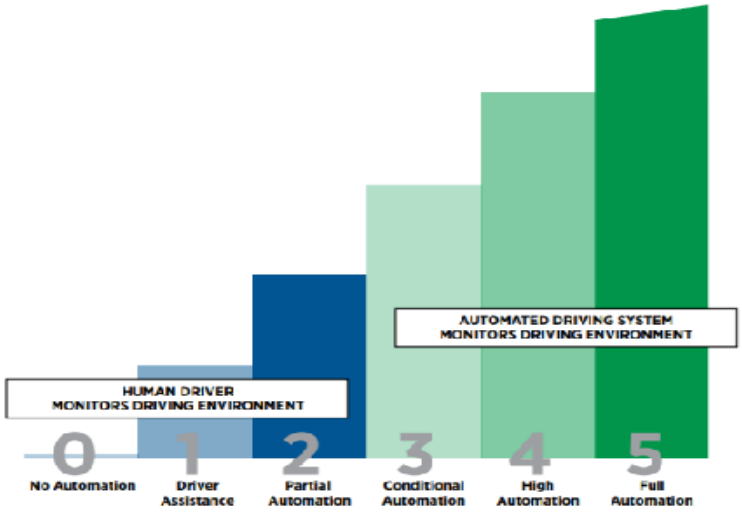


Figure 1: Six Levels of Autonomous

While some models are being produced in practice with some level of automation, automakers, and technology companies are developing other, more advanced prototypes.

Therefore, it is natural for mechanics and other maintenance specialists to prepare for the near future in which autonomous vehicles of varying degrees of autonomy become common. There is no doubt that it will contribute to making our daily lives more comfortable and safer.

Based on automakers' estimates and technology estimates, Level 4 autonomous vehicles will practically go on sale within three years.

1.5. Self-Driving Car Technologies

The idea of self-driving vehicles is to reach the point where no driver is possible, so the technologies for these vehicles must be developed to reach a level of reliable safety. To make this scenario possible, several innovations have been developed, improved, and integrated into self-driving cars; Below we review some of them:

Equipment

The equipment in self-driving cars is responsible for detecting environmental characteristics and transmitting data to the vehicle's computer. The most commonly used devices include cameras, radars, acoustic detection systems, and LIDAR.

Cameras: Cameras provide visual data that the car’s software utilizes to identify objects, read traffic signs, and detect lane markings.

Stereoscopic Camera: This device contains two or more lenses to capture images from different angles, enabling the creation of a sense of depth that mimics human vision to produce three-dimensional images.

Infrared Camera: Infrared cameras facilitate precise viewing in areas with low or no illumination. They identify objects by capturing infrared rays based on temperature differences.

Radar: Radar systems emit radio waves in a specific direction. The echoes of these waves, which bounce off obstacles, are measured to determine the size and distance of the obstacles based on the speed and intensity of the reflected beams.

Audio Detection: Acoustic detection operates similarly to radar but uses inaudible sound waves instead of radio waves.

LIDAR: LIDAR systems utilize laser pulses to create thousands of bright dots, scanning the surrounding environment. LIDAR provides a faster signal and covers a wider 360-degree area with greater accuracy.

Electronic Stability Control: This technology makes driving corrections based on the speed of each wheel and the car's inclination and yaw, thereby enhancing stability and safety.

Vacuum Electromechanical Servo Brake: The vacuum electromechanical servo brake can generate brake pressure in less than 120 milliseconds, which is three times faster than conventional braking systems, thereby increasing safety during emergency braking.

GPS, Speedometer, and Odometer: For autonomous navigation in urban environments, self-driving cars require updated maps and precise location control. GPS equipment, combined with a speedometer and odometer, enables the car's computer to determine its location even in the absence of a satellite signal.

Software

Software functions as the brain of a self-driving car, processing information from the car's sensors to determine appropriate actions such as driving, stopping, or slowing down. There are three primary software systems for self-driving cars: perception, planning, and control.

Perception: The perception system allows self-driving cars to understand their surroundings through sensors. It uses images to identify objects, such as other vehicles, pedestrians, or obstacles. This system is analogous to human vision, where sensors (photoreceptors) in the eyes detect light waves and convert them into electrochemical signals. These signals are transmitted via neural networks to the brain's visual cortex, enabling object recognition.

Planning: The planning system enables the car to make decisions aimed at achieving logical and safe goals. It integrates information from sensors with predefined policies and experiences about navigating the environment. This system ensures the car can respond appropriately in various scenarios, such as stopping at a red light or overtaking another vehicle.

Control: The control system translates the planning system's decisions into physical actions. It instructs specific car components to execute the desired movements. For instance, if the planning system determines the car needs to slow down for a red light, the control system will activate the brakes. This process is similar to the function of the cerebellum in humans, which manages basic motor functions.

1.6. Advantages and disadvantages of Self-Driving Cars

The advances in wireless networking, software-defined networking, and information and communication technology (ICT) have been applied to intelligent transportation systems (ITS) to reduce collisions, decrease pollution, ameliorate mobility issues, provide new methods of public transportation, and optimize the sharing of resources, materials, and space. According to studies, there are 1.3 million deaths annually due to drunk, drugged, distracted, and drowsy driving, which could potentially be prevented by autonomous AI systems that eliminate some of the human errors. The following advantages motivate current research in self-driving cars:

Advantages

For Users:

- **Reduced Stress:** Autonomous vehicles can handle the complexities of driving, allowing passengers to relax.
- **Faster Commutes and Reduced Travel Times:** Optimized routing and the elimination of human-related delays can lead to quicker journeys.
- **Enhanced User Productivity:** Passengers can utilize travel time for work or leisure, increasing overall productivity.
- **Optimum Fuel Consumption:** Efficient driving patterns and optimized routes can lead to better fuel economy.
- **Reduced Carbon Emissions:** Autonomous cars can be programmed to drive in ways that minimize emissions.
- **Defensive Driving:** These vehicles can be programmed to drive defensively, avoid blind spots, and adhere strictly to speed limits.

For Governments:

- **Traffic Enforcement Assistance:** Self-driving cars can improve compliance with traffic laws and regulations.
- **Enhanced Roadway Capacity:** Autonomous vehicles can drive more closely and efficiently, increasing the number of vehicles that roads can handle.
- **Reduction in Road Casualties:** Fewer accidents and fatalities are expected due to the elimination of human error.
- **Better Observance of Speed Limits:** Autonomous cars can consistently adhere to speed regulations, enhancing safety.

General Benefits:

- **Elimination of Drunk and Distracted Driving:** Self-driving cars can eliminate issues related to impaired and distracted driving, including texting and other cell phone use.
- **Reduced Braking and Accelerating:** Smooth driving patterns result in less wear and tear on vehicles and roads, and reduce congestion.
- **Beneficial for Vulnerable Populations:** Reduced accidents will particularly benefit children and the elderly, making people more comfortable with autonomous vehicles.
- **Environmental Benefits:** Autonomous electric vehicles can lead to reduced greenhouse gas emissions and noise pollution.
- **Increased Mobility:** Elderly and disabled individuals will experience enhanced mobility.
- **Efficient Use of Space:** Parking lots could be repurposed for parks and other green infrastructure, as self-driving cars can reduce the need for parking spaces.
- **Improved Scheduling and Routing:** Self-driving cars can determine the best routes to improve travel times and lower travel costs.
- **Shared Access and Personalized Transportation:** Although car ownership might decrease, shared autonomous vehicles can provide personalized, efficient, and reliable transportation.

Disadvantages

Technical and Safety Concerns:

- **Complexity of Technology:** Developing and maintaining the technology is complex and expensive.
- **Cybersecurity Risks:** Autonomous vehicles are susceptible to hacking and cyber-attacks.
- **System Failures:** There is a risk of system failures that could lead to accidents.

Economic and Social Impacts:

- **Job Losses:** Widespread adoption of self-driving cars could lead to job losses in driving-related professions, such as truck drivers, taxi drivers, and delivery drivers.
- **High Initial Costs:** The initial cost of self-driving cars is high, potentially limiting accessibility.
- **Legal and Ethical Issues:** There are unresolved legal and ethical issues regarding liability in the event of an accident involving an autonomous vehicle.

Infrastructure Requirements:

- **Upgrading Infrastructure:** Significant investment in infrastructure, such as smart roads and communication systems, is necessary to support autonomous vehicles.
- **Interoperability:** Ensuring that self-driving cars can interact safely with human-driven vehicles is a challenge.

1.7. Probable Disadvantages and Drawbacks of Self-Driving Cars

Cars are among the most widespread and readily available modes of transportation. Despite advancements in technology that have made cars safer, driving remains a hazardous activity. Self-driving cars create a scenario where lines of source code, combined with artificial intelligence (AI), make decisions that impact human lives. Some disadvantages of self-driving cars are outlined as follows:

Disadvantages

Job Elimination:

- The most significant and catastrophic consequence of self-driving cars could be the elimination of jobs in the transportation industry. This includes jobs for truck drivers, taxi drivers, and delivery personnel, which are crucial for many individuals' livelihoods.

Ethical and Societal Acceptance:

- Although the role of AI in our society is continually evolving, AI systems that make critical decisions must respect societal values and conform to social norms to gain widespread acceptance. The acceptance of self-driving technology at philosophical, ethical, and technological levels is a fundamental research problem in psychology and cognitive science.
- There is an argument that if autonomous vehicles and AI systems malfunction, people might be safer if they were in control of the system. Trust in these systems is crucial for their acceptance.

Challenges in Complex Driving Scenarios:

- Navigating intersections without traffic lights, malfunctioning traffic lights, uncontrolled intersections, busy intersections, and areas with significant pedestrian activity presents substantial challenges for self-driving cars.
- Self-driving cars rely on global positioning systems (GPS) for localization. This reliance makes them unsuitable for driving in non-mapped or poorly mapped areas, limiting their functionality in certain environments.

Cybersecurity and Privacy Concerns:

- The connectivity of self-driving cars, being online at all times, makes them susceptible to hacking. This vulnerability could lead to unauthorized control of the vehicle, posing significant safety risks.
- The convenience and safety offered by self-driving cars might come at the cost of passenger privacy. The movements of passengers can be tracked and logged, raising concerns about data privacy and surveillance.

Technical Limitations:

- Despite advances, self-driving technology is still imperfect and can malfunction, potentially leading to accidents. The technology must be robust enough to handle unexpected situations reliably.
- The integration of self-driving cars into the existing traffic ecosystem, which includes human-driven vehicles, pedestrians, and cyclists, poses additional challenges. Ensuring smooth interaction between autonomous and non-autonomous elements is complex.

1.8. Development of Self-Driving Cars

Autonomous cars are vehicles driven by digital technologies without any human intervention. These vehicles are capable of driving and navigating themselves on the roads by sensing the environmental impacts. Their design is optimized to occupy less space on the road, thereby avoiding traffic jams and reducing the likelihood of accidents. Although there have been significant advancements, as of 2017, the automated cars allowed on public roads are not fully autonomous; each requires a human driver who can take control of the vehicle when necessary.

The concept of self-driving cars dates back to the Middle Ages, centuries before the invention of the car itself. Evidence of this can be found in sketches by Leonardo Da Vinci, which contain rough plans for such vehicles. Later, literature and science fiction novels featured robots and vehicles controlled by them.

The first prototypes of driverless cars appeared in the 1920s, though they looked quite different from today's versions. Despite the absence of a driver, these vehicles relied heavily on specific external inputs. One early solution involved a car being controlled by another car behind it. This prototype, known as "the American Wonder" or "Phantom Auto," was introduced in New York and Milwaukee.

Figure 2: Development of Self-Driving Cars

Many prominent automotive companies, including Mercedes-Benz, Audi, BMW, Tesla, and Hyundai, have begun developing or forming partnerships around autonomous technology. These companies have invested significant resources in this area, aiming to lead the market for self-driving cars. Despite numerous advancements in aids, software, and sensors, full autonomy has not yet been achieved.

Self-driving cars utilize lasers, specifically LIDAR (Light Detection and Ranging), to test the environment. This optical technology senses the shape and movement of objects around the car. Combined with digital GPS maps, LIDAR helps detect white and yellow lines on the road, as well as all standing and moving objects within the car's perimeter. Currently, autonomous vehicles can only operate independently if a human driver can take control when necessary.

1.8.1. Current Features of Driverless Cars

Self-driving cars are already equipped with several features, including:

- **Collision Avoidance:** Technology to prevent accidents by detecting and reacting to potential hazards.
- **Lane Departure Warning:** Alerts the driver if the vehicle begins to drift out of its lane.

- **Blind-Spot Detection:** Monitors areas that are not visible to the driver to prevent collisions.
- **Enhanced Cruise Control:** Adjusts the vehicle's speed to maintain a safe distance from other vehicles.
- **Self-Parking:** Enables the car to park itself without human intervention.

1.8.2. Key Players in the Industry

Below, we briefly present some companies playing significant roles in the innovation of this sector:

Tesla Elon Musk, CEO of Tesla, claims that every Tesla car will be completely autonomous within two years. Tesla's Model S is a semi-autonomous vehicle where different cars can learn from each other while working together. The signals processed by the sensors are shared with other cars, enabling continuous improvement in detecting obstacles and changing lanes. Since October 2016, all Tesla vehicles have been built with Autopilot Hardware 2, featuring a sensor and computing package that the company claims will enable complete self-driving without human intervention.

Google The Google team has been working on driverless cars for years, and in recent years, they presented a working prototype. Google also supports other car manufacturers, such as Toyota Prius, Audi TT, and Lexus RX450h, with self-driving car technologies. Their autonomous vehicle uses Bosch sensors and other equipment manufactured by LG and Continental. In 2014, Google planned a driverless car without pedals and steering wheels for the general public by 2020, but current trends suggest that achieving this goal may take longer.

nuTonomy A small group of graduates from the Massachusetts Institute of Technology (MIT) created the nuTonomy software and algorithms specifically for self-driving cars. In Singapore, nuTonomy has equipped a Mitsubishi i-MiEV electric car prototype with sensors, allowing nuTonomy algorithms to control the car on complex urban roads using GPS and LIDAR sensors. Additionally, in November 2016, nuTonomy announced that their self-driving cars would be tested in Boston.

1.9. Current Limitations and Challenges

Sensor Reliability and Accuracy: Self-driving cars heavily rely on a complex array of sensors, including cameras, radar, and lidar, to perceive their environment. Ensuring the reliability and accuracy of these sensors in diverse weather conditions (such as heavy rain, fog, or snow) and road environments (such as poorly marked roads or construction zones) remains a significant challenge.

Algorithmic Decision-Making: The software algorithms that guide the decision-making of self-driving cars must be capable of handling a wide range of complex and unpredictable scenarios. These scenarios include navigating through dense traffic, responding to sudden changes in road conditions, and making split-second decisions to avoid collisions. Developing algorithms that can effectively prioritize safety while considering factors such as traffic laws, road etiquette, and human behavior is crucial.

Cybersecurity Vulnerabilities: Self-driving cars are essentially mobile computers, equipped with a multitude of sensors, processors, and communication systems. This connectivity makes them susceptible to cyber attacks that could compromise the vehicle's safety and security. Ensuring robust cybersecurity measures, including encryption, authentication, and intrusion detection systems, is essential to protect against potential threats.

Ethical Considerations: In the event of an unavoidable accident, self-driving cars must be programmed to make difficult ethical decisions. These decisions may involve prioritizing the safety of passengers or pedestrians, weighing the likelihood of injury or damage, and adhering to legal and moral principles. Addressing these ethical considerations requires careful consideration of societal values, cultural norms, and legal frameworks.

Legal and Regulatory Challenges: The lack of clear and consistent regulations governing the testing and deployment of self-driving cars presents a significant barrier to widespread adoption. Uncertainty surrounding liability, insurance, licensing, and certification requirements creates challenges for manufacturers, developers, and policymakers alike. Establishing comprehensive and harmonized regulatory frameworks is essential to ensure the safe and responsible development and deployment of self-driving technology.

1.10. Proposed Solution

Despite the challenges, researchers and engineers are actively working to address the limitations of self-driving cars and pave the way for their widespread adoption. Some of the proposed solutions and future developments include:

Sensor Fusion and Redundancy:

- **Sensor Fusion:** Combining data from multiple sensors, such as cameras, radar, and lidar, to create a more comprehensive and reliable perception of the environment. By integrating information from diverse sensor modalities, self-driving cars can enhance their ability to detect and respond to various driving scenarios.

- **Redundancy:** Incorporating redundant sensor systems to ensure the continued safe operation of the vehicle in case of sensor failure. Redundancy measures can include backup sensors, fail-safe mechanisms, and redundant processing units to maintain critical functions even in the event of component failure.

Advancements in AI and Machine Learning:

- **Sophisticated Algorithms:** Developing more sophisticated and adaptable algorithms capable of handling a wider range of complex driving scenarios. These algorithms must be able to address edge cases and unpredictable situations, such as adverse weather conditions, road construction, and unusual traffic patterns.
- **Deep Learning and Neural Networks:** Leveraging the power of deep learning and neural networks to improve the decision-making capabilities of self-driving cars. By training AI models on vast amounts of data, self-driving systems can learn to recognize and respond to diverse driving conditions with greater accuracy and efficiency.

Cybersecurity Measures:

- **Robust Protocols:** Implementing robust cybersecurity protocols and safeguards to protect self-driving cars from cyber attacks. This includes ensuring secure communication channels, encrypting data transmissions, and deploying intrusion detection systems to detect and mitigate potential threats.
- **Continuous Monitoring:** Establishing mechanisms for continuous monitoring and updating of cybersecurity measures to adapt to evolving threats and vulnerabilities. Regular security audits, penetration testing, and software updates are essential to maintain the integrity and security of self-driving systems.

Ethical Frameworks and Regulations:

- **Clear Guidelines:** Establishing clear ethical guidelines and regulations that govern the decision-making processes of self-driving cars. These guidelines should prioritize safety and fairness in the event of unavoidable accidents, ensuring that self-driving systems make decisions that align with societal values and legal frameworks.
- **Collaborative Approach:** Fostering collaboration between industry stakeholders, policymakers, and ethicists to develop comprehensive ethical frameworks and regulatory standards. By involving diverse perspectives and expertise, we can create governance mechanisms that promote transparency, accountability, and trust in self-driving technology.

Chapter 2

Potential problems

2.1. Potential Problems with Self-Driving Cars

There is a shared hope among researchers and manufacturers that the introduction of self-driving vehicles will contribute to safer roads. Indeed, an estimated 94 percent of motor vehicle accidents are believed to involve some form of human error. Self-driving cars, operating on computer technology, offer the promise of reducing driving mistakes and, consequently, the occurrence of tragic car crashes.

However, alongside this optimism, there are also concerns surrounding the use of self-driving vehicles. Both existing semi-autonomous cars on our roadways and the fully autonomous models of the future have experts across various fields worried about potential consequences. These concerns encompass several domains:

Safety:

- Despite the potential for improved safety, there are concerns regarding the reliability and robustness of self-driving technology. Issues such as sensor malfunctions, software bugs, and system failures could compromise the safety of self-driving cars and their passengers.

Technology:

- The complex nature of self-driving technology introduces challenges related to development, testing, and validation. Ensuring the accuracy, efficiency, and resilience of autonomous systems requires extensive research, development, and rigorous testing protocols.

Autonomy:

- Questions arise concerning the level of autonomy and decision-making authority granted to self-driving cars. Striking a balance between human oversight and machine autonomy is crucial to ensure safe and responsible operation on public roads.

Social Implications:

- The widespread adoption of self-driving cars could have significant social and economic implications. Concerns include job displacement in industries reliant on driving, changes to urban infrastructure and transportation systems, and ethical considerations surrounding the prioritization of safety and liability in autonomous driving scenarios.

Addressing these concerns requires a collaborative effort involving researchers, engineers, policymakers, and the public. By carefully addressing safety, technology,

autonomy, and social implications, we can work towards realizing the full potential of self-driving technology while minimizing risks and maximizing benefits.

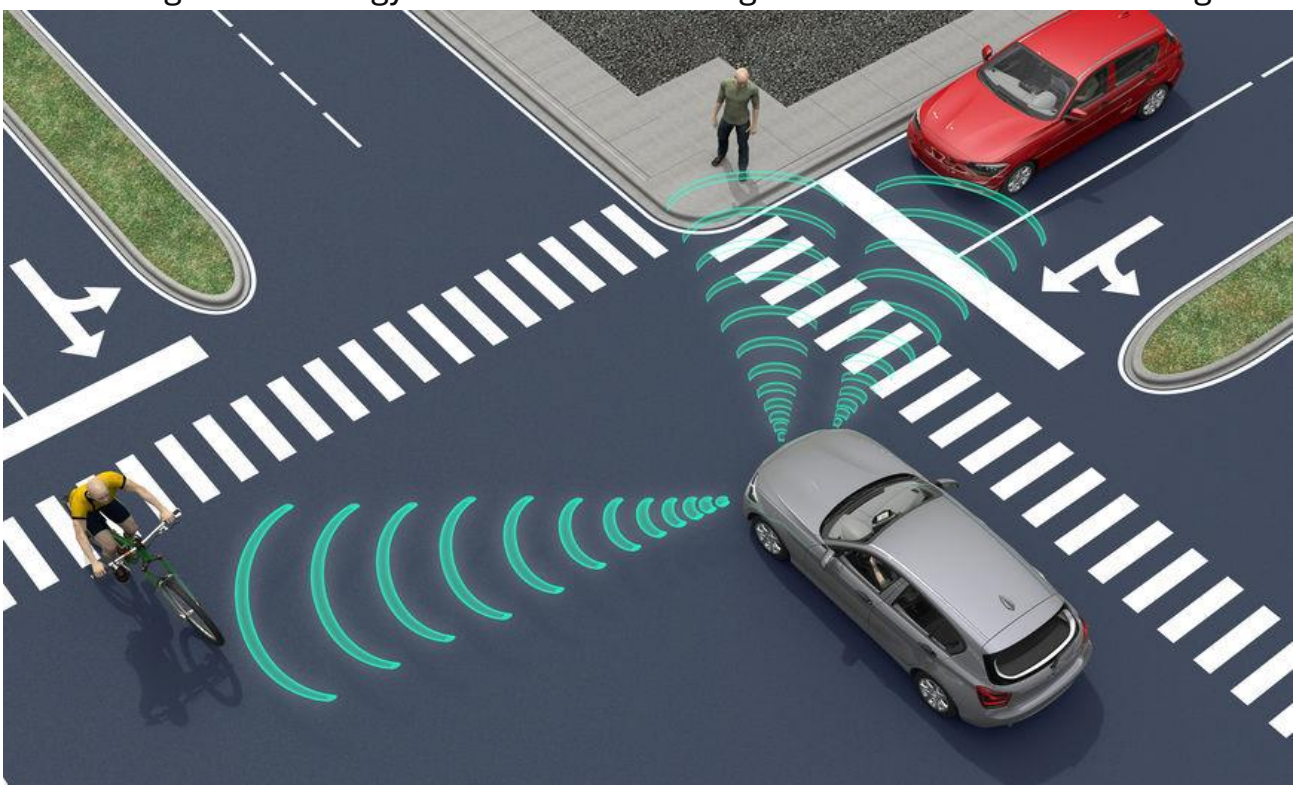


Figure 3: Social Implications

2.2. Workforce Effects

The emergence of self-driving vehicles carries potential implications for the workforce. While self-driving car technology is still in development, many manufacturers envision fully autonomous vehicles performing various functions and jobs, including commercial transportation.

There is growing concern that the workforce comprising drivers of cars, trucks, buses, and taxis may face displacement due to this advancing technology, potentially resulting in significant unemployment. According to the U.S. Bureau of Labor Statistics, more than 2 million individuals are employed as tractor-trailer truck drivers, and over 680,000 people work as passenger vehicle drivers.

The prospect of automation replacing human drivers raises questions about the future of these professions and the livelihoods of those currently employed in the transportation industry. As self-driving technology continues to evolve, it is essential to consider the potential workforce effects and explore strategies to mitigate any adverse impacts on affected workers.

There is growing concern regarding the vulnerability of self-driving vehicles to computer hackers. Like any other computerized system, the software used in driverless cars can potentially be exploited for malicious purposes. A notable incident occurred in 2015 when two individuals remotely hacked into a Tesla Model S car in Las Vegas to test the vehicle’s security measures.

As self-driving cars become more prevalent, there is a heightened risk that hackers could identify and exploit security flaws, potentially gaining unauthorized control over the vehicle. This could pose significant safety risks for passengers and other road users, as hackers could potentially manipulate the vehicle's behavior or access sensitive data stored within its systems.

Addressing cybersecurity vulnerabilities in self-driving cars requires ongoing efforts to enhance software security, implement robust encryption protocols, and develop intrusion detection systems. Additionally, collaboration between automotive manufacturers, cybersecurity experts, and regulatory bodies is essential to establish comprehensive cybersecurity standards and protocols for self-driving vehicles.

2.3. Hackability

There is growing concern regarding the vulnerability of self-driving vehicles to computer hackers. Like any other computerized system, the software used in driverless cars can potentially be exploited for malicious purposes. A notable incident occurred in 2015 when two individuals remotely hacked into a Tesla Model S car in Las Vegas to test the vehicle's security measures.

As self-driving cars become more prevalent, there is a heightened risk that hackers could identify and exploit security flaws, potentially gaining unauthorized control over the vehicle. This could pose significant safety risks for passengers and other road users, as hackers could potentially manipulate the vehicle's behavior or access sensitive data stored within its systems.

Addressing cybersecurity vulnerabilities in self-driving cars requires ongoing efforts to enhance software security, implement robust encryption protocols, and develop intrusion detection systems. Additionally, collaboration between automotive manufacturers, cybersecurity experts, and regulatory bodies is essential to establish comprehensive cybersecurity standards and protocols for self-driving vehicles.

2.4. Safety

One of the primary concerns surrounding self-driving cars is their safety. While driverless vehicles rely on advanced technology to perceive and navigate their surroundings, there remains a significant challenge in ensuring their safety on the roads. These vehicles utilize perception and decision-making technology to process their environment and make judgment calls.

However, according to some reports, existing self-driving cars may "incorrectly perceive something in their environment once every tens of thousands of hours." This highlights a potential vulnerability that could contribute to safety issues and increase the risk of car accidents. In essence, the technology powering self-driving cars is not flawless, and there is a need for continuous improvement and refinement.

Addressing safety concerns in self-driving cars requires rigorous testing, validation, and refinement of the technology. This includes developing robust perception systems, enhancing decision-making algorithms, and implementing fail-safe mechanisms to mitigate the risk of accidents or malfunctions. Additionally, regulatory oversight and industry standards play a crucial role in ensuring the safety and reliability of self-driving vehicles on public roads.

2.5. Autonomy

Current self-driving technology often requires human intervention or oversight in certain situations, limiting the extent of autonomy in these vehicles. Drivers may be prompted to keep their hands on the steering wheel or remain attentive to the road, particularly during critical moments. As a result, self-driving cars are not fully autonomous, and the potential for human error still exists.

Semi-autonomous cars available on the market have been involved in fatal crashes while using driving assistance systems. For example, Tesla, a prominent manufacturer, has integrated semi-autonomous driving assistance technology, known as Autopilot, into their vehicles.

In 2019, Tesla faced a lawsuit after a driver of a Model S car crashed into a tractor-trailer on a Florida highway and died shortly after activating Autopilot. Similarly, in 2020, a Tesla vehicle in Pleasanton, CA, veered over a curb and collided with a brick wall, resulting in the death of the driver. These tragic incidents underscore the potential safety risks associated with autonomous software and highlight the need for further development and refinement of self-driving technology.

Addressing concerns about autonomy in self-driving cars requires advancements in perception systems, decision-making algorithms, and fail-safe mechanisms to enhance safety and reliability. Additionally, clear guidelines and regulations are essential to ensure responsible deployment and use of autonomous driving systems on public roads.

2.6. What if I'm Injured in a Self-Driving Car Accident?

Whether you drive a semi-autonomous self-driving vehicle or utilize a driverless transportation service, there's always a risk of being involved in a car crash.

The circumstances surrounding a self-driving car accident may differ from those of a typical motor vehicle accident. For instance, if you sustain injuries while a self-driving vehicle is in autonomous mode, the manufacturing company may be deemed at fault. Individuals who suffer injuries in a self-driving vehicle accident may be entitled to compensation for various damages, including:

- **Medical Expenses:** Coverage for medical treatment, hospital bills, rehabilitation costs, and other healthcare expenses incurred as a result of the accident.
- **Future Medical Care:** Compensation for anticipated medical care and treatment needed in the future due to injuries sustained in the accident.

- **Lost Wages:** Reimbursement for income lost during the recovery period, including wages, salaries, and other forms of income lost due to an inability to work.
- **Pain and Suffering:** Compensation for physical pain, emotional distress, and mental anguish resulting from the accident and associated injuries.
- **Wrongful Death:** In cases where the accident results in the loss of life, compensation may be available to surviving family members for the wrongful death of their loved one.

Seeking compensation for damages resulting from a self-driving car accident may involve complex legal processes. It's advisable to consult with a personal injury attorney who can assess your case, determine liability, and advocate on your behalf. The legal team at William Mattar can provide guidance and representation to help you pursue the compensation you deserve.

2.7. Consult a Self-Driving Car Accident Attorney Today

If you've sustained injuries in a self-driving car accident, William Mattar is here to help. Our team of experienced legal professionals is dedicated to guiding you through the legal process and helping you secure the financial recovery you deserve.

To discuss your case and explore your legal options, call us at (716) 638-2422 today to speak with an attorney. Alternatively, you can schedule a free initial consultation by filling out our online form.

Don't navigate the complexities of a self-driving car accident alone. Let William Mattar provide the expert legal representation and support you need to pursue justice and compensation.

Chapter 3

Background

3.1. Artificial Intelligence (AI)

Artificial intelligence (AI) represents a cutting-edge field in science and engineering, focusing on developing machines capable of performing tasks that typically require human intelligence. Intelligence can be measured in terms of rationality, fidelity to human understanding, critical thinking, and reasoning. AI encompasses various domains, including:

- **Human speech understanding**
- **Strategic gaming systems**, such as chess
- **Imperfect information systems**, like self-driving cars, content supply networks, and military simulations

3.1.1. Autonomous Vehicles

Autonomous vehicles, or self-driving cars, have gained prominence in recent years, offering numerous benefits to society, including improved travel efficiency and enhanced safety. AI plays a crucial role in autonomous vehicles, enabling them to navigate and operate independently on roads. The development of AI-powered vehicles requires anticipatory research to address challenges and ensure safe and efficient transportation systems.

AI techniques have experienced a resurgence in the 21st century due to advancements in computer capabilities and the availability of massive amounts of data. These techniques have found applications in various industries, including computer science, operations research, and software engineering.

3.1.2. History of Autonomous Vehicle Technology

The journey of autonomous vehicle technology spans centuries, beginning with early conceptual designs and evolving into modern advancements. Key milestones include:

Early Concepts:

- **Leonardo da Vinci**: In the 15th century, da Vinci conceptualized a self-propelled cart powered by springs, demonstrating early ideas of autonomous motion.

19th Century:

- **Robert Whitehead (1868)**: Invented a torpedo capable of propelling itself underwater, laying the groundwork for autonomous propulsion systems.

20th Century:

- **Mechanical Mike (1933)**: The Gyroscope Co. developed an aircraft autopilot system, showcasing early attempts at automated flight control.
- **Ralph Teetor (1945)**: Invented cruise control for automobiles, laying the foundation for automated driving assistance systems.
- **James Adams (1961)**: Developed a self-driving wheeled vehicle equipped with cameras, demonstrating early attempts at autonomous navigation.
- **Tsukuba Mechanical Engineering Company (1977)**: Designed an autonomous passenger vehicle capable of recognizing street markings, marking significant progress in autonomous navigation.
- **Ernst Dickmanns (1987)**: Developed micro-processing modules for object detection in street environments, contributing to the advancement of autonomous vehicle sensing technologies.

Late 20th Century to Present:

- **General Atomics MQ-1 Predator (1995)**: Introduced crewless aircraft technology with advanced sensing capabilities, paving the way for autonomous vehicle systems.
- **DARPA Challenges (2004-2013)**: The U.S. Department of Defense Research Arm funded competitions to spur innovation in autonomous vehicle technology, leading to significant advancements in navigation and control systems.
- **Tesla Autopilot (2015)**: Tesla introduced a semi-autonomous driving feature, enabling hands-free control and freeway driving with a single software update, marking a significant milestone in consumer autonomous vehicle technology.
- **University of Michigan's MCity (2015)**: Launched as a world-class test organization for autonomous vehicle technology, providing a controlled environment for testing and development.
- **Ford Testing (2015)**: Ford became one of the first companies to test autonomous vehicles, contributing to the rapid growth of the autonomous vehicle industry.

3.1.3. Details and Description of AI in Driverless Cars

AI technologies play a crucial role in powering the advanced functionalities of driverless car systems. Manufacturers harness vast amounts of data from various sources, including image recognition structures, neural networks, and AI algorithms, to develop autonomous driverless systems.

Sensor Fusion and Perception:

- **Neural Networks**: These networks analyze pattern data, including images captured by interior cameras, to identify and understand the vehicle's

surroundings. Key elements such as curbs, pedestrians, street signs, traffic lights, and obstacles are recognized and processed in real-time.

Waymo Google Autonomous Car Project:

- **Sensor Integration**: The Waymo Google autonomous car project utilizes a combination of devices, including lidar, sensors, and cameras, to perceive its environment. Data generated by these sensors is fused to create a comprehensive understanding of the surroundings.
- **Predictive Analysis**: The system forecasts the actions of surrounding objects and anticipates their movements, enabling proactive decision-making in navigating complex driving scenarios.

Operation and Control:

- **Navigation**: Passengers set destinations, and the vehicle calculates the optimal route using data from lidar devices and 3D mapping. The system continuously updates the vehicle's location relative to its environment, ensuring precise positioning.
- **AI Software Integration**: All sensors are interconnected with AI software, which processes input from interior video cameras and external sources like Google Street View. This integration enables the system to mimic human perception and make decisions autonomously.
- **Safety Features**: Driverless cars incorporate safety features such as hands-free steering, adaptive cruise control (ACC) with automatic braking, and lane-centering steering. These features enhance safety and driving comfort while providing the driver with the ability to override system controls if necessary.

3.1.4. Advantages of AI in Driverless Cars

- **Improved Road Safety**: Driverless cars equipped with AI technology have the potential to significantly reduce road accidents by eliminating human error, which is a major contributing factor to crashes. AI-powered systems can make split-second decisions based on real-time data, leading to safer driving conditions.
- **Reduced Traffic**: Driverless cars can optimize traffic flow through efficient route planning and coordination, leading to reduced congestion and shorter travel times for commuters. AI algorithms can analyze traffic patterns and adjust driving behavior to minimize delays and bottlenecks.
- **Convenient Parking**: AI-enabled driverless cars can autonomously navigate to parking spaces, eliminating the need for drivers to search for parking spots. This can lead to more efficient use of urban space and reduce traffic congestion in crowded areas.

- **Environmental Benefits**: Driverless cars can be programmed to operate more efficiently, leading to reduced fuel consumption and lower emissions. By optimizing driving behavior and route planning, AI technology can contribute to environmental sustainability and mitigate the impact of transportation on climate change.

3.2. Deep Learning

Deep learning plays a crucial role in the development of self-driving technology, enabling artificial neural networks to perform various tasks related to autonomous driving. Here's a breakdown of how deep learning is applied in self-driving systems:

1. **Perception**
    - Deep learning models interpret sensor data, such as images from cameras, point clouds from lidar, and radar signals, to identify objects like pedestrians, vehicles, traffic signs, and road markings.
    - Convolutional Neural Networks (CNNs) are commonly employed for image-based perception tasks due to their ability to learn hierarchical features.
2. **Object Detection and Segmentation**
    - Deep learning models are trained to detect and segment objects in the environment using algorithms like YOLO and Faster R-CNN for real-time object detection and classification.
3. **Localization and Mapping**
    - Deep learning techniques aid in localization and mapping by integrating sensor data with a map of the environment using networks like RNNs or LSTM for sequential data processing tasks like sensor fusion and localization.
4. **Prediction**
    - Recurrent neural networks and sequence-to-sequence models are used to forecast future trajectories of other agents in the environment, such as pedestrians and vehicles, based on historical data and current observations.
5. **Decision Making**
    - Deep reinforcement learning (DRL) can be employed for decision-making tasks, allowing the vehicle to learn optimal driving policies in complex scenarios by interacting with the environment and receiving rewards or penalties based on its actions.
6. **Control**
    - Deep learning models are used to directly control vehicle actions such as steering, acceleration, and braking, often trained using reinforcement learning or imitation learning techniques.
7. **Simulation and Training**

- Deep learning models are extensively trained in simulation environments before being deployed in real-world scenarios, enabling safe and efficient exploration of various driving scenarios and conditions.

Levels of Automation in Self-Driving Cars

When discussing self-driving cars, it's important to consider the levels of automation, which indicate the proportion of driving performed by a computer as compared to a human. The levels of automation are as follows:

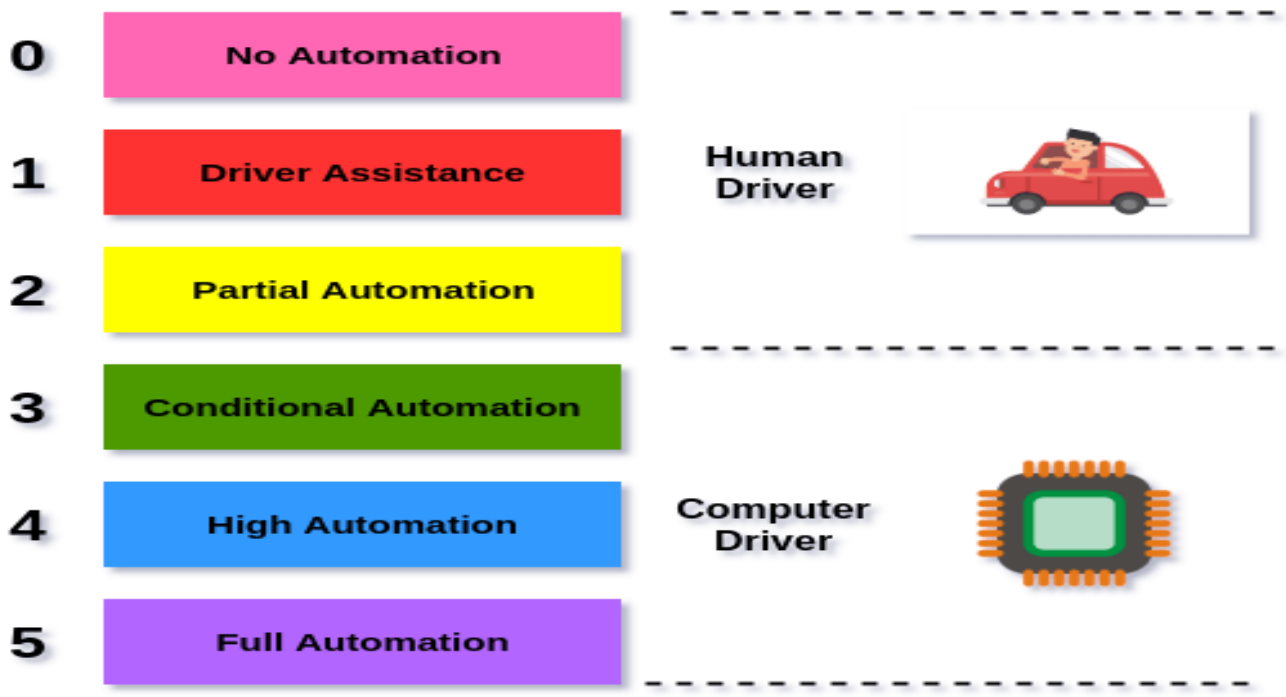


Figure 4: Levels of Automation

- Level 0: All car features and systems are controlled by humans.
- Level 1: Minor functions can be controlled by the computer one at a time.
- Level 2: The computer can perform two or more automated functions simultaneously, but a human is always needed for safe operation and emergency procedures.
- Level 3: The computer can simultaneously control all critical car operations under most conditions, but a human driver should always be present.
- Level 4: The car is fully autonomous in certain driving scenarios, with no need for a human driver.
- Level 5: The car is completely capable of driving autonomously in all situations.

Current Status Most self-driving cars in the news today, such as those made by Tesla and Waymo, are at level 2. While they can drive fairly well on their own, a human driver is still required to ensure the safe operation of the vehicle.

Chapter 4

CANBUS

4.1. Introduction

- The Controller Area Network Protocol, developed by Bosch during the late 1980s, serves as a pivotal communication standard.
- CAN operates as a half-duplex Asynchronous serial communication protocol, primarily facilitating communication among Electronic Control Units (ECUs) within vehicles, such as the engine control unit and transmission unit.

What is an ECU?

- Within an automotive CAN bus system, an ECU refers to a microcontroller responsible for overseeing the functions of various vehicle components, such as the engine, airbags, and audio systems. Each ECU harbors information essential for network-wide sharing.

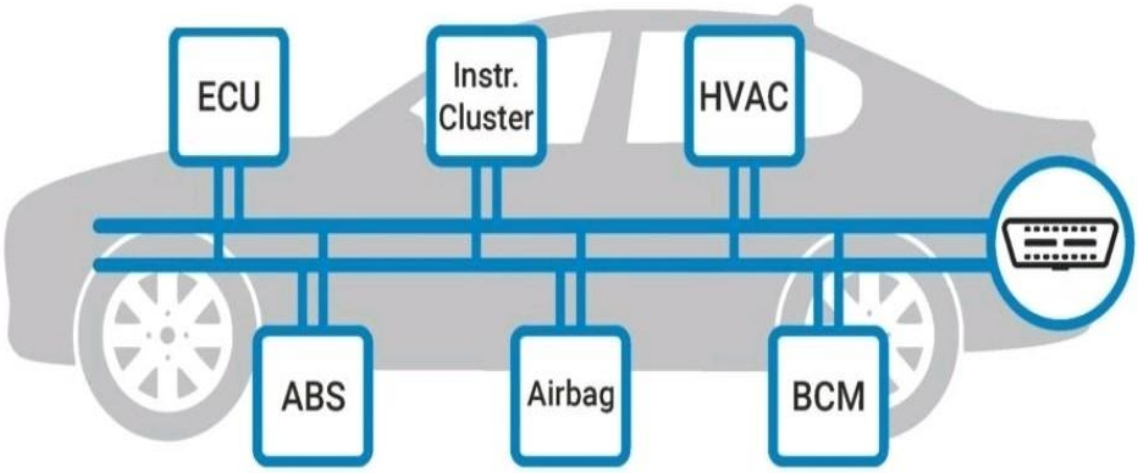


Figure 5: ECU

Top 4 Benefits of CAN Bus

1. **Simplicity and Cost Efficiency**: ECUs interact via a unified CAN system, replacing intricate analog signal lines. This integration minimizes errors, reduces weight, simplifies wiring, and slashes costs.
2. **Robustness**: CAN systems exhibit robust resistance to electrical disturbances and electromagnetic interference, rendering them ideal for safety-critical applications, particularly in vehicles.
3. **Accessibility**: The CAN bus serves as a singular access point for communicating with all networked ECUs, facilitating centralized diagnostics, data logging, and configuration.
4. **Efficiency**: Prioritization of CAN frames by ID ensures immediate bus access for top-priority data, preventing interruptions to other frames or causing CAN errors.

CAN Bus and OSI Model

- The OSI (Open Systems Interconnection) Model provides a conceptual framework for delineating the functions of a networking system.

Types of CAN

1. **High-Speed**: Utilizes 2 wires.
2. **Mid-Speed**: Also employs 2 wires.
3. **Low-Speed**: Operates with a single wire.

4.2. The CAN Bus Physical Layer

- The Controller Area Network Protocol was developed by Bosch in the late 1980s.
- CAN operates as a half-duplex Asynchronous serial communication protocol.
- Primarily utilized for communication among Electronic Control Units (ECUs) within vehicles, such as the engine control unit and transmission unit.
- CAN Bus is characterized by its robustness, low cost, high speed, noise immunity, and error detection features.

What is an ECU?

- In an automotive CAN bus system, ECUs encompass components such as the engine control unit, airbags, and audio systems, each with information requiring sharing across the network.
- The ECU functions as a microcontroller governing the operations of an internal combustion engine and making decisions accordingly.

Top 4 Benefits of CAN Bus

1. **Simplicity & Cost Efficiency**: ECUs communicate via a single CAN system, obviating the need for complex analog signal lines. This reduces errors, weight, wiring, and costs.
2. **Robustness**: CAN systems exhibit robust resistance to electric disturbances and electromagnetic interference, making them ideal for safety-critical applications, particularly in vehicles.
3. **Easy Access**: The CAN bus serves as a central entry point for communication with all networked ECUs, facilitating centralized diagnostics, data logging, and configuration.
4. **Efficiency**: CAN frames are prioritized by ID, ensuring immediate bus access for top-priority data without interrupting other frames or causing CAN errors.

CAN Bus and OSI Model

- The OSI (Open Systems Interconnection) Model provides a conceptual framework for delineating the functions of a networking system.

Types of CAN

1. **High-Speed**: Utilizes 2 wires.
2. **Mid-Speed**: Also employs 2 wires.
3. **Low-Speed**: Operates with a single wire.

4.3. The CAN Bus Data Link Layer

CAN Network Topology:

- CAN network operates on a Broadcast type topology, unlike traditional networks such as I2C, USB, or Ethernet. Messages are globally broadcasted within the network, and all nodes can hear these messages without requiring specific addressing.
- Each unit connected to the Bus can both transmit and receive messages over the CAN bus (Multi-Master).
- It is a Half-duplex system, meaning devices cannot send and receive simultaneously.
- Asynchronous communication: The transmitter and receiver do not share a common clock signal.

Carrier Sense Multiple Access Networks:

- In a Carrier Sense Network, each node must sense the bus before transmitting any data to minimize the chance of collision. This concept is commonly known as "nodes Listen before Talk." The bus has two states:
    - Carrier Busy: Transmission is ongoing in the bus.
    - Carrier Idle: No transmission is currently taking place on the bus.
- Despite the sensing mechanism, the possibility of collision exists due to propagation delay. A node may sense the bus as idle because it has not received the first bit sent by another node yet.

CSMA with Collision Detection:

- CSMA/CD: In this method, if two nodes sense the bus as idle and begin transmitting simultaneously, both nodes detect the collision immediately. They should then cease transmission, wait for a random period, and attempt to send data again. This method is commonly used in Ethernet LAN networks.
- CSMA/CD + AMP (Arbitration on Message Priority): It incorporates collision detection and arbitration based on message priority.

- CAN Network operates using CSMA/CD + AMP:
    - When two or more nodes initiate message transmission simultaneously, collision detection occurs.
    - The message with the highest priority will overwrite other messages using bit-wise arbitration. Consequently, only the highest priority message remains on the bus

4.3. The CAN Bus Data Link Layer

CAN Message Types:

- **Data Frame**: This is the most common message type used for actual data transmission.
- **Remote Frame**: While data transmission typically occurs autonomously, a destination node can request data from a source node by sending a Remote Frame.
- **Error Frame**: Special messages transmitted when a node detects an error in a message. Upon detection, the original transmitter automatically retransmits the message.
- **Overload Frame**: This frame, though less common, is transmitted by a node that becomes too busy. It resembles the error frame in format and is primarily used to introduce an additional delay between CAN Message Format messages.

CAN Message Format:

- **Start of Frame (SOF)**: A single dominant bit marks the beginning of a message, synchronizing the nodes on the bus after being idle.
- **Identifier**: Standard CAN uses an 11-bit identifier to establish message priority. Lower binary values indicate higher priority.
- **RTR (Remote Transmission Request)**: A dominant bit signifies that information is required from another node. Responding data is received by all nodes, with the identifier determining the specified node.
- **R (Reserved Bit)**: Reserved for potential use in future standard amendments.
- **DLC (Data Length Code)**: A 4-bit code indicating the number of bytes of data being transmitted. Up to 64 bits of application data may be transmitted.
- **CRC (Cyclic Redundancy Check)**: A code generated by the transmitter for error detection.
- **CRC Delimiter**: A recessive delimiter marking the end of the CRC field. It serves to provide synchronization time and plays a specific role in error detection.

- **ACK (Acknowledgment)**: The transmitter sends a recessive bit, and any receiver can assert dominance.

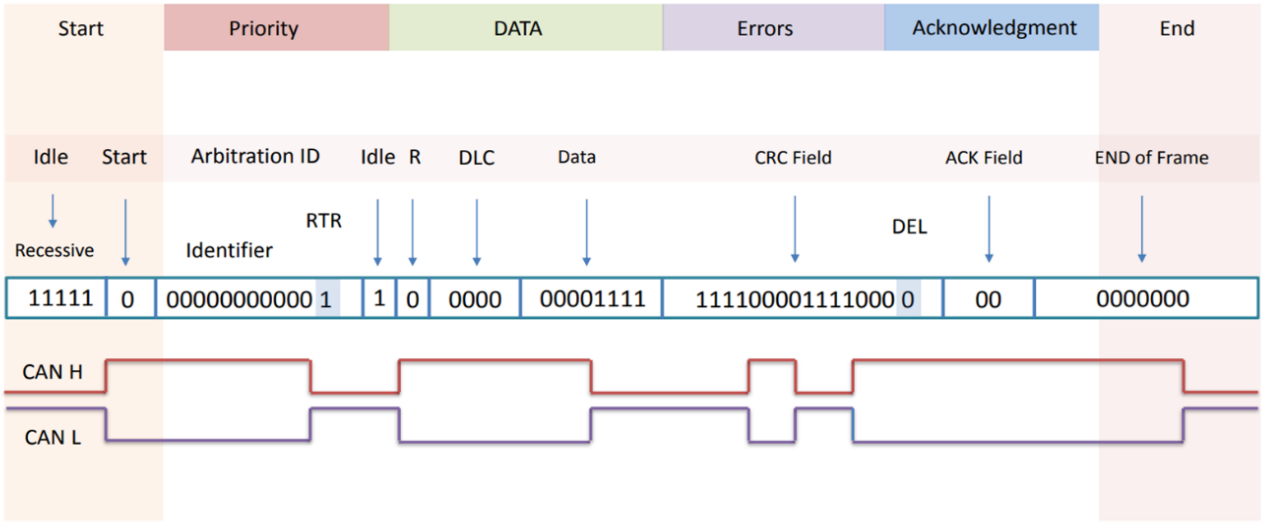


Figure 6: CAN Message Format

CRC Field:

- CAN utilizes a 15-bit CRC code for error detection, generated using the polynomial: x^15 + x^14 + x^10 + x^8 + x^7 + x^4 + x^3 + 1.
- The sender calculates and produces CRC1, while the receiver calculates CRC2. If CRC1 and CRC2 match, the data is deemed valid; otherwise, it is considered corrupt, and the recipient requests a message resend.

4.4. The CAN Bus Security

Types of Threats and Vulnerabilities:

1. **Unauthorized Access:**
   - Intruders gaining unauthorized access to the CAN bus network.
2. **Message Manipulation:**
   - Falsification or injection of unauthorized messages to manipulate system behavior.
3. **Denial-of-Service (DoS) Attacks:**
   - Flooding the bus with excessive messages to disrupt normal communication.
4. **Physical Layer Attacks:**
   - Manipulation of physical components or signals to disrupt or intercept communication.

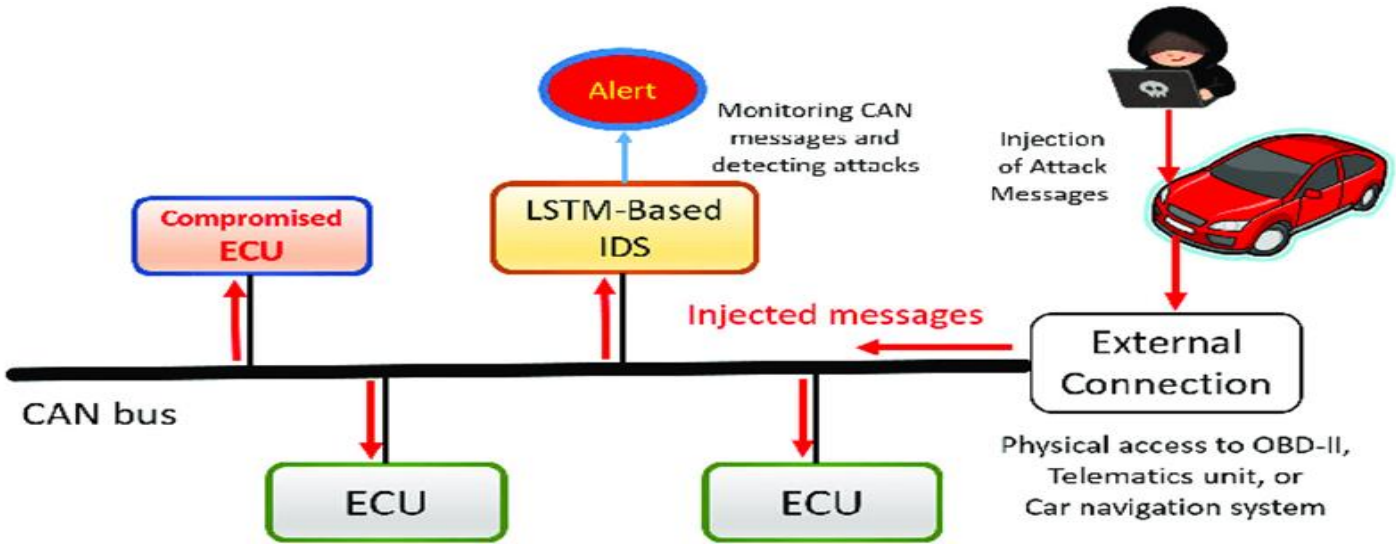


Figure 7: CAN Bus Security

Security Measures:

1. **Access Control:**
   - Implementing access controls such as authentication mechanisms to restrict unauthorized access to the CAN bus network.
2. **Encryption:**
   - Encrypting data transmitted over the CAN bus to ensure confidentiality and integrity.
3. **Intrusion Detection Systems (IDS):**
   - Deploying IDS to monitor bus traffic for suspicious activities and potential security breaches.
4. **Firewalls:**
   - Installing firewalls to filter and control the traffic entering and leaving the CAN bus network.
5. **Secure Coding Practices:**
   - Following secure coding practices to minimize vulnerabilities in CAN bus firmware and software.
6. **Physical Security:**
   - Securing physical access to CAN bus components to prevent tampering or unauthorized access.
7. **Regular Audits and Updates:**
   - Conducting regular security audits and updates to identify and patch vulnerabilities in the CAN bus network.

4.4.1. Types of Attacks

When an attacker gains access to the Controller Area Network (CAN) bus, several types of attacks can compromise communication integrity, confidentiality, or availability:

1. **Message Manipulation or Injection:**
    - **Spoofing:** Sending falsified sensor data or injecting malicious commands by impersonating a legitimate node.
    - **Replay Attacks:** Recording and retransmitting legitimate messages to manipulate system behavior.
2. **Denial-of-Service (DoS) Attacks:**
    - **Flooding:** Overwhelming the CAN bus with a high volume of messages to disrupt normal communication.
    - **Bus-off Attacks:** Forcing specific nodes into an inoperable state by transmitting faulty or malformed messages.
3. **Physical Layer Attacks:**
    - **Electromagnetic Interference (EMI):** Introducing electromagnetic interference to disrupt CAN bus communication.
    - **Voltage Attacks:** Disturbing or damaging electronic components by introducing abnormal voltage levels.

Securing the CAN bus against unauthorized access and implementing measures to detect and mitigate potential attacks is crucial for maintaining system security and reliability, particularly in automotive and industrial contexts.

Attacks on CAN Bus:

1. **Bus Flood Attack:**
    - **Description:** Flooding the CAN bus with messages to overwhelm the network.
    - **Objective:** Disrupting normal communication by consuming available bandwidth.
    - **Impact:** Performance degradation and potential malfunctions in systems relying on timely communication.
2. **Simple Frame Spoofing:**
    - **Description:** Sending unauthorized messages by mimicking legitimate ones.
    - **Objective:** Injecting false information or commands into the network.
    - **Impact:** Incorrect system behavior due to nodes acting on malicious data.
3. **Adaptive Spoofing:**
    - **Description:** Dynamically adjusting attack strategy based on system responses.

- **Objective:** Evading detection and more effectively manipulating the system over time.
- **Impact:** Increased difficulty in detecting and preventing malicious activity.

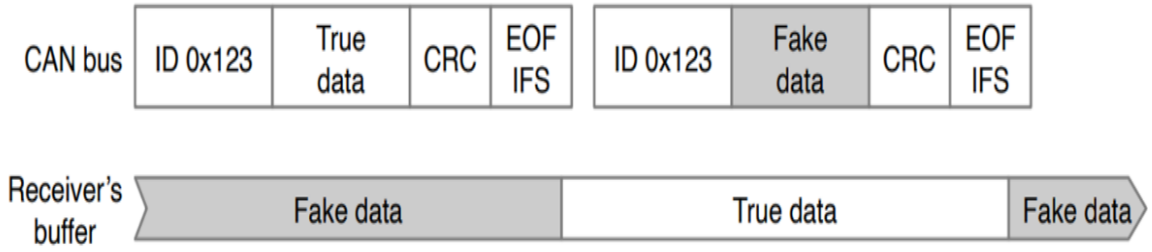


Figure 2: maximising the chances of a receiver seeing fake data

Figure 7: Adaptive Spoofing

4. **Error Passive Spoofing Attack:**
   - **Description:** Introducing errors into spoofed messages to mimic node communication issues.
   - **Objective:** Disrupting normal bus activity and potentially forcing nodes into an error-passive state.
   - **Impact:** Confusion and reduced reliability in the CAN bus, affecting systembehavior.

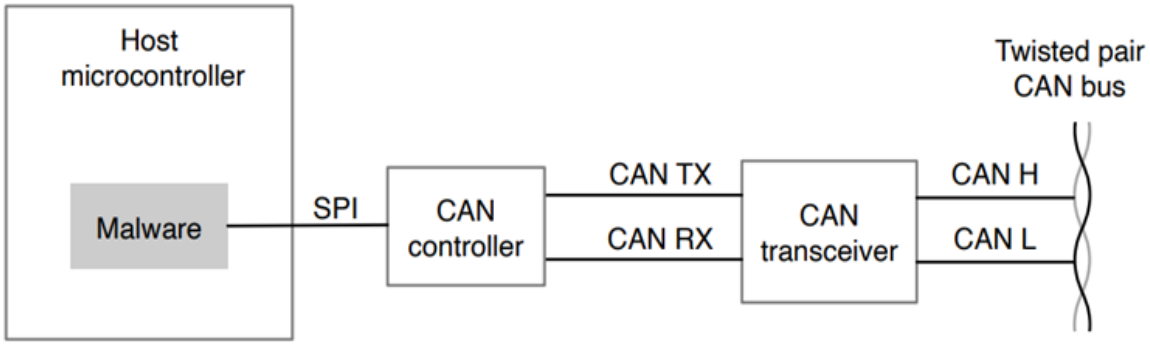


Figure 3: An external CAN controller with malware unable to directly access CAN TX and RX pins

Figure 8: Error Passive Spoofing Attack

5. **Wire-Cutting Spoofing Attack:**
   - **Description:** Physically intercepting the CAN bus wires to inject malicious messages.
   - **Objective:** Gaining unauthorized access to the bus for manipulation.
   - **Impact:** Unauthorized control over the vehicle or system, compromising safety and functionality.

6. **Double Receive Attack:**
    - **Description:** Injecting a message into the bus while mimicking it from a legitimate node.
    - **Objective:** Creating confusion by introducing redundancy, making it challenging for nodes to distinguish between legitimate and malicious messages.
    - **Impact:** Potential errors in decision-making due to conflicting or redundant information.

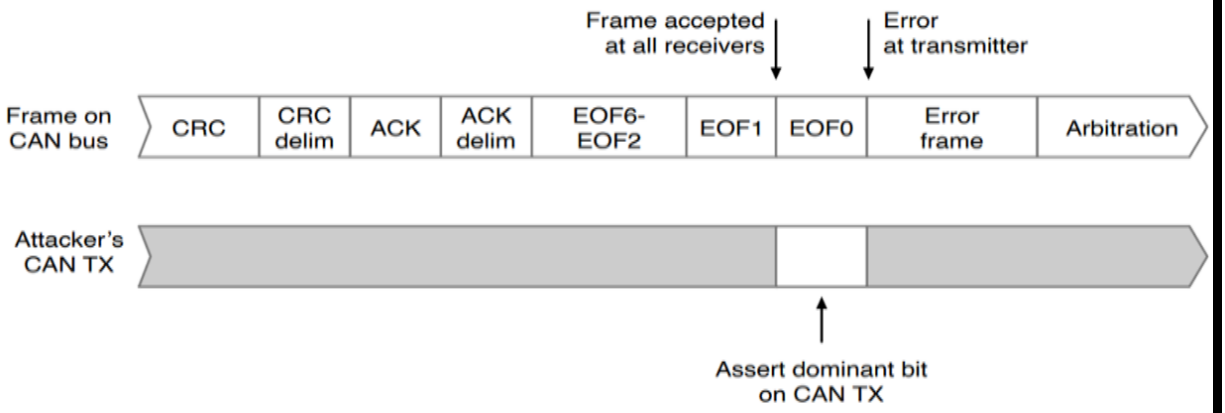


Figure 7: Asserting a dominant bit in EOF0 to trigger a double receive

Figure 9: Double Receive Attack

7. **Freeze-Frame (Doom Loop) Attack:**
    - **Description:** Sending a continuous stream of identical messages with a high-priority identifier to monopolize the bus.
    - **Objective:** Creating a denial-of-service situation by monopolizing bandwidth.
    - **Impact:** Delays or loss of critical messages, affecting system performance and safety.

Securing the CAN bus against these attacks involves implementing robust authentication mechanisms, encryption protocols, intrusion detection systems, and physical security measures to prevent unauthorized access to the bus and mitigate potential threats.

4.4.2. Attack Mitigation Techniques

In addressing potential threats to the Controller Area Network (CAN) bus, several effective mitigation techniques can be employed:

- **Intrusion Detection:** Intrusion detection involves the scrutiny of bus traffic to identify abnormal behaviors that may indicate a potential attack. While it is an essential tool for monitoring and identifying security breaches, it's crucial to note that without hardware support, intrusion detection alone may not be sufficient to prevent an attack. However, it serves critical roles in intelligence gathering and post-incident forensics, aiding in understanding and mitigating the impact of security breaches.
- **Security Gateway:** A security gateway implements a hardware-based approach using a device equipped with multiple CAN bus interfaces. This device selectively copies legitimate traffic between a trusted bus, often a vehicle control network, and an untrusted bus containing potentially compromised devices. By segregating traffic between trusted and untrusted domains, security gateways can prevent malicious actors from infiltrating critical systems while still allowing necessary communication.
- **Encryption:**
  Encryption is primarily a software-based technique, although it may be supplemented by hardware assistance. In this approach, an Electronic Control Unit (ECU) encrypts its CAN bus communications using cryptographic methods. Only receivers possessing the appropriate decryption key can decode and authenticate the message's validity. However, it's important to acknowledge that practical implementation of encryption for CAN protection faces several challenges, including key management, performance overhead, and compatibility issues with existing hardware and software systems.
- **CAN Security Hardware:** These methodologies involve the integration of dedicated hardware devices onto a Printed Circuit Board (PCB). These devices monitor CAN signals to and from the CAN bus, providing various levels of protection against potential security breaches. CAN security hardware solutions offer real-time monitoring and response capabilities, enhancing the overall security posture of the CAN bus system. However, it's essential to ensure compatibility with existing infrastructure and consider potential deployment challenges, such as integration complexity and cost-effectiveness.

Intrusion Detection Systems (IDS) serve as critical components in fortifying the security of the Controller Area Network (CAN) bus. They monitor and analyze network traffic, promptly identifying any suspicious or malicious activities. Here, we delve into the functionalities of IDS within the CAN bus context, focusing on three key components:

Real-Time Traffic Analysis:

- **Description:** Real-time traffic analysis entails the continuous monitoring of live data traversing the CAN bus, discerning any abnormal patterns or behaviors indicative of intrusion attempts.
- **How it works:** IDS conducts ongoing assessments of CAN bus traffic, comparing observed patterns against established norms. Any deviations, such as unexpected surges in message frequency or irregular message identifiers, trigger alerts or alarms.
- **Benefits:** This approach enables swift detection of anomalous behavior, facilitating prompt responses to potential security threats.

Payload Analysis:

- **Description:** Payload analysis involves scrutinizing the content of CAN messages, specifically focusing on the transmitted data (payload).
- **How it works:** IDS inspects message payloads for discernible patterns or signatures associated with known attack vectors. This encompasses identifying unauthorized commands, malicious data injections, or attempts to exploit protocol vulnerabilities.
- **Benefits:** Payload analysis enhances the capability to detect attacks involving manipulation of CAN message content, offering a deeper level of inspection beyond structural message attributes.

Hardware Support:

- **Description:** Hardware-supported intrusion detection involves integrating security functionalities directly into CAN bus hardware or leveraging dedicated hardware components to bolster IDS capabilities.
- **How it works:** Specialized hardware may be employed to offload certain security tasks, alleviating processing burdens on primary control units. This includes dedicated security modules for encryption, authentication, and monitoring.
- **Benefits:** Hardware-supported IDS enhances overall performance and efficiency, providing a robust defense against attacks attempting to compromise or circumvent software-based security measures.

Additionally, addressing the detection of false data from sensors, such as cameras, due to attacks can be approached through the utilization of deep learning models tailored

for anomaly detection. Here's a generalized guide:

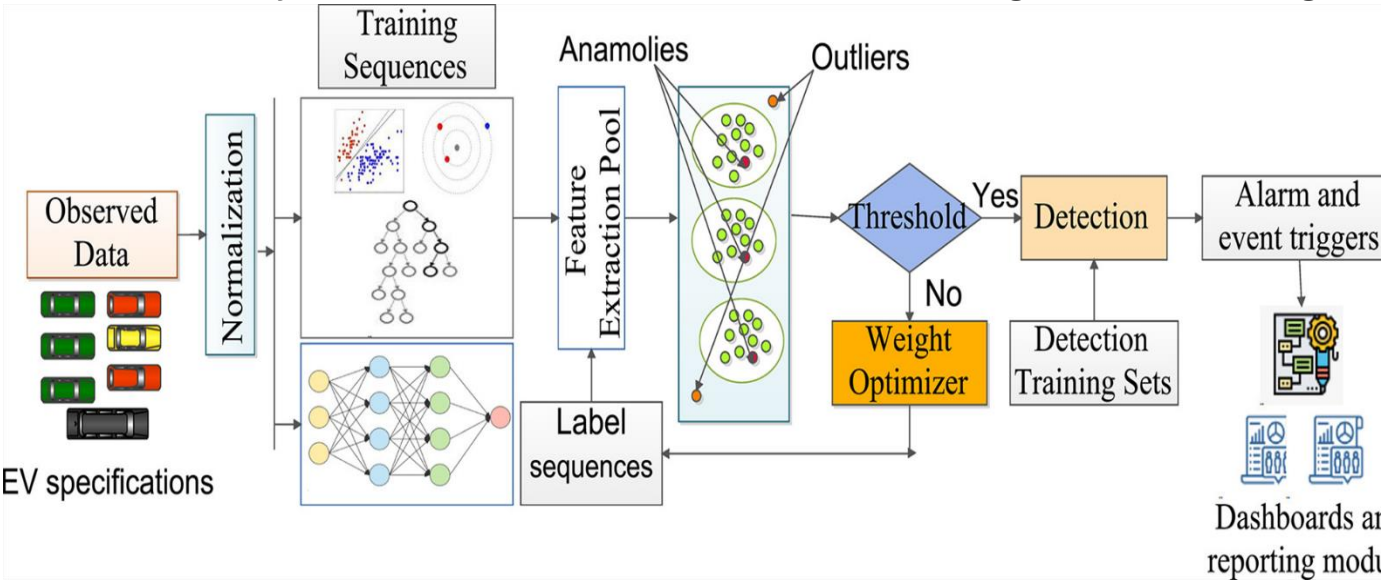


Figure 10: anomaly detection

1. **Data Collection:** Assemble a diverse dataset comprising both normal and anomalous scenarios, including simulated attacks on the camera system, introducing manipulated or false data.
2. **Feature Extraction:** Extract pertinent features from camera data to capture characteristics like color distribution, object shapes, and textures, crucial for identifying anomalies.
3. **Model Training:** Train a deep learning model on the labeled dataset, prioritizing learning normal camera behavior. The model learns to reconstruct normal images and identify anomalies introduced during attacks.
4. **Validation and Tuning:** Validate the model's performance using a separate dataset containing both normal and attacked scenarios. Fine-tune the model to enhance its ability to discern false data from genuine sensor inputs.
5. **Real-Time Monitoring:** Integrate the trained model into the self-driving car's system for real-time monitoring of camera data. Continuously analyze incoming frames to promptly identify anomalies and potential security breaches.

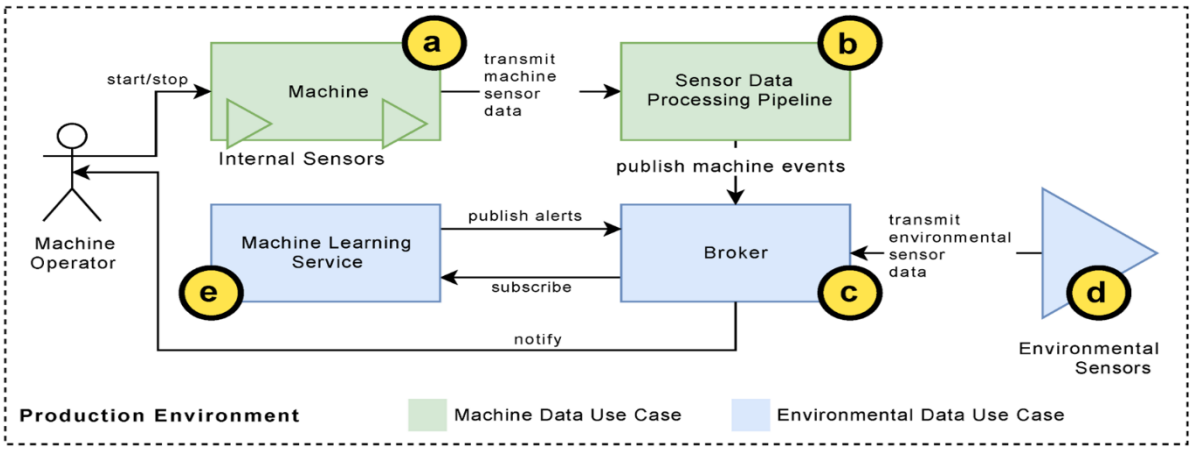


Figure 11: anomaly detection 2

4.4.3. Intrusion Detection Systems

Intrusion Detection Systems (IDS) are essential for maintaining the security and integrity of Controller Area Network (CAN) bus systems. They monitor and analyze network traffic to identify suspicious or malicious activities. Here are three critical components of IDS in the context of the CAN bus:

Real-Time Traffic Analysis

Description: Real-time traffic analysis involves continuous monitoring of live data on the CAN bus to identify abnormal patterns or behaviors that may indicate an intrusion or attack.

How it Works:

- The IDS continuously scans the CAN bus traffic, comparing the observed patterns to a baseline of normal behavior.
- It detects deviations from this baseline, such as unexpected surges in message frequency, unusual message identifiers, or unexpected communication sequences.
- When deviations are detected, the IDS triggers alerts or alarms, prompting further investigation or immediate action.

Benefits:

- Enables rapid detection of anomalous behavior, allowing for timely responses to potential security threats.
- Helps in maintaining the integrity and reliability of the CAN bus network by quickly identifying and addressing suspicious activities.

Payload Analysis

Description: Payload analysis involves inspecting the content of CAN messages, focusing on the actual data (payload) being transmitted rather than just the message structure.

How it Works:

- The IDS examines the payload of CAN messages to identify patterns or signatures associated with known attacks.
- This includes detecting unauthorized commands, malicious data injection, or attempts to exploit vulnerabilities in the communication protocol.
- Advanced payload analysis may utilize pattern recognition, machine learning, and heuristic analysis to identify sophisticated attack signatures.

Benefits:

- Enhances the ability to detect attacks that involve manipulating the content of CAN messages.
- Provides a deeper level of inspection beyond structural aspects, increasing the detection accuracy for content-based attacks.

Hardware Support

Description: Hardware support for intrusion detection involves integrating security features directly into the CAN bus hardware or using dedicated hardware components to enhance IDS capabilities.

How it Works:

- Specialized hardware components can offload specific security tasks from the main control units, such as encryption, authentication, and real-time monitoring.
- These components may include dedicated security modules designed to handle intensive tasks, improving overall performance.
- Hardware support can also include sensors and other devices equipped with built-in security features to detect and mitigate attacks.

Benefits:

- Improves the overall performance and efficiency of intrusion detection by reducing the processing burden on the main control units.
- Provides a more robust defense against attacks that attempt to compromise or bypass software-based security measures.
- Enhances the reliability and effectiveness of IDS by incorporating physical security measures directly into the hardware.

4.4.4. Detecting False Data from Sensors Using Deep Learning

Overview: Detecting false data from sensors, such as cameras in self-driving cars, can be effectively approached using deep learning models specifically designed for anomaly detection.

Steps to Implement Deep Learning for Sensor Anomaly Detection:

1. **Data Collection:**
    - Gather a diverse dataset that includes both normal and anomalous scenarios.

- Simulate attacks on the camera system to introduce manipulated or false data, ensuring the dataset captures a wide range of potential anomalies.

2. **Feature Extraction:**
   - Extract relevant features from the camera data that capture the characteristics of normal sensor data.
   - Features may include color distribution, object shapes, textures, and motion patterns.
3. **Training the Model:**
   - Train a deep learning model using the labeled dataset, focusing on normal camera behavior.
   - Models such as autoencoders or convolutional neural networks (CNNs) can be used to learn to reconstruct normal images and identify anomalies during attacks.
4. **Validation and Tuning:**
   - Validate the model's performance using a separate dataset that includes both normal and attacked scenarios.
   - Fine-tune the model's parameters to improve its ability to distinguish false data from genuine sensor inputs.
5. **Real-Time Monitoring:**
   - Integrate the trained model into the self-driving car's system to monitor camera data in real-time.
   - The model continuously analyzes incoming frames, identifying anomalies and flagging potential false data.

Benefits of Deep Learning for Anomaly Detection:

- Enhances the accuracy and reliability of detecting false data from sensors.
- Provides a scalable solution that can adapt to various types of sensors and data inputs.
- Helps maintain the integrity and safety of systems relying on sensor data, such as autonomous vehicles.

Chapter 5

Related Work

5.1.Related Work

5.1.1. Anomaly Detection Based on Deep Learning Architecture

The trajectory of intelligent vehicular advancements must inexorably gravitate towards unmanned operations integrated with comprehensive information dissemination and network connectivity. The amalgamation of in-vehicle data necessitates a multifaceted convergence facilitated by an array of sensors. However, the integrity of in-vehicle information stands as a formidable impediment to the seamless operation of autonomous vehicles. Hence, the imperative arises for the expeditious establishment of an in-vehicle information anomaly detection infrastructure.

Within scholarly discourse, a pioneering entropy-based approach to attack detection in in-vehicle networks is elucidated. This method leverages entropy as a metric to elucidate the degree of uncertainty within a dataset. Preprocessed CAN bus messages undergo entropy computation and relative distance measurement, subsequently juxtaposed against a baseline sample library established during the calibration phase. Detection of anomalies is effectuated, signified by the activation of alarms, thereby ensuring the unimpeded functionality of the CAN bus network by preempting potential attacks.

A methodical framework is introduced for the detection of anomaly messages within in-vehicle networks, affording the discernment of attacks during vehicular operation while mitigating false positives. Subsequent investigations delineate an actual attack model employing a malevolent smartphone application within the connected car ecosystem, substantiated through empirical experimentation. Concurrently, a security protocol tailored for vehicular environments is devised. Further studies advocate a suite of broadcast authentication protocols predicated on computationally efficient one-way functions, devoid of reliance on disclosure delays inherent in time-synchronized protocols or those reliant on one-way chains.

Additionally, a novel intrusion detection algorithm is posited, engineered to identify malicious CAN messages surreptitiously injected by assailants into the contemporary vehicle CAN bus. Characterized by its parsimonious memory footprint and computational efficiency, this algorithm is particularly suited for contemporary bus Electronic Control Units (ECUs). In light of the burgeoning proliferation of external interfaces within the vehicle control network, such as the On-Board Diagnostic (OBD) port and auxiliary media port, the imminent realization of vehicle-to-vehicle/vehicle-to-infrastructure technology necessitates preemptive measures. A methodology is advanced to detect aberrant flow patterns by scrutinizing the irregular refresh rates of specific commands, thus fortifying the CAN bus against potentially life-threatening attacks.

5.1.2. Anomaly Detection Based on Deep Learning Architecture

The burgeoning realm of machine learning has indeed attracted significant attention, especially regarding its application in anomaly detection to safeguard in-vehicle information. Renowned for its innate capacity to autonomously discern data features, dynamically adjust learning parameters, and exhibit commendable prediction accuracy, deep learning methodologies have found pervasive utility across a spectrum of anomaly detection endeavors.

In scholarly discourse [16], an innovative intrusion detection framework is postulated, leveraging a deep neural network (DNN) to iteratively refine weight matrices and distill feature vectors from original CAN data packets. Through the extraction of real-time data packets, the DNN furnishes probabilities associated with each classification, facilitating the discrimination between normal and adversarial packets. Thus, sensors can promptly discern and preemptively counteract malevolent incursions targeting vehicular systems.

Moreover, within the scholarly corpus [17,18], a DNN architecture is delineated, initially trained on normative data to glean higher-order features. Subsequently, this trained model is harnessed to prognosticate forthcoming data values, enhancing the anticipatory capabilities of anomaly detection systems.

By incorporating deep learning methodologies into anomaly detection systems, a notable advancement is achieved in fortifying the security of in-vehicle information. The utilization of DNNs enables the identification of subtle patterns indicative of potential threats, thereby empowering vehicles to proactively safeguard against malicious attacks.

5.1.3. Triplet Loss Network

The triplet loss function is a pivotal component in deep learning, specifically tailored for training samples exhibiting subtle differences, such as handwriting or facial features. Within the feed data packet, both positive and negative anchor samples are incorporated. The essence of the triplet loss lies in optimizing the similarity calculation among samples by minimizing the distance between anchor samples and their positive counterparts while maximizing the distance from negative samples.

Presently, the triplet loss network has garnered widespread adoption, particularly in domains like face recognition and object detection. For instance, in the scholarly discourse [19], triplet networks are deployed to tackle the challenge of local image descriptor learning. Addressing the intricacies of text-independent speaker verification against short utterances, the literature [20] introduces an end-to-end system that directly maps speech features to compact, fixed-length discrimination embeddings. The

system employs Euclidean distance to gauge similarity between experiments, while an enhanced priori network featuring residual blocks optimizes the triplet loss function to learn feature mapping.

In the context of this paper, an advanced anomaly detection system for intelligent vehicles is presented, amalgamating DNN technology with a triplet loss network. Initially, the system harnesses deep networks to extract data features, yielding a set of vectors. Subsequently, utilizing the triplet loss mechanism, the system computes the similarity between real-time extracted CAN data sequences and a calibrated reference sequence from three data sequences, thereby discerning anomalous data instances.

5.2. Proposed Method

In this section, we outline the Siamese structure employed in our proposed system and provide a detailed description of the deep neural network (DNN) architecture utilized in this study. Furthermore, we elucidate the network training process and the methodology employed for data feature extraction. Finally, we introduce the utilization of a triplet loss network to evaluate the similarity between two randomly extracted data samples and annotated data, respectively.

5.2.1. The Overall Framework

Figure 2 illustrates the comprehensive structure of CAN bus messages anomaly detection adopted in this study. We define the annotated CAN bus messages as an anchor, which undergoes pre-processing offline. The negative and positive instances represent two collected real-time CAN bus messages. Each batch comprises the first three CAN bus messages. Over time, numerous CAN data sequences are collected, with two sequences gathered in each adjacent time forming a batch alongside the annotated sequence. Consequently, multiple batch data sequences are generated. Essentially, the objective of this study is to identify abnormal sequences from these batches.

Similar to a convolutional neural network (CNN), the role of a deep neural network (DNN) is to extract data features. Three data sequences from the same batch are fed into three networks, which maintain structural consistency and share weights with each other. Subsequently, the extracted three sets of data features are represented as three independent feature vectors.

Finally, drawing inspiration from similarity calculation methodologies employed in handwritten image recognition and face recognition, the triplet loss function is utilized to calculate the similarity between random data sequences and the anchor. This function aims to minimize the distance between normal data sequences and the annotated data sequence while simultaneously maximizing the distance between abnormal data sequences and the annotated data sequence.

5.2.2. The Shared-Weight DNN Module

In recent years, deep neural network (DNN) algorithms have garnered significant attention in the realm of machine learning [25-27], contributing to notable improvements in recognition rates across various detection tasks. Within a standard CAN communication system, four types of data frames exist: data frames, remote frames, error frames, and overload frames. While these frames differ in length and function, only data frames facilitate meaningful operations within the CAN communication process.

The syntax of a standard CAN packet, as depicted in Figure 3, comprises several seven-bit fields: the start of the frame, arbitration field, control field, data field, cyclic redundancy check (CRC) field, acknowledgement character (ACK) field, and the end of the frame. The start of the frame denotes the commencement of the data frame and the remote frame, indicated by a single "dominant" bit. The arbitration field encompasses an identifier and RTR (Radio Teletype Receiver) bits, with the standard frame format differing from the extended frame format in the arbitration field structure.

The control field encompasses six bits, including two reserved bits (r0 and r1) and four data length code bits, permitting data lengths ranging from zero to eight bytes. Similarly, the data field transmits a buffer based on the length code, indicating the length of the transmitted or received data, which can range from zero to eight bytes. Each byte comprises eight bits, with the first bit being the most significant bit (MSB).

The CRC code field comprises a CRC field of 15 bits and a CRC boundary character, employed in CRC computation across various domains. The ACK field consists of two recessive bits sent by the sender, wherein all nodes receiving the correct CRC sequence convert the received recessive bits into dominant bits in the reply gap of the sending node.

A standard CAN bus packet encompasses several distinct parts, each serving unique functions, thereby possessing distinct characteristics.

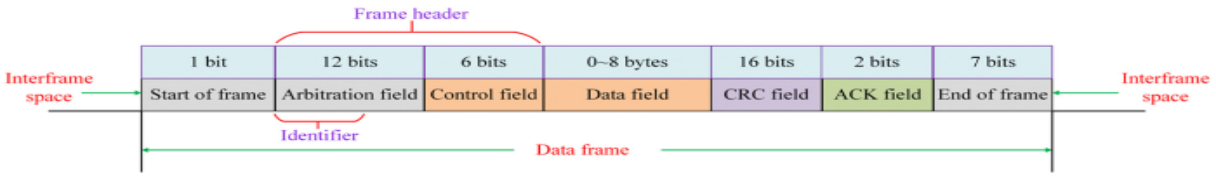


Figure 3. The syntax structure of a standard CAN packet. CRC = cyclic redundancy check, ACK = Acknowledgement Character.

Figure 12: standard CAN packet

Chapter 6

Requirements Specifications

6.1. CAN Bus Architecture

The Controller Area Network (CAN) bus is a foundational component of modern vehicle architecture, facilitating efficient communication between electronic control units (ECUs) distributed throughout the vehicle. This robust communication protocol has become widely adopted across the automotive industry due to its reliability, simplicity, and versatility.

CAN bus architecture is designed around a decentralized network topology, where multiple ECUs communicate with each other via a shared communication medium, typically a twisted-pair cable. This distributed approach allows for seamless integration of various vehicle systems, including engine control units, transmission control units, anti-lock braking systems, airbag control modules, and more.

At the heart of the CAN bus architecture is the concept of message-based communication. ECUs exchange data in the form of messages, which are transmitted over the bus and received by other ECUs as needed. These messages contain information about vehicle parameters, sensor readings, control commands, and diagnostic data.

Key features of CAN bus architecture include:

1. Message-based communication: Data exchange between ECUs occurs through discrete messages, each containing information relevant to specific functions or systems within the vehicle.
2. Broadcast communication: Messages are broadcasted over the CAN bus, allowing all connected ECUs to receive and process relevant data. This enables real-time sharing of information across different subsystems without the need for point-to-point connections.
3. Deterministic communication: CAN bus architecture ensures deterministic communication, meaning that messages are transmitted and received with predictable timing and reliability. This is critical for safety-critical applications where precise synchronization is required.
4. Multi-master architecture: CAN bus supports a multi-master architecture, allowing multiple ECUs to transmit and receive messages independently. This decentralized approach enhances system flexibility and fault tolerance.
5. Error detection and fault tolerance: CAN bus incorporates robust error detection mechanisms, such as cyclic redundancy checks (CRC), to ensure data integrity. Additionally, it features built-in fault tolerance capabilities, enabling the network to continue functioning even in the presence of faulty nodes or communication errors.

Overall, CAN bus architecture provides a standardized and reliable communication framework for modern vehicles, facilitating seamless integration of electronic systems and enabling advanced functionalities such as autonomous driving, telematics, and vehicle diagnostics. Understanding the fundamental principles of CAN bus architecture is essential for designing, implementing, and maintaining efficient and reliable automotive systems.

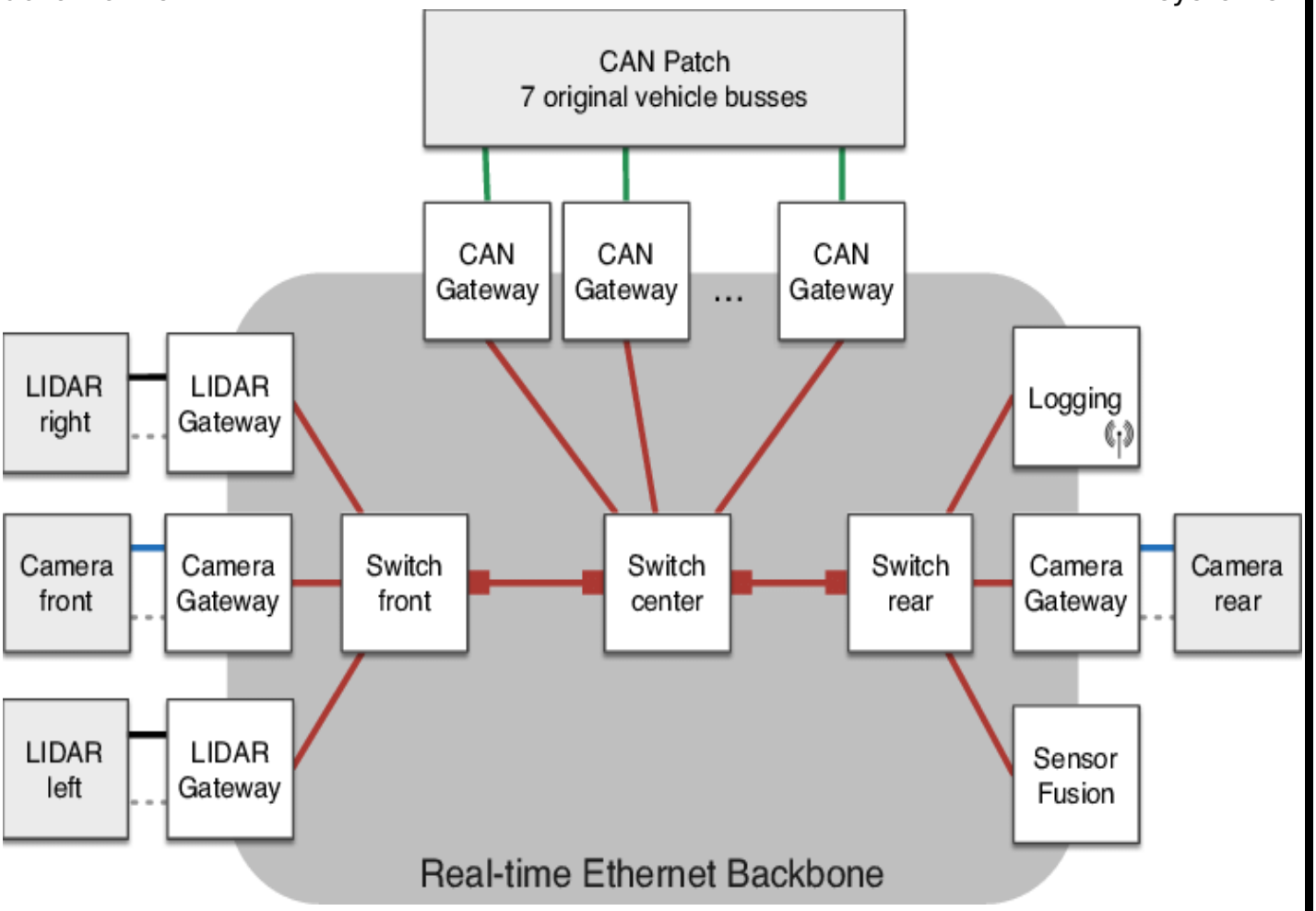


Figure 13:Real-time Ethernet Backbone

6.1.1. CAN Bus Protocol Overview

The CAN bus protocol serves as the backbone of communication within a vehicle's electronic control units (ECUs), dictating how these units exchange data and coordinate operations. Operating at the data link layer of the OSI model, the CAN protocol establishes a reliable, efficient communication framework characterized by its decentralized, message-based architecture.

Key aspects of the CAN bus protocol include:

1. **Message-based communication:** The CAN protocol enables ECUs to communicate through discrete messages, each containing data or control information relevant to specific functions or systems within the vehicle.

2. **Decentralized architecture:** In the CAN protocol, there is no central controller governing communication. Instead, each ECU operates as a peer node on the network, capable of initiating message transmissions independently.
3. **Deterministic arbitration:** CAN protocol utilizes a priority-based arbitration scheme to manage access to the bus. Messages are transmitted based on their priority, with higher-priority messages taking precedence over lower-priority ones. This ensures timely delivery of critical data.
4. **Error detection and handling:** The protocol incorporates robust error detection mechanisms, including cyclic redundancy checks (CRC), to detect and correct data transmission errors. Additionally, error frames are generated and transmitted to alert other ECUs of detected errors.
5. **Remote frame transmission:** CAN protocol supports remote frame transmission, allowing ECUs to request specific data from other nodes on the bus. This feature enhances system flexibility and enables efficient data exchange.
6. **Message filtering:** ECUs can filter incoming messages based on their identifiers, allowing them to selectively process relevant data while ignoring extraneous information. This helps optimize bandwidth utilization and reduce processing overhead.

6.1.2. CAN Bus Physical Layer Requirements

The physical layer of the CAN bus architecture is essential for maintaining reliable communication between electronic control units (ECUs) within a vehicle. Meeting the stringent requirements of the automotive environment necessitates careful consideration of several key aspects:

1. **Cabling and Wiring:** High-quality, shielded twisted-pair cables are essential for withstanding the harsh conditions of the vehicle environment. These cables must minimize signal attenuation and crosstalk while ensuring reliable data transmission over varying distances within the vehicle.
2. **Transceiver Specifications:** CAN bus transceivers must exhibit high noise immunity and short-circuit protection to operate reliably in automotive settings. They should also withstand voltage spikes and transients commonly encountered in vehicle electrical systems. Additionally, transceivers must adhere to strict power consumption and electromagnetic compatibility (EMC) requirements to prevent interference with other vehicle systems.
3. **Termination and Impedance Matching:** Proper termination of the CAN bus is critical for preventing signal reflections and maintaining reliable communication. The physical layer specifications should outline appropriate termination resistors and impedance matching techniques to ensure consistent characteristic impedance throughout the bus network, minimizing signal distortion and ensuring optimal signal integrity.

4. **Connector Design:** Automotive-grade connectors are necessary to withstand the rigors of the vehicle environment, including vibration, temperature extremes, and exposure to moisture and contaminants. These connectors should provide secure, reliable connections to prevent signal loss or intermittent communication issues.
5. **Electromagnetic Compatibility (EMC):** Compliance with electromagnetic compatibility (EMC) standards is essential to prevent the CAN bus from generating excessive electromagnetic interference (EMI) that could disrupt other vehicle systems or electronic devices. Rigorous EMC testing and adherence to regulatory requirements are necessary to ensure seamless integration and operation within the vehicle.

By addressing these physical layer requirements, the CAN bus architecture can reliably support the communication needs of modern vehicles, enabling efficient data exchange and facilitating the implementation of advanced automotive functionalities.

6.1.3. CAN Bus Data Link Layer Requirements

The data link layer of the CAN bus architecture is essential for facilitating reliable and efficient data transfer between electronic control units (ECUs) within a vehicle. To meet the demanding requirements of the automotive environment, the CAN data link layer must adhere to a comprehensive set of specifications:

1. **Message Framing and Prioritization:** Standardized message frame formats define critical information such as message identifiers, data payload, and error detection mechanisms. Priority-based arbitration ensures that high-priority messages take precedence, even in scenarios with simultaneous transmission attempts.
2. **Error Detection and Handling:** Robust error detection mechanisms, including cyclic redundancy checks (CRC) and bit monitoring, enable ECUs to detect and respond to transmission errors quickly. This prevents corrupted data from propagating throughout the network, maintaining data integrity.
3. **Message Filtering and Acceptance:** ECUs can selectively process incoming messages based on their identifiers, reducing computational load and enhancing network efficiency. This selective message acceptance improves overall system performance.
4. **Bus Access Arbitration:** A deterministic, priority-based arbitration scheme manages access to the shared bus when multiple ECUs attempt to transmit simultaneously. This mechanism ensures that higher-priority messages are transmitted without collisions, facilitating timely delivery of critical data.
5. **Remote Frame Transmission:** Support for remote frame transmission allows ECUs to request data from other ECUs without separate data transmissions. This

feature reduces the number of messages required for information exchange, improving network efficiency.

By meeting these requirements, the CAN data link layer enables reliable, deterministic, and efficient communication between vehicle systems and components. This facilitates the integration and coordination of advanced automotive functionalities, enhancing overall vehicle performance and safety.

6.1.4. CAN Bus Application Layer Requirements

The CAN bus application layer is crucial for defining specific data and functionalities exchanged between electronic control units (ECUs) within a vehicle. To ensure seamless integration and interoperability, it must meet several requirements:

1. **Standardized Message Definitions:** Define a standardized set of message IDs and data formats universally understood by all ECUs. This ensures smooth information exchange and coordinated functionality across the vehicle.
2. **Application-Specific Data Structures:** Specify efficient data structures for various automotive applications to convey necessary information while optimizing bus utilization.
3. **Diagnostic and Maintenance Support:** Incorporate diagnostic and maintenance functionality, enabling ECUs to report fault codes, diagnostic data, and sensor information for efficient troubleshooting and maintenance.
4. **Plug-and-Play Compatibility:** Support a plug-and-play architecture, allowing easy addition or replacement of ECUs without extensive reconfiguration, enhancing vehicle adaptability.
5. **Bandwidth and Performance Management:** Optimize bus bandwidth utilization to transmit critical messages with minimal latency, implementing message prioritization and dynamic bandwidth allocation.

Meeting these requirements enables seamless integration and coordination of automotive systems, supporting the development of advanced vehicles with enhanced safety, performance, and user experience. Additionally, robust diagnostic and maintenance capabilities ensure the CAN bus operates efficiently throughout the vehicle's lifecycle, facilitating efficient issue identification, troubleshooting, and maintenance procedures.

6.1.5. CAN Bus Security Requirements

The security of the CAN bus architecture is paramount as vehicles become increasingly connected and technologically advanced. To address this, comprehensive security requirements are necessary:

1. **Access Control and Authentication:** Implement stringent access control mechanisms to prevent unauthorized access to the CAN bus. Secure authentication protocols should verify the identity of ECUs before granting access, preventing malicious message injection or hijacking of legitimate communication.
2. **Encryption and Data Protection:** Mandate the use of robust encryption algorithms to safeguard the confidentiality and integrity of CAN bus communications. Encryption ensures that messages are protected from eavesdropping and tampering, preserving the confidentiality of sensitive vehicle data.
3. **Intrusion Detection and Incident Response:** Incorporate intrusion detection capabilities to monitor the CAN bus for anomalous activity or cyber threats. Clear incident response procedures should outline steps for isolating affected ECUs, initiating corrective actions, and notifying relevant authorities or vehicle owners in the event of a security breach.
4. **Secure Software Development and Updates:** Ensure secure development and deployment of software updates for connected ECUs. Implement secure software update mechanisms, including code signing and version control, to maintain the integrity and authenticity of firmware updates and mitigate the risk of introducing vulnerabilities or malware.

By adhering to these security requirements, vehicle manufacturers can enhance the integrity and resilience of the CAN bus architecture, safeguarding sensitive data and protecting against cyber threats.

6.1.6. CAN Bus Reliability and Fault Tolerance Requirements

Ensuring the reliability and fault tolerance of the CAN bus architecture is essential for the safe and consistent operation of modern vehicles. Here are key requirements to address these aspects:

1. Redundant Bus Design: Specify a redundant bus architecture where critical systems are connected to multiple CAN bus networks. This redundancy ensures that the failure of a single bus segment or component does not result in a complete loss of communication and functionality. Failover mechanisms should seamlessly transition to backup CAN buses to maintain uninterrupted operation.
2. Error Handling and Fault Tolerance: Incorporate robust error detection and handling mechanisms to maintain data integrity despite noise, electromagnetic interference, or component failures. Features like cyclic redundancy checks (CRC), bit monitoring, and error signaling enable quick identification and response to transmission errors, preventing corrupted data propagation.
3. Fault Isolation and Containment: Define strategies for isolating and containing faults within the network to prevent a single point of failure from affecting the

entire vehicle. Smart transceivers, bus guardians, or specialized hardware can detect and isolate faulty segments or ECUs, ensuring continued operation of unaffected parts of the CAN bus network.

4. Diagnostics and Prognostics: Implement advanced diagnostic and prognostic capabilities to continuously monitor the health and performance of the CAN bus network. This includes detecting impending failures, predicting maintenance needs, and proactively notifying vehicle owners or service personnel for timely interventions and preventive maintenance.

By adhering to these reliability and fault tolerance requirements, vehicle manufacturers can enhance the resilience and availability of the CAN bus architecture, ensuring consistent performance and safety across diverse operating conditions.

6.1.7. CAN Bus Integration with Other Vehicle Systems

Ensuring the seamless integration of the CAN bus architecture with other vehicle systems is crucial for modern automobiles. Here are key requirements to address this aspect:

1. **Interfacing with Other Vehicle Networks:** Specify standardized gateways, bridges, or multiprotocol ECUs that can translate and relay messages between the CAN bus and other in-vehicle communication networks, such as FlexRay, LIN, or Ethernet-based systems. These components ensure that critical data and commands can flow freely throughout the vehicle, facilitating coordinated operation of all electronic subsystems.
2. **Accommodating Advanced Driver Assistance Systems (ADAS):** Design the CAN bus to accommodate the integration of ADAS features, which require real-time, high-bandwidth data exchange between sensors, actuators, and control units. Address bandwidth, message prioritization, and deterministic communication requirements to support seamless integration of ADAS technologies, including adaptive cruise control, lane-keeping assist, and collision avoidance systems.
3. **Ensuring Compatibility and Interoperability:** Define compatibility and interoperability requirements to ensure that CAN bus components and protocols adhere to standardized specifications and can seamlessly communicate with components from different manufacturers or suppliers. This promotes flexibility and facilitates the integration of aftermarket or third-party components into the vehicle's CAN bus network.
4. **Supporting Future Expansion and Upgradability:** Plan for future expansion and upgradability by designing the CAN bus architecture to accommodate new technologies and functionalities. This may involve providing sufficient bandwidth, scalability, and flexibility to incorporate emerging vehicle systems and features,

such as autonomous driving capabilities, vehicle-to-everything (V2X) communication, and over-the-air (OTA) software updates.

By meeting these integration requirements, vehicle manufacturers can ensure that the CAN bus architecture serves as an effective communication backbone for the entire vehicle, enabling seamless coordination and interoperability among diverse electronic systems and components.

6.2. CAN Bus Protocol

The Controller Area Network (CAN) bus protocol is a robust, multi-master, message-based communication protocol designed for real-time control applications. It is widely adopted in automotive and industrial systems due to its speed, flexibility, and fault tolerance.

Multi-Master Architecture

The CAN bus operates on a multi-master principle, meaning any node (or Electronic Control Unit, ECU) connected to the bus can initiate communication. This decentralized approach enhances system reliability and flexibility by eliminating the need for a central controller. Each ECU can independently send and receive messages, enabling a high degree of modularity and scalability in system design.

Message-Based Communication

CAN bus uses a message-based communication system, where data is transmitted in small, fixed-format messages known as frames. Each frame contains a unique identifier, indicating the message's priority and type. This identifier allows nodes to filter and process only the relevant messages, optimizing network efficiency.

Deterministic Arbitration Mechanism

The CAN bus employs a deterministic arbitration mechanism to resolve conflicts when multiple nodes attempt to transmit simultaneously. This mechanism uses the message identifier to determine priority: messages with lower identifier values have higher priority. During arbitration, each node transmits its identifier bit by bit, and the node with the highest priority (lowest identifier) continues transmission while others withdraw. This ensures that critical messages are transmitted promptly without collisions or delays.

Efficient Error Handling

The CAN protocol incorporates robust error detection and handling capabilities to maintain communication integrity. Key features include:

- **Cyclic Redundancy Check (CRC):** Each message includes a CRC field for error detection. The receiving node recalculates the CRC and compares it with the transmitted value to verify data integrity.
- **Bit Monitoring:** Nodes continuously monitor the bus state during transmission. If a node detects a discrepancy between the transmitted bit and the actual bus state, it flags an error.
- **Error Signaling:** When an error is detected, nodes transmit an error frame to alert all nodes on the bus. This frame interrupts the current transmission, prompting a retry.

Fault Tolerance

CAN bus is designed to maintain reliable communication even in the presence of hardware faults or electromagnetic interference. The protocol includes mechanisms to isolate faulty nodes and ensure that the rest of the network continues to function. Additionally, the use of shielded twisted-pair cables and stringent transceiver specifications helps mitigate the impact of noise and interference.

Real-Time Control Applications

The combination of a distributed architecture, message-based design, and robust error handling makes CAN bus ideal for real-time control applications. These features enable fast, reliable data exchange between ECUs, which is critical for applications such as engine control, transmission systems, anti-lock braking systems (ABS), and advanced driver assistance systems (ADAS).

In summary, the CAN bus protocol's multi-master, message-based architecture, deterministic arbitration, and efficient error handling make it a powerful and reliable communication system for real-time control applications. Its ability to maintain reliable communication under adverse conditions and its flexibility in handling multiple nodes without a central controller are key factors in its widespread adoption in the automotive and industrial sectors.

6.2.1. Overview of CAN Bus Architecture

The Controller Area Network (CAN) bus architecture is designed to facilitate efficient and reliable communication between various electronic control units (ECUs) within a vehicle or industrial system. Unlike traditional point-to-point wiring, the CAN bus uses a shared two-wire bus system, allowing multiple ECUs to connect and communicate seamlessly.

Multi-Master, Broadcast Communication Model

The CAN bus operates on a multi-master, broadcast communication model. Each ECU can act as both a sender (master) and receiver (slave), enabling any ECU to initiate communication at any time. This decentralized and distributed topology eliminates the need for a central controlling unit, enhancing system robustness and flexibility.

Shared Two-Wire Bus

The shared bus consists of two wires, CAN_H (high) and CAN_L (low), which form a differential pair to transmit data. This differential signaling helps to reduce the impact of electromagnetic interference (EMI), ensuring reliable data transmission even in noisy environments typically found in automotive and industrial settings.

CAN Controllers and Transceivers

Each ECU connected to the CAN bus is equipped with a CAN controller and a CAN transceiver. The CAN controller handles the assembly and disassembly of CAN frames, while the transceiver converts the digital signals from the controller into the differential signals required for transmission over the bus and vice versa.

Producer-Consumer Paradigm

The CAN bus architecture follows a producer-consumer paradigm. When an ECU (the producer) has data to share, it constructs a CAN message frame containing the relevant information and broadcasts it over the bus. All other connected ECUs (the consumers) monitor the bus and receive any frames that match their filtering criteria. This approach enables efficient data exchange, as each ECU only processes the messages pertinent to its functionality.

Message-Based Communication

In CAN bus communication, data is transmitted in structured message frames. Each frame includes an identifier, which indicates the priority of the message, and a data payload, which contains the actual information being transmitted. This message-based approach simplifies network design and allows for easy addition of new ECUs without the need to reconfigure existing point-to-point connections.

Arbitration Mechanism

A key feature of the CAN bus is its robust arbitration mechanism. When multiple ECUs attempt to transmit simultaneously, the CAN protocol uses a bitwise arbitration process based on the message identifiers to resolve conflicts. During arbitration, each ECU transmits its message identifier bit by bit. The bus state reflects the dominant (logical 0)

and recessive (logical 1) states. If a transmitting ECU detects that the bus state does not match the bit it sent (i.e., it sent a recessive bit but the bus state is dominant), it stops transmitting, conceding the bus to the ECU with the higher priority message (lower identifier value). This ensures that the highest priority message is transmitted first without collision, supporting the real-time control requirements.

Scalability and Flexibility

The CAN bus architecture is highly scalable and flexible. New ECUs can be added to the network with minimal effort, as they simply need to connect to the shared bus and start communicating using the predefined message identifiers. This makes the CAN bus an ideal solution for complex systems that require frequent updates or expansions.

Error Detection and Handling

The CAN bus includes robust error detection and handling mechanisms, such as cyclic redundancy checks (CRC), bit monitoring, and error signaling. These features ensure data integrity and enable the network to quickly identify and respond to transmission errors, maintaining reliable communication.

In summary, the CAN bus architecture's multi-master, broadcast communication model, combined with its robust arbitration mechanism and message-based communication, provides a flexible, scalable, and reliable solution for real-time control applications in automotive and industrial systems.

6.2.2. CAN Bus Message Structure

The CAN bus protocol defines a highly structured and standardized message format, known as a CAN frame, to facilitate reliable data exchange between electronic control units (ECUs) connected to the network. Each CAN frame is composed of several distinct fields, each serving a specific purpose to ensure the integrity and efficiency of the communication process. Here's a detailed breakdown of the CAN message structure:

1. Start of Frame (SOF)

- **Purpose:** Indicates the beginning of a new CAN frame.
- **Description:** Consists of a single dominant bit (0). This bit allows all nodes to synchronize with the start of the frame.

2. Arbitration Field

- **Purpose:** Determines the priority of the message.

- **Description:** Contains the message identifier and the Remote Transmission Request (RTR) bit.
    - **Identifier:** In Standard CAN (CAN 2.0A), the identifier is 11 bits long. In Extended CAN (CAN 2.0B), it is 29 bits long.
    - **RTR Bit:** Indicates whether the frame is a data frame (dominant bit) or a remote frame (recessive bit).

3. Control Field

- **Purpose:** Provides metadata about the frame.
- **Description:** Contains the Data Length Code (DLC) and, in the case of extended frames, additional bits.
    - **DLC:** 4 bits indicating the number of bytes in the data field (0-8 bytes).
    - **Reserved Bits:** Two reserved bits (r0 and r1) in extended frames.

4. Data Field

- **Purpose:** Carries the actual payload.
- **Description:** Can be 0 to 8 bytes long. Each byte consists of 8 bits. This field contains the information being transmitted, such as sensor readings or control commands.

5. Cyclic Redundancy Check (CRC) Field

- **Purpose:** Ensures the integrity of the transmitted message.
- **Description:** Consists of a 15-bit CRC sequence followed by a 1-bit CRC delimiter. The CRC is computed from the preceding bits of the frame and allows the receiver to detect errors in the transmission.

6. Acknowledgment (ACK) Field

- **Purpose:** Confirms the successful reception of the frame.
- **Description:** Includes two bits:
    - **ACK Slot:** A single recessive bit that is overwritten as a dominant bit by any node that successfully receives the frame.
    - **ACK Delimiter:** A single recessive bit that separates the ACK Slot from the End of Frame field.

7. End of Frame (EOF)

- **Purpose:** Marks the end of the CAN frame.
- **Description:** Consists of 7 recessive bits. This field allows for proper frame separation and ensures that the bus remains idle before the next frame begins.

8. Interframe Space

- **Purpose:** Ensures separation between consecutive CAN frames.
- **Description:** Consists of at least 3 recessive bits. It allows nodes to process the last frame and prepare for the next one.

Example of a Standard CAN Frame

1. **Start of Frame (SOF):** 1 bit (dominant)
2. **Arbitration Field:**
    - **Identifier:** 11 bits
    - **RTR Bit:** 1 bit
3. **Control Field:**
    - **DLC:** 4 bits
4. **Data Field:** 0 to 64 bits (0 to 8 bytes)
5. **CRC Field:**
    - **CRC Sequence:** 15 bits
    - **CRC Delimiter:** 1 bit (recessive)
6. **ACK Field:**
    - **ACK Slot:** 1 bit
    - **ACK Delimiter:** 1 bit (recessive)
7. **End of Frame (EOF):** 7 bits (recessive)

6.2.3. CAN Bus Frame Format

frame format to ensure efficient, reliable, and error-free communication between electronic control units (ECUs) in a network. The CAN frame format is composed of several distinct fields, each serving a specific purpose. The two primary types of CAN frames are the Standard Frame (CAN 2.0A) and the Extended Frame (CAN 2.0B), which differ mainly in the length of the identifier field. Below is a detailed explanation of the CAN frame format.

Standard CAN Frame Format (CAN 2.0A)

1. **Start of Frame (SOF)**
    - **Length:** 1 bit
    - **Purpose:** Indicates the start of a new frame with a dominant bit (0).
2. **Arbitration Field**
    - **Length:** 12 bits (11-bit identifier + RTR bit)
    - **Identifier:** 11 bits
        - Unique identifier for the message, used for message prioritization.
    - **RTR Bit:** 1 bit

        - Remote Transmission Request bit; dominant (0) for data frames, recessive (1) for remote frames.
3. **Control Field**
    - **Length:** 6 bits
    - **IDE Bit:** 1 bit
        - Identifier Extension bit; dominant (0) for standard frames.
    - **r0 Bit:** 1 bit
        - Reserved bit, set to dominant (0).
    - **DLC (Data Length Code):** 4 bits
        - Indicates the number of bytes in the data field (0-8 bytes).
4. **Data Field**
    - **Length:** 0 to 64 bits (0 to 8 bytes)
    - **Purpose:** Contains the actual data being transmitted.
5. **CRC Field**
    - **Length:** 15 bits (CRC sequence) + 1 bit (CRC delimiter)
    - **Purpose:** Error-checking field. The CRC sequence is a checksum calculated from the preceding bits, allowing receivers to detect transmission errors.
6. **ACK Field**
    - **Length:** 2 bits
    - **ACK Slot:** 1 bit
        - Acknowledgment slot; recessive (1) but overwritten by a dominant bit (0) by any node that correctly receives the frame.
    - **ACK Delimiter:** 1 bit
        - Always recessive (1).
7. **End of Frame (EOF)**
    - **Length:** 7 bits
    - **Purpose:** Marks the end of the frame with seven recessive bits (1).
8. **Interframe Space**
    - **Length:** Minimum of 3 bits
    - **Purpose:** Ensures separation between consecutive frames.

Extended CAN Frame Format (CAN 2.0B)

1. **Start of Frame (SOF)**
    - **Length:** 1 bit
2. **Arbitration Field**
    - **Length:** 32 bits (29-bit identifier + SRR + IDE + RTR bit)
    - **Identifier:** 29 bits
        - Extended identifier for messages.
    - **SRR Bit:** 1 bit
        - Substitute Remote Request; dominant (0) for compatibility with standard frames.

    - **IDE Bit:** 1 bit
        - Identifier Extension; recessive (1) for extended frames.
    - **RTR Bit:** 1 bit
        - Same as in standard frames.
3. **Control Field**
    - **Length:** 6 bits
    - **r0 Bit:** 1 bit
    - **r1 Bit:** 1 bit (extended frame)
    - **DLC:** 4 bits
4. **Data Field**
    - **Length:** 0 to 64 bits
5. **CRC Field**
    - **Length:** 15 bits (CRC sequence) + 1 bit (CRC delimiter)
6. **ACK Field**
    - **Length:** 2 bits
7. **End of Frame (EOF)**
    - **Length:** 7 bits
8. **Interframe Space**
    - **Length:** Minimum of 3 bits

Key Components of the CAN Frame

Arbitration ID

- **Purpose:** Determines the priority of the message. Lower numerical values have higher priority.
- **Length:** 11 bits for standard frames, 29 bits for extended frames.

Control Field

- **DLC (Data Length Code):** Indicates the size of the data field.
- **IDE Bit:** Differentiates between standard and extended frames.
- **RTR Bit:** Indicates if the frame is a data frame or a remote frame.

Error Checking

- **CRC Field:** Ensures data integrity by allowing receivers to detect errors.
- **ACK Field:** Confirms receipt of the message.

6.2.4. CAN Bus Arbitration and Prioritization

Bitwise Arbitration

The CAN bus protocol employs a bitwise arbitration mechanism to manage access to the shared communication bus when multiple electronic control units (ECUs) attempt to transmit simultaneously. This process relies on the dominant-recessive bit signaling inherent in CAN communication. Here's how it works:

- **Dominant vs. Recessive Bits:** In CAN, a dominant bit (logical 0) overrides a recessive bit (logical 1) on the bus. This physical layer property is utilized to implement the arbitration process.
- **Arbitration Process:** Each ECU begins transmitting its message starting with the most significant bit (MSB) of the arbitration ID. As bits are transmitted, ECUs monitor the bus to see if the bit they sent matches the state of the bus.
    - If an ECU sends a recessive bit but detects a dominant bit, it recognizes that another ECU with a higher priority message (lower numerical arbitration ID) is also trying to transmit. This ECU will then cease transmission and wait for the bus to become free.
    - The ECU with the lowest numerical arbitration ID will have its bits match the bus state throughout the arbitration process and will thus win the arbitration and continue transmitting its entire message.

Priority-based Access

The arbitration ID serves a dual purpose:

- **Unique Identification:** Each message has a unique identifier.
- **Priority Determination:** The numerical value of the arbitration ID determines message priority. Lower values indicate higher priority.
    - **High Priority Messages:** Emergency notifications, critical sensor data, and time-sensitive control commands are typically assigned lower numerical IDs to ensure they are transmitted promptly.
    - **Low Priority Messages:** Less urgent data, such as routine status updates or diagnostics, are assigned higher numerical IDs and will wait if higher priority messages are being transmitted.

This priority-based access ensures that the most critical data is transmitted first, providing the deterministic behavior required for real-time control applications.

Dynamic Prioritization

In some systems, the priority of messages may need to change dynamically based on current operational conditions. CAN supports this flexibility through remote frame transmissions:

- **Remote Frames:** An ECU can request the latest data from another node by sending a remote frame, which contains only the arbitration ID and no data field. The node receiving the remote frame responds with the requested data.
- **Adaptive Prioritization:** This mechanism allows the system to dynamically adjust the priorities of messages. For example, if a particular sensor value becomes critical due to changing conditions, its priority can be increased by frequently requesting its data using remote frames.

This dynamic prioritization capability enhances the responsiveness and adaptability of CAN-based systems, allowing real-time adjustments to communication priorities as needed.

Configuring Arbitration IDs

The assignment of arbitration IDs is a crucial aspect of designing a CAN-based network:

- **Design Phase:** During the design phase, arbitration IDs are carefully assigned to ensure that the most critical messages have the highest priority (i.e., the lowest numerical IDs).
- **System Scalability:** The arbitration ID scheme should be scalable to accommodate future additions of ECUs or changes in system requirements.
- **Consistency and Coordination:** Proper planning and coordination are essential to maintain a consistent and logical prioritization scheme, especially in large and complex networks.
- **Dynamic Allocation:** In some advanced systems, arbitration IDs can be dynamically allocated or adjusted based on current system states or operational needs. This requires sophisticated management to ensure that real-time requirements are still met.

By meticulously planning and configuring arbitration IDs, CAN networks can achieve reliable, deterministic communication that meets the stringent requirements of automotive, industrial, and embedded control systems.

6.2.5. CAN Bus Error Handling and Diagnostics

The CAN bus protocol includes sophisticated error handling and diagnostic mechanisms to ensure reliable and fault-tolerant communication among the connected electronic control units (ECUs). These features are crucial for maintaining the integrity and responsiveness of CAN-based systems, especially in safety-critical environments such as automotive and industrial applications.

Error Detection Mechanisms

The CAN protocol defines several types of errors that can occur during communication, each with specific detection methods:

- **Bit Errors:** Occur when a bit value is read differently than it was transmitted. Each node monitors the bus while transmitting and expects the bit value to match what it sent. If it detects a discrepancy, a bit error is flagged.
- **Stuff Errors:** CAN uses bit stuffing to maintain synchronization. After five consecutive bits of the same polarity, a bit of the opposite polarity is inserted. A stuff error occurs if six consecutive bits of the same polarity are detected.
- **CRC Errors:** The cyclic redundancy check (CRC) field helps verify data integrity. If the calculated CRC does not match the transmitted CRC, a CRC error is flagged.
- **Form Errors:** These occur if fixed-form fields (like the ACK delimiter or EOF) contain illegal values.
- **Acknowledgment Errors:** If a message is transmitted but no node acknowledges it (by sending a dominant bit during the ACK slot), an acknowledgment error occurs.

Error Frames and Error Handling

When a node detects an error, it transmits an error frame to alert other nodes of the issue:

- **Active Error Frames:** Sent by nodes in an error-active state. These frames consist of six dominant bits, followed by eight recessive bits, indicating an error.
- **Passive Error Frames:** Sent by nodes in an error-passive state, using six recessive bits, followed by eight recessive bits, indicating a less severe error state.

Nodes increment internal error counters when they detect or contribute to errors. Based on these counters, nodes transition between three states:

1. **Error Active:** Normal operation. The node can transmit and receive messages and can send active error frames.

2. **Error Passive:** The node can still participate in communication but with a reduced role. It sends passive error frames instead of active ones.
3. **Bus Off:** The node is isolated from the bus after too many errors, preventing it from disrupting the network.

Automatic Retransmission

If an error is detected during transmission, the CAN controller will automatically attempt to retransmit the message. This continues up to a predefined number of retries. If the error persists, the node may enter the error-passive or bus-off state to mitigate its impact on the network.

Diagnostic Capabilities

CAN-based systems incorporate advanced diagnostic features to facilitate error identification and resolution:

- **Self-Testing:** Many ECUs perform self-tests to monitor their own health and operation.
- **Error Logging:** ECUs often log detailed error information, including timestamps, error types, and operational context.
- **Diagnostic Tools:** Technicians use specialized CAN diagnostic tools to access error logs and analyze network performance. These tools can decode error frames, interpret error counters, and provide insights into network health.

Importance of Robust Error Handling

The robust error-handling and diagnostic features of the CAN protocol are essential for several reasons:

- **Reliability:** Ensures that messages are accurately transmitted and received, maintaining system reliability.
- **Safety:** In safety-critical applications, such as automotive systems, robust error handling prevents erroneous data from causing unsafe conditions.
- **Maintainability:** Advanced diagnostics enable rapid identification and correction of faults, reducing downtime and maintenance costs.
- **Scalability:** Effective error management supports the addition of new nodes and features without compromising network integrity.

By employing these error detection, handling, and diagnostic strategies, the CAN protocol ensures resilient and reliable communication across a wide range of applications, from automotive systems to industrial automation and beyond.

6.2.6. Integrating CAN Bus into Requirements Specifications

Integrating CAN bus requirements into the overall system specifications is crucial for developing reliable, efficient, and safe CAN-based systems. This process involves several steps to ensure that all necessary aspects of the CAN bus are thoroughly addressed and documented. Here's a comprehensive approach to integrating CAN bus requirements into system specifications:

1. Functional Requirements

ECU Identification and Roles:

- Define the number and types of ECUs that will be connected to the CAN bus.
- Document the specific roles and functions of each ECU, including sensors, actuators, controllers, and gateways.

Data Exchange Specifications:

- List the types of data each ECU will send and receive.
- Specify the data formats, including the size and structure of each data message.
- Detail any data conversion or translation requirements for interoperability with other communication protocols (e.g., FlexRay, LIN, Ethernet).

Real-Time Requirements:

- Establish the real-time responsiveness required for time-critical functions.
- Set deadlines for message delivery to ensure timely execution of control functions.

2. Technical Specifications

Arbitration and Prioritization:

- Define the arbitration ID scheme, including the priority levels for different messages.
- Document how priorities will be assigned and managed to ensure critical data is transmitted promptly.

Bus Utilization and Performance:

- Set thresholds for bus utilization to avoid congestion and ensure smooth communication.
- Specify expected message rates and bus load under different operating conditions.

Fault Tolerance and Error Handling:

- Outline the mechanisms for error detection, reporting, and handling.
- Define the actions to be taken when errors occur, including retransmission attempts and error state transitions (e.g., error-active, error-passive, bus-off).
- Include requirements for diagnostic capabilities, such as error logging and self-testing routines.

3. Deployment and Integration Considerations

Physical Layer Specifications:

- Describe the physical layout of the CAN bus wiring, including length, topology, and shielding.
- Document power supply and grounding requirements to ensure stable and reliable operation.

Electromagnetic Compatibility (EMC):

- Set EMC requirements to minimize interference and ensure compliance with regulatory standards.
- Specify shielding and filtering measures to protect against electromagnetic interference (EMI).

Installation and Maintenance:

- Include guidelines for the installation and maintenance of the CAN bus network.
- Provide requirements for connectors, terminations, and other physical components.

4. Collaboration and Documentation

Cross-Functional Collaboration:

- Facilitate collaboration between electrical engineers, software developers, system architects, and other stakeholders.
- Hold regular meetings and reviews to ensure all requirements are understood and agreed upon.

Requirements Documentation:

- Create detailed documentation of all CAN bus requirements, including functional, technical, deployment, and integration aspects.

- Use clear and precise language to avoid ambiguities and ensure all team members have a common understanding.

Validation and Verification:

- Establish a validation and verification plan to test the CAN bus network against the specified requirements.
- Include both simulation and real-world testing to ensure the network performs as expected under various conditions.

Example: Integrating CAN Bus Requirements in an Automotive System

Functional Requirements:

- **ECUs:** Engine Control Unit (ECU), Transmission Control Unit (TCU), Anti-lock Braking System (ABS), Infotainment System.
- **Data Exchange:** Engine speed and torque data from ECU to TCU, brake status from ABS to ECU, multimedia commands from Infotainment System.
- **Real-Time Requirements:** Engine speed data must be updated every 10 ms, brake status within 5 ms of change.

Technical Specifications:

- **Arbitration:** ECU messages have highest priority (ID 0x100), TCU next (ID 0x200), ABS (ID 0x300), Infotainment lowest (ID 0x400).
- **Bus Utilization:** Maximum bus load should not exceed 70%.
- **Error Handling:** Use CRC for error detection, retransmit messages up to 3 times, log errors with timestamps.

Deployment and Integration:

- **Physical Layout:** Star topology with maximum bus length of 5 meters, shielded cables.
- **EMC Requirements:** Use twisted pair cables, include EMI filters at critical points.
- **Maintenance:** Regular inspection of connectors every 6 months, immediate replacement of damaged cables.By integrating these detailed CAN bus requirements into the overall system specifications, the development team can ensure the CAN network is robust, efficient, and capable of meeting the system's functional and performance needs.

6.2.7. CAN Bus Configuration and Parameterization

Configuring and parameterizing a CAN bus network is critical for achieving reliable and efficient communication between electronic control units (ECUs) within a system. This process involves several key steps, each focused on ensuring that the network operates seamlessly and meets the specific requirements of the application.

Network Architecture

Topology and Layout:

- **Topology:** Define the network topology, which could be linear (daisy-chain), star, or hybrid, based on the application's requirements.
- **Node Count and Types:** Determine the number of nodes (ECUs) and their respective types, such as sensors, actuators, gateways, and controllers.
- **Physical Layout:** Plan the physical layout, including the routing of CAN bus wires. Ensure optimal placement to minimize signal degradation and EMI (Electromagnetic Interference).
- **Redundancy:** Incorporate redundancy where necessary, such as dual CAN buses for critical systems to ensure fault tolerance.

Arbitration ID Scheme

Unique IDs:

- **ID Assignment:** Assign unique arbitration IDs to each message. Use a structured approach to categorize messages (e.g., critical control messages, sensor data, diagnostics).
- **Priority Management:** Prioritize messages by assigning lower numerical values to higher priority messages, ensuring that critical data is transmitted first during bus arbitration.
- **Future Proofing:** Plan for future expansions by reserving ID ranges for potential new ECUs or messages.

Bit Timing and Baud Rate

Bit Timing Configuration:

- **Sample Point:** Set the sample point to an optimal position within the bit period to maximize data integrity.
- **Synchronization Jump Width (SJW):** Configure the SJW to accommodate clock discrepancies between nodes, allowing the network to remain synchronized.

- **Phase Segments:** Adjust the length of phase segments (Phase Segment 1 and Phase Segment 2) to ensure reliable communication, especially over longer distances or in environments with high EMI.

Baud Rate:

- **Max Baud Rate:** Determine the maximum baud rate supported by the network, balancing the need for high-speed communication with the physical limitations of the bus (e.g., cable length, noise).
- **Common Baud Rates:** Typical baud rates range from 125 kbps to 1 Mbps. For applications requiring higher data rates, CAN FD (Flexible Data-rate) can be considered.

Error Handling and Diagnostics

Error Handling:

- **Error Detection:** Configure error detection mechanisms, including cyclic redundancy check (CRC), bit monitoring, and acknowledgment checks.
- **Retransmission Attempts:** Set limits on the number of retransmission attempts for error recovery, ensuring messages are resent a reasonable number of times before the node enters an error state.
- **Error States:** Define thresholds and conditions for nodes to transition into error-active, error-passive, and bus-off states to prevent a faulty node from disrupting the entire network.

Diagnostics:

- **Error Logging:** Implement logging mechanisms to capture error information, including error frames, error counters, and timestamps.
- **Diagnostic Tools:** Use diagnostic tools and software to monitor and analyze CAN bus traffic, enabling proactive maintenance and troubleshooting.
- **Self-Testing:** Incorporate self-testing routines within ECUs to regularly check for proper operation and identify potential issues early.

Example Configuration for an Automotive CAN Bus Network

Network Architecture:

- **Topology:** Linear topology with redundancy for critical systems.
- **Nodes:** 10 ECUs, including Engine Control Unit (ECU), Transmission Control Unit (TCU), Anti-lock Braking System (ABS), Infotainment System, and various sensors and actuators.

- **Layout:** Shielded twisted pair cables with star points at critical junctions to minimize EMI.

Arbitration ID Scheme:

- **ECU Messages:** Assign lower IDs (0x100 to 0x1FF) to high-priority control messages (e.g., ECU to TCU commands), and higher IDs (0x200 to 0x2FF) to sensor data and diagnostics.
- **Future Expansion:** Reserve ID range 0x300 to 0x3FF for future expansions.

Bit Timing and Baud Rate:

- **Sample Point:** Set sample point at 80% of the bit period.
- **SJW:** Configure SJW to 2 time quanta.
- **Phase Segments:** Set Phase Segment 1 to 7 time quanta and Phase Segment 2 to 2 time quanta.
- **Baud Rate:** Operate at 500 kbps to balance speed and reliability.

Error Handling and Diagnostics:

- **Error Detection:** Enable CRC, bit monitoring, and acknowledgment checks.
- **Retransmission:** Allow up to 3 retransmission attempts.
- **Error States:** Transition to error-passive state after 128 errors, bus-off state after 256 errors.
- **Logging:** Implement onboard error logging with timestamp and error code details.
- **Diagnostics:** Utilize diagnostic tools for real-time monitoring and periodic maintenance checks.

By following these steps and considerations, engineers can ensure that the CAN bus network is configured optimally to meet the specific needs of their application, whether in automotive, industrial automation, or other embedded system contexts.

6.2.8. CAN Bus Testing and Validation

Conformance Testing

Purpose: Conformance testing ensures that the CAN bus implementation adheres to the ISO 11898 standard. This process verifies that each electronic control unit (ECU) on the network correctly formats, transmits, and interprets CAN frames according to the protocol specifications.

Steps:

1. **Frame Format Verification:** Ensure all CAN frames (data, remote, error, and overload frames) adhere to the standard format.
2. **Message Transmission:** Check that ECUs transmit messages correctly, including arbitration ID, control field, data length, and CRC.
3. **Message Reception:** Verify that ECUs correctly receive and interpret messages, acknowledging them appropriately.
4. **Protocol Compliance Tools:** Use specialized tools and software to automate the conformance testing process, generating detailed reports on compliance status.

Benefits:

- Identifies non-compliant behavior early in development.
- Reduces risk of interoperability issues.
- Ensures consistent and reliable communication across the CAN network.

Performance Validation

Purpose: Performance validation assesses the real-time behavior of the CAN bus network under various conditions, ensuring it meets the required specifications for determinism, throughput, and fault tolerance.

Steps:

1. **Latency Measurement:** Measure the time taken for messages to travel from the sender to the receiver across the network.
2. **Bus Utilization Analysis:** Monitor the network to determine the percentage of time the bus is actively transmitting messages versus being idle.
3. **Load Testing:** Simulate high traffic conditions to evaluate how the network handles increased message loads and to identify potential bottlenecks.
4. **Stress Testing:** Expose the network to extreme conditions (e.g., maximum load, varying environmental conditions) to ensure reliable performance.

5. **Simulation Tools:** Utilize CAN bus analyzers and simulators to create controlled test environments and capture performance metrics.

Benefits:

- Ensures the network can handle real-time control requirements.
- Identifies performance bottlenecks and timing issues.
- Optimizes CAN bus configuration for mission-critical applications.

Fault Injection and Error Handling

Purpose: Fault injection testing evaluates the robustness and resilience of the CAN bus network by simulating various error conditions and fault scenarios. This process ensures that the network can detect, report, and recover from errors effectively.

Steps:

1. **Bit Error Injection:** Introduce single or multiple bit errors into the CAN frames to test the error detection and correction mechanisms.
2. **Frame Error Injection:** Simulate corrupted frames to observe how the network handles frame-level errors.
3. **Interference Simulation:** Introduce electromagnetic interference (EMI) to test the network's ability to maintain communication integrity in noisy environments.
4. **Error Handling Validation:** Verify that ECUs correctly transition between error-active, error-passive, and bus-off states as per the protocol.
5. **Fault Recovery Testing:** Ensure that the network can recover from errors, including retransmission of messages and resynchronization of nodes.

Benefits:

- Validates the CAN bus network's fault tolerance.
- Ensures reliable communication even under adverse conditions.
- Enhances overall system reliability and robustness.

6.3. CAN Bus Physical Layer Requirements

The physical layer of a Controller Area Network (CAN) bus plays a crucial role in facilitating reliable and robust communication between electronic control units (ECUs) within a vehicle or industrial system. Meeting the stringent requirements of the CAN bus physical layer is essential to ensure the integrity of data transmission, even in challenging environmental conditions. Below are the key specifications and considerations for the CAN bus physical layer:

Cabling and Wiring

- **High-Quality Shielded Twisted-Pair Cables:** Utilize cables capable of withstanding harsh environmental conditions, including temperature extremes, vibration, and electromagnetic interference (EMI).
- **Low Signal Attenuation:** The cable design should minimize signal attenuation to maintain signal integrity over long distances within the vehicle or industrial environment.
- **Crosstalk Reduction:** Implement shielding and twisted-pair configuration to minimize crosstalk between adjacent wires, ensuring reliable data transmission.

Transceiver Specifications

- **High Noise Immunity:** Transceivers must have robust noise immunity to tolerate interference from external sources and other electrical systems within the vehicle or industrial setting.
- **Short-Circuit Protection:** Incorporate features to protect against short circuits, ensuring the integrity of the CAN bus network even in the event of electrical faults.
- **Voltage Spike and Transient Protection:** Transceivers should withstand voltage spikes and transients commonly encountered in vehicle electrical systems without compromising performance or reliability.
- **Low Power Consumption:** Adhere to strict power consumption requirements to minimize energy usage and heat generation, especially in battery-powered or energy-efficient systems.
- **Electromagnetic Compatibility (EMC):** Ensure transceivers comply with EMC standards to prevent interference with other electronic devices and systems.

Termination and Impedance Matching

- **Proper Termination:** Implement termination resistors at both ends of the CAN bus line to minimize signal reflections and ensure signal integrity.
- **Impedance Matching:** Maintain consistent characteristic impedance throughout the bus network to minimize signal distortion and reflections, optimizing signal transmission.

Connector Design

- **Robust Automotive-Grade Connectors:** Utilize connectors specifically designed for automotive or industrial applications, capable of withstanding environmental factors such as vibration, temperature extremes, moisture, and contaminants.
- **Secure and Reliable Connections:** Connectors should provide secure and reliable connections to prevent signal loss or intermittent communication issues, even under harsh operating conditions.

Electromagnetic Compatibility (EMC)

- **Compliance with EMC Guidelines:** Design the CAN bus physical layer to operate within strict electromagnetic compatibility (EMC) guidelines to prevent excessive electromagnetic interference (EMI) that could disrupt other vehicle systems or electronic devices.
- **Rigorous EMC Testing:** Conduct comprehensive EMC testing and compliance verification to ensure adherence to regulatory requirements and standards.

6.3.1. Electrical Characteristics

The CAN bus physical layer must meet stringent electrical requirements to ensure reliable data transmission and fault tolerance within the network. Here are the key electrical specifications:

Voltage Levels:

- **Dominant State (Logical 0):** Typically corresponds to a voltage level between 0V and 3.5V.
- **Recessive State (Logical 1):** Typically corresponds to a voltage level between 2.0V and 5.0V.

Signal Impedance:

- **Characteristic Impedance:** The signal impedance should match the characteristic impedance of the CAN bus cabling, which is typically 120Ω. This impedance matching minimizes signal reflections and ensures proper signal integrity.

Common-Mode Voltage Range:

- **Up to 30V:** The CAN bus lines must be able to handle common-mode voltages up to 30V to accommodate potential differences between connected devices and to provide robustness against electrical noise and interference.

Fault Tolerance:

- **Short Circuit Protection:** The transceiver should be designed to withstand short circuits on the bus lines without being damaged. It should have built-in protection mechanisms to limit the current in case of a short circuit.
- **Open Circuit Detection:** The transceiver should be capable of detecting open circuit faults on the bus lines and report them to the connected CAN controller.
- **Bus-to-Ground and Bus-to-Supply Fault Handling:** The transceiver should be able to handle faults where the bus lines are shorted to ground or the supply voltage, ensuring that such faults do not cause damage to the transceiver or other components in the network.

6.3.2. Bit Timing and Synchronization

Achieving precise bit timing and synchronization is critical for the reliable operation of a CAN bus network. Here's an overview of the key aspects related to bit timing and synchronization:

Bit Timing Configuration:

- **Baud Rate:** The baud rate determines the rate at which bits are transmitted on the bus, typically expressed in bits per second (bps).
- **Time Quanta:** Each bit period is divided into time quanta, with the number of quanta per bit configurable to achieve the desired baud rate.
- **Sample Point:** The sample point is the time within each bit period at which the receiver samples the bus to determine the transmitted bit's value. It's crucial to set the sample point correctly to ensure accurate data reception.

Synchronization Mechanisms:

- **Synchronization Segment:** The synchronization segment at the beginning of each bit allows nodes to synchronize their internal clocks by aligning with the edges of the transmitted signal.
- **Resynchronization Jump Width:** Nodes use resynchronization jump widths to adjust their bit timing dynamically, compensating for clock differences between nodes and ensuring synchronization is maintained.
- **Propagation Delay Compensation:** Propagation delay compensation accounts for the time it takes for the signal to propagate across the bus. It adjusts the sample point to ensure that the receiver samples the bus at the correct time, even with varying cable lengths and node distances.

Importance:

- **Reliable Data Transfer:** Precise bit timing and synchronization are essential for accurate data transmission and reception, minimizing errors and ensuring reliable communication between nodes.
- **Fault Tolerance:** Proper synchronization mechanisms help maintain robust timing, even in the presence of variations in cable lengths, node counts, and environmental conditions, enhancing the network's fault tolerance and resilience.

6.3.3. Transceiver Specifications

The transceiver is a crucial component in the CAN bus physical layer, responsible for driving signals onto the bus and interfacing with the CAN controller. Here are the key specifications and considerations for CAN transceivers:

Electrical Characteristics:

- **Voltage Levels:** Transceivers must support the voltage levels specified by the CAN standard, typically ranging from 0 to 5 volts.
- **Common-Mode Range:** They should tolerate common-mode voltages up to 30V to accommodate potential differences between connected devices.
- **Impedance Matching:** Transceivers should have a characteristic impedance of 120Ω to match the twisted-pair cabling used in CAN bus wiring.

Protection Circuitry:

- **Fault Tolerance:** Transceivers must withstand common fault conditions, such as short circuits, over-voltage, and electrostatic discharges, without sustaining damage.
- **ESD Protection:** Electrostatic discharge (ESD) protection circuits are essential to prevent damage from static electricity buildup during handling and operation.

Advanced Features:

- **Bus Fault Detection:** Transceivers may include features to detect bus faults, such as short circuits or open circuits, and report them to the CAN controller for diagnostics.
- **Error Counters:** Transmit and receive error counters help monitor the health of the bus and identify potential issues early.
- **Bus Load Monitoring:** Some transceivers offer the ability to monitor bus load, providing insights into network congestion and performance.

- **Bus-Off State Management:** Transceivers can manage the bus-off state, where a node is temporarily unable to transmit messages due to excessive errors, to ensure proper network operation and recovery.

Integrated Intelligence:

- **Enhanced Diagnostics:** By integrating advanced diagnostic features, transceivers can provide valuable feedback to the CAN controller, enabling proactive fault detection, diagnostics, and maintenance.
- **High Availability:** Intelligent transceivers contribute to the high availability of CAN bus systems by facilitating rapid fault detection and recovery, minimizing downtime and ensuring reliable operation

6.3.4. Cabling and Connectors

The physical layer of a CAN bus relies on robust cabling and connectors to ensure reliable communication between devices. Here's an overview of the key considerations for CAN bus cabling and connectors:

Cabling:

- **Twisted-Pair Shielded Cables:** CAN bus typically uses twisted-pair shielded cables with a characteristic impedance of 120 ohms. Twisting the wires helps minimize electromagnetic interference (EMI) and crosstalk.
- **Shielding:** High-quality shielding helps prevent EMI from interfering with CAN signals. The cable shield should be grounded at multiple points along the bus to provide a low-impedance return path for noise.
- **Noise Immunity:** Shielded cables are essential for maintaining signal integrity and preventing noise-induced errors, ensuring reliable communication in noisy environments.

Connectors:

- **Industrial-Grade Connectors:** CAN bus connectors are typically rugged, industrial-grade connectors designed to withstand harsh operating conditions, including vibration, shock, and temperature extremes.
- **D-Sub or M12 Connectors:** Common connector types for CAN bus include D-Sub (e.g., DB9) and M12 connectors, chosen for their durability and reliability in automotive, industrial, and transportation applications.
- **Reliable Connection:** Connectors must provide a secure, low-resistance connection to the bus lines to maintain signal integrity and minimize the risk of signal reflections or EMC issues.

- **Characteristic Impedance Matching:** Proper connector design and installation are essential to ensure that the connectors do not introduce impedance mismatches or signal distortions that could impair communication on the CAN bus.

Installation Considerations:

- **Proper Termination:** Ensure proper termination of the CAN bus with termination resistors at each end of the bus to minimize signal reflections and maintain signal integrity.
- **Routing and Dressing:** Carefully route and dress the cables to minimize interference from other electrical systems and sources of noise, such as motors or power cables.
- **Environmental Protection:** Consider environmental factors such as temperature, moisture, and exposure to chemicals when selecting cabling and connectors to ensure long-term reliability and durability in the intended operating environment.

6.3.5. Termination Requirements

Proper termination is crucial for maintaining reliable communication on a CAN bus network. Here's a detailed overview of termination requirements:

Importance of Termination:

- **Reliable Data Transmission:** Proper termination prevents signal reflections that can distort the communication signals, ensuring reliable data transmission across the network.
- **Prevention of Signal Distortions:** Without proper termination, signal distortions and noise can occur, leading to erratic behavior and compromised network integrity.

Placement of Termination Resistors:

- **At Each End of the Bus:** 120Ω termination resistors should be placed as close as possible to each end of the CAN bus.
- **Minimize Stub Lengths:** This minimizes unterminated stub lengths and ensures proper signal matching along the entire transmission line.
- **Avoiding Middle Placement:** Termination resistors should never be placed in the middle of the bus, as this can cause multiple reflections and degrade signal quality.

Redundant Termination for Fault Tolerance:

- **Recommended for Critical Applications:** Redundant termination circuits provide an additional layer of fault tolerance, ensuring high availability in safety-critical systems.
- **Parallel Configuration:** A second set of 120Ω termination resistors is placed in parallel with the primary terminators.
- **Failure Mitigation:** In case of a primary terminator failure, redundant termination maintains the proper impedance match and keeps the bus operating reliably.

Termination Power Considerations:

- **Heat Dissipation:** Termination resistors dissipate power in the form of heat, requiring proper management.
- **Power Handling Capability:** Select termination resistors with sufficient power handling capability to accommodate the maximum bus load and voltage levels.
- **Active Termination Circuits:** In some cases, active termination circuits with power dissipation management may be needed to ensure stable and reliable termination across the CAN network's operating range.

6.3.6. EMC and EMI Considerations

Electromagnetic Compatibility (EMC) is crucial for ensuring the reliable operation of a CAN bus network in the presence of electromagnetic interference (EMI). Here's an overview of EMC and EMI considerations:

Electromagnetic Compatibility (EMC):

- **Emissions Control:** CAN bus signals, with their fast rise and fall times, can emit conducted and radiated emissions that may interfere with nearby electronic equipment.
- **Adherence to Standards:** The CAN physical layer must adhere to strict EMC standards, such as CISPR 25 for automotive applications or EN 55011 for industrial environments.
- **Mitigation Techniques:** Techniques such as shielded cabling, common-mode chokes, and filtering are used to limit emissions from the CAN bus.

Immunity to External Interference:

- **External Disturbances:** CAN bus must withstand external electromagnetic disturbances generated by sources like power lines, radio transmitters, or switching power supplies.

- **Compliance with Standards:** Immunity requirements, specified in standards like ISO 11452 or IEC 61000-4, ensure reliable operation even in the presence of high levels of EMI.
- **Protection Circuitry:** Common-mode filters, transient suppressors, and other protection circuitry are used to enhance immunity to external interference.

Grounding and Shielding:

- **Proper Grounding:** Proper grounding and shielding of the CAN bus cabling and components are essential for maintaining EMC.
- **Continuous Return Path:** Cable shield should be connected to a low-impedance ground at multiple points to create a continuous return path for high-frequency noise currents.
- **Prevention of Ground Loops:** Ensuring that CAN transceivers and connected devices are well-grounded to the same reference potential prevents ground loops and common-mode voltage differences.

6.3.7. Fault Tolerance and Diagnostics

Fault Detection and Reporting:

- Robust fault detection and reporting mechanisms are essential for identifying and responding to various failure modes in the CAN bus physical layer.
- Continuous monitoring of the bus state enables CAN transceivers to detect faults such as short circuits, open circuits, and bus-to-ground faults.
- Fault reporting alerts connected controllers to these faults, allowing for appropriate actions like isolating affected nodes, rerouting traffic, or triggering failsafe modes.

Fault Tolerance and Redundancy:

- The CAN physical layer should be designed with fault tolerance to ensure continued operation in case of failures.
- Redundancy mechanisms include redundant bus lines, backup termination circuits, and dual-channel transceivers.
- These redundancy features provide multiple signal paths and failover mechanisms to maintain operation even with component or wiring failures.

Advanced Diagnostic Features:

- Modern CAN transceivers incorporate advanced diagnostic capabilities to monitor the health and performance of the physical layer.

- Diagnostic features may include bus load monitoring, error counters, bus-off state management, and detailed fault logging.
- Exposing these diagnostic parameters to connected CAN controllers enables predictive maintenance, early issue identification, and comprehensive troubleshooting.

6.3.8. Testing and Validation

Comprehensive Testing:

- Rigorous testing of the CAN bus physical layer is essential to ensure its reliability, performance, and compliance with industry standards.
- Various tests, including electrical characterization, signal integrity analysis, EMC verification, and functional validation, are conducted to identify and address potential issues before deployment.

Electrical Characterization:

- Electrical parameters such as signal levels, impedance, common-mode voltages, and slew rates are measured and validated to ensure compliance with specifications.
- Advanced test equipment like oscilloscopes, network analyzers, and impedance meters are used for capturing and analyzing electrical characteristics.

Signal Integrity Validation:

- Rigorous testing evaluates the quality and fidelity of transmitted signals, analyzing factors like eye patterns, jitter, and signal-to-noise ratios.
- Signal integrity analysis identifies potential issues such as reflections, crosstalk, or noise that could lead to communication errors or data loss.

Compliance Verification:

- Compliance with industry standards, such as those defined by ISO, SAE, or IEC, is verified to ensure interoperability and compatibility across different platforms.
- Testing confirms that the CAN bus physical layer meets required electrical, timing, and environmental specifications outlined in these standards.

Thorough testing and validation of the CAN bus physical layer ensure that it operates reliably and meets the stringent requirements of modern automotive, industrial, and other embedded systems applications. By identifying and addressing potential issues early in the development process, engineers can deploy CAN-based systems with

confidence, minimizing the risk of field failures and ensuring optimal performance in real-world environments.

6.3.9. Compliance with Industry Standards

Adherence to Industry Specifications:

- CAN bus physical layer designs must comply with industry standards such as ISO 11898, ISO 16845, ISO 15031, and IEC 61784.
- Compliance ensures interoperability, reliability, and safety across various systems and environments.

Validation and Certification:

- Designs undergo rigorous testing including electrical characterization, signal integrity analysis, EMC testing, and functional validation.
- Accredited test laboratories use standardized methodologies to verify compliance.
- Once validated, designs can be certified by industry bodies, enabling their use in safety-critical and mission-critical applications.

Collaboration with Standards Organizations:

- Standards development involves collaboration among organizations like ISO, SAE, and CiA.
- Experts from automotive, industrial, and transportation sectors define technical requirements, test procedures, and certification processes.
- Active participation ensures that standards evolve to meet industry needs and allows contribution to shaping future technology directions.

Compliance with industry standards is essential for CAN bus physical layer designs to ensure compatibility, reliability, and safety across diverse applications. Thorough validation and collaboration with standards organizations enable manufacturers to meet evolving industry requirements and contribute to advancing CAN bus technology.

6.4. CAN Bus Data Link Layer

The Controller Area Network (CAN) bus operates at the data link layer of the OSI model, facilitating communication between electronic control units (ECUs) within embedded systems, notably in automotive applications. Understanding the CAN data link layer is crucial for optimizing system performance, reliability, and security. This section provides a comprehensive overview of its frame structure, arbitration mechanism, error handling, and other key features.

Frame Structure

- **Message Format:** CAN frames encapsulate application-level data for transmission.
- **Arbitration ID:** Unique identifier prioritizes messages during bus access.
- **Control Field:** Contains metadata such as data length and error signaling.
- **Data Field:** Carries payload data up to 8 bytes.
- **CRC Field:** Checksum ensures data integrity.
- **ACK Field:** Confirms successful reception.
- **Start/End of Frame:** Delimit frame boundaries for synchronization.

Arbitration Mechanism

- **Bitwise Arbitration:** Determines message priority during bus access.
- **Priority-Based Access:** Lower arbitration ID indicates higher message priority.
- **Dynamic Prioritization:** Supports adjusting message priority dynamically.
- **Configurable IDs:** IDs assigned based on message criticality.

Error Handling

- **Error Detection:** Detects bit errors, stuff errors, CRC errors, and form errors.
- **Error Signaling:** Transmits error frames to indicate fault conditions.
- **Retransmission:** Automatic retransmission upon error detection.
- **Error Recovery:** Nodes transition to error-passive or bus-off states.

Robustness and Reliability

- **Efficient Communication:** Message-based design supports real-time control.
- **Fault Tolerance:** Distributed architecture maintains reliable communication.
- **EMC Resilience:** Withstands electromagnetic interference.
- **Deterministic Access:** Arbitration ensures time-critical data delivery.

6.4.1. CAN Bus Data Link Layer Overview

The CAN bus data link layer serves as the backbone for seamless communication between electronic control units (ECUs) in embedded systems, particularly in the automotive industry. This layer performs crucial functions to facilitate reliable and efficient data transfer over the CAN network. Here's an overview of its key features and functionalities:

Frame Encapsulation

- **Standardized Frames:** Encapsulates application-level data into standardized CAN frames.
- **Efficient Structure:** Frames consist of an arbitration ID, control field, data field, CRC field, ACK field, and start/end of frame, facilitating efficient communication.

Deterministic Arbitration

- **Fair Bus Access:** Implements a deterministic arbitration mechanism for fair and prioritized bus access.
- **Bitwise Arbitration:** Prioritizes messages based on their arbitration IDs, ensuring critical data is delivered first.
- **Real-Time Communication:** Supports real-time, safety-critical applications with predictable and reliable communication.

Error Detection and Handling

- **Robust Error Detection:** Implements robust error detection mechanisms to maintain data integrity.
- **Error Signaling:** Transmits error frames to indicate fault conditions and enable error recovery.
- **Automatic Retransmission:** Upon error detection, initiates automatic retransmission to ensure data delivery.
- **Fault Confinement:** Supports fault confinement mechanisms to isolate faulty nodes and prevent network disruptions.

Additional Capabilities

- **Message Filtering:** Provides mechanisms for message filtering to optimize data transfer.
- **Remote Frame Handling:** Supports remote frame transmissions for dynamic message prioritization.
- **Advanced Features:** Enables support for advanced features like fault diagnosis and bus diagnosis, enhancing system performance and reliability.

6.4.2. CAN Bus Frame Structure

The CAN data link layer utilizes a well-defined frame structure to encapsulate application-level data and transmit it over the shared bus. This frame structure is crucial for ensuring efficient and reliable communication between the connected ECUs. The key elements of the CAN bus frame include:

Start of Frame (SOF): This single-bit field marks the beginning of a CAN frame and allows all nodes to synchronize their bit timing.

Arbitration Field: This field contains the 11-bit (standard) or 29-bit (extended) identifier that determines the priority of the message. It also includes the Remote Transmission Request (RTR) bit, which indicates whether the frame is a data frame or a remote frame.

Control Field: This field specifies the data length code (DLC), which indicates the number of data bytes in the frame, and other control information.

Data Field: This field contains the actual application-level data being transmitted, with a maximum of 8 bytes.

CRC Field: The Cyclic Redundancy Check (CRC) field is used to detect errors in the transmitted frame by calculating a checksum over the entire frame.

Acknowledge (ACK) Field: This field allows receiving nodes to acknowledge the successful reception of a frame by setting the ACK slot to a dominant state.

End of Frame (EOF): The End of Frame field marks the end of the CAN frame and allows for bus idle time between consecutive transmissions.

The precise formatting and timing of these frame elements are critical for ensuring reliable and deterministic communication over the CAN bus. The CAN data link layer's frame structure, combined with its arbitration mechanism and error-handling capabilities, enables the efficient and secure exchange of data between the connected ECUs, making it a fundamental component of modern embedded systems and industrial automation solutions.

6.4.3. CAN Bus Arbitration Mechanism

Contention Resolution: When multiple nodes on a CAN bus try to transmit data at the same time, the CAN arbitration mechanism ensures fair and deterministic access to the shared communication channel. It uses a bitwise arbitration process that prioritizes the node transmitting the message with the highest-priority identifier. Each node monitors the bus during transmission and backs off if it detects a dominant bit (logical '0') while it's transmitting a recessive bit (logical '1'), indicating that a higher-priority message is being sent.

Identifier-Based Prioritization: The arbitration process relies on the message identifier, found in the CAN frame's arbitration field, to determine message priority. Lower numeric

values represent higher priority. This identifier-based prioritization ensures critical data, like brake commands or engine control signals, is transmitted before lower-priority messages, maintaining real-time responsiveness in automotive and industrial control systems.

Non-Destructive Arbitration: CAN arbitration is "non-destructive" because losing nodes don't have their messages corrupted or overwritten. They simply stop transmitting and wait for the bus to become idle before trying again. This prevents data loss, ensuring all messages are eventually delivered, even with high bus utilization and multiple simultaneous transmission requests.

6.4.4. CAN Bus Error Handling

The CAN data link layer incorporates robust error detection and handling mechanisms to ensure reliable and consistent data delivery across the shared communication bus. This error management system is crucial for maintaining the overall integrity and deterministic behavior of CAN-based embedded systems, particularly in safety-critical applications.

Error Detection Techniques: The CAN protocol employs several error detection techniques, including frame structure checking, bit error monitoring during transmission, and validation of the cyclic redundancy check (CRC) value in each frame. If an error is detected, the offending node immediately transmits an error flag to notify all other nodes on the bus, interrupting the current transmission and triggering a bus-wide error recovery process.

Fault Confinement Strategy: CAN implements a fault confinement strategy to isolate faulty nodes from the network. Each node maintains transmit and receive error counters, which are incremented upon error detection. If these counters exceed predefined thresholds, the node enters one of several error states, from error active (still participating in bus communications) to bus off (completely removed from the network until manually reset).

Multi-Layered Error Handling: This multi-layered approach, combined with CAN's ability to prioritize critical messages, ensures high reliability, fault tolerance, and real-time performance. Even in the presence of sporadic errors or temporary bus disturbances, CAN-based systems maintain integrity and safety. Proper implementation of these error detection and confinement mechanisms is essential for meeting the demanding requirements of industrial automation, automotive, and other mission-critical applications.

6.4.5. CAN Bus Message Prioritization

Identifier-Based Prioritization: CAN prioritizes messages based on their unique identifier. Each frame contains an arbitration field with an 11-bit (standard) or 29-bit (extended) identifier. Lower numeric values correspond to higher priority messages. This ensures critical data, like emergency stop commands or engine control signals, is transmitted before lower-priority information.

Dynamic Message Prioritization: CAN allows dynamic message prioritization using the Remote Transmission Request (RTR) bit. By setting the RTR bit to a recessive state, a node can request data from another node, boosting the priority of the responsive message. This ensures responsive messages are transmitted ahead of pending messages, enhancing real-time responsiveness.

Fault-Tolerant Prioritization: CAN's message prioritization is fault-tolerant. The bitwise arbitration process, coupled with robust error handling, ensures higher-priority messages are successfully transmitted without corruption. Even in the presence of errors or bus disturbances, critical data is reliably delivered, maintaining the real-time responsiveness and safety-critical operation of CAN-based systems.

6.4.6. CAN Bus Timing Requirements

Bit Timing: CAN operates at a fixed bit rate, typically ranging from 10 kbit/s to 1 Mbit/s. Nodes must be configured with the same bit rate for synchronization. Timing segments like synchronization, propagation, and phase allow accurate bit sampling and compensate for propagation delays.

Interframe Spacing: A minimum interframe spacing, called intermission, is required between consecutive CAN frames. This provides bus idle time, allowing nodes to regain access after a successful transmission, maintaining communication channel integrity and preventing collisions.

Error Handling Timing: CAN has strict timing requirements for error detection and handling. Nodes must promptly transmit error flags upon detection, ensuring rapid response and channel recovery. Timing thresholds for error counters determine when nodes enter restrictive error states like error passive or bus off.

Bit Stuffing Timing: CAN employs bit stuffing to maintain synchronization, adding a complementary bit after every 5 consecutive bits of the same value. Consistent application and detection of bit stuffing timing are crucial for frame integrity.

Adherence to these timing requirements ensures CAN operates reliably, deterministically, and in real-time. This enables seamless integration and coordination of electronic control units (ECUs) in automotive and industrial automation systems

6.4.7. CAN Bus Physical Layer Interface

Cabling: The CAN physical layer typically employs twisted-pair cables with a characteristic impedance of 120 ohms. This configuration minimizes noise and crosstalk, ensuring signal integrity.

Transceivers: Transceiver circuits at each node drive the bus to appropriate voltage levels and detect incoming signals. They feature fault protection, bus fault confinement, and slew rate control to enhance reliability.

Electrical Characteristics: Strict electrical specifications include common-mode voltage ranges, differential voltage thresholds, and bit timing parameters. Proper configuration ensures optimal performance and interoperability between connected ECUs.

Fault Protection: Transceivers incorporate fault protection mechanisms to withstand short circuits, over-voltage, and other faults without damage. This ensures continuous operation and prevents bus disruption.

Bus Fault Confinement: In the event of a fault, transceivers isolate affected nodes to prevent propagation of errors throughout the network. This enhances reliability and fault tolerance.

Slew Rate Control: Slew rate control regulates the transition speed of signals, reducing electromagnetic interference and ensuring signal integrity over long cable lengths.

Validation: Thorough validation of physical layer attributes is essential to ensure compliance with specifications and achieve reliable data transmission across the network, even in harsh environmental conditions.

By adhering to these physical layer interface requirements, CAN bus systems can reliably transmit data in automotive and industrial environments, where electromagnetic interference and other environmental factors are common.

6.4.8. CAN Bus Diagnostic and Testing Requirements

Error Handling and Fault Confinement:

- Comprehensive error handling mechanisms allow quick identification and isolation of faults.
- Monitoring internal error counters and bus state facilitates rapid troubleshooting.

Bus Monitoring and Diagnostics:

- Robust tools for bus monitoring enable analysis of historical bus activity and error patterns.
- Proactive detection of potential issues aids in preemptive maintenance and troubleshooting.

Fault Injection and Stress Testing:

- Rigorous fault injection and stress testing simulate real-world scenarios to validate system resilience.
- Evaluation of fault detection, isolation, and recovery mechanisms ensures robustness.

Interoperability Testing:

- Validation of interoperability between CAN-enabled components from different manufacturers ensures seamless integration.
- Testing compatibility across diverse applications and environments enhances reliability and performance.

Conformance Testing:

- Rigorous conformance testing verifies adherence to CAN protocol specifications.
- Validation of frame structure, arbitration, and error handling ensures compatibility and reliable operation.

Comprehensive diagnostic and testing capabilities are essential for optimizing the performance, reliability, and maintainability of CAN-based embedded systems. By implementing these measures, system designers and maintenance personnel can deliver robust solutions that meet the demanding requirements of modern industrial, automotive, and mission-critical applications.

6.5. CAN Bus Application Layer

Key Components of the CAN Bus Application Layer:

1. **Message Encoding**: The application layer defines how data is encoded within CAN messages. This includes specifying the format of message identifiers, data fields, and any additional control or status information.
2. **Message Types**: Different types of messages are defined within the application layer to represent various types of data and commands. Common message types include:
    - Data Frames: Used for transmitting application-level data between nodes.
    - Remote Frames: Used to request data from a remote node.

    - Error Frames: Used to indicate transmission errors or bus faults.
3. **Message Filtering**: The application layer may define filtering mechanisms to allow nodes to selectively receive messages based on their identifiers or content. This helps reduce bus traffic and improve system efficiency.
4. **Network Management**: Some CAN-based systems include network management functions within the application layer. This may include features for node discovery, configuration, and dynamic reconfiguration of the network topology.
5. **Application-specific Protocols**: In many cases, the CAN application layer incorporates higher-level protocols tailored to specific application domains. For example, automotive systems may use protocols like OBD-II (On-Board Diagnostics) or J1939 for vehicle diagnostics and communication.

Role of the Application Layer:

- **Data Exchange**: Facilitates the exchange of application-specific data between nodes in the CAN network.
- **Command and Control**: Supports command and control functions, allowing ECUs to send instructions or request actions from other nodes.
- **System Configuration**: Manages system configuration and initialization, including setting node parameters and configuring message filtering rules.
- **Error Handling**: Defines error detection and handling mechanisms at the application layer, such as message timeouts or retries.
- **Integration with Higher-level Systems**: Interfaces with higher-level systems or applications, translating between CAN messages and higher-level protocols or data formats.

Importance of the Application Layer:

- **Customization**: Allows customization of communication protocols to suit the specific requirements of different applications and industries.
- **Efficiency**: Defines efficient message formats and filtering mechanisms to optimize bus utilization and reduce latency.
- **Interoperability**: Enables interoperability between different nodes and systems by defining standardized message formats and protocols.
- **Fault Tolerance**: Implements error handling and recovery mechanisms to maintain system reliability and integrity in the face of transmission errors or bus faults

6.5.1. Purpose and Scope of the Requirements Specification

This requirements specification document provides a comprehensive framework for the development and implementation of the CAN bus application layer. Its primary purpose

is to define the essential functional, technical, and operational requirements necessary for seamless integration and interoperability of CAN-enabled devices and systems.

The document's scope covers the entire lifecycle of the CAN application layer, from initial design and configuration to ongoing maintenance and upgrades. It aims to establish a clear and consistent framework for designing, implementing, and testing CAN bus application layer components. Additionally, it ensures compliance with relevant industry standards and best practices for CAN bus communication protocols.

Key objectives of this document include providing a detailed technical reference for engineers, developers, and system integrators involved in integrating CAN-enabled devices and applications. It also aims to facilitate the development of robust, scalable, and future-proof CAN bus systems by addressing potential challenges and risks upfront.

By defining clear requirements and expectations for the CAN application layer, this document serves as a crucial guide for project stakeholders, including manufacturers, vendors, and end-users. It informs the development of corresponding test plans, validation procedures, and deployment strategies to ensure the integrity and reliability of the CAN bus infrastructure throughout its lifecycle

6.5.2. Applicable Standards and Protocols

The CAN bus application layer operates within a well-defined ecosystem of industry standards and communication protocols, ensuring interoperability, reliability, and compatibility across a wide range of CAN-enabled devices and systems. The key standards and protocols governing the CAN application layer include:

1. **ISO 11898 Series:** Developed by the International Organization for Standardization (ISO), this series specifies the physical layer, data link layer, and application layer requirements for CAN bus communication. It covers aspects such as electrical signaling, message formatting, and error handling, providing a comprehensive framework for CAN-based systems.
2. **SAE J1939 Standard:** Widely utilized in automotive and commercial vehicle industries, SAE J1939 defines a specific application layer protocol and message set for CAN-based vehicle networks. It includes standardized data elements, message priorities, and diagnostic capabilities tailored for automotive applications.
3. **DeviceNet:** Developed by Allen-Bradley (now Rockwell Automation), DeviceNet is an open industrial network protocol leveraging the CAN bus physical layer and data link layer. It provides a standardized application layer for connecting industrial automation devices and equipment, facilitating seamless integration and interoperability in industrial environments.

4. **CANopen:** An open, high-level communication protocol for industrial automation and embedded systems, CANopen specifies application layer functions, device profiles, and communication protocols for CAN-based networks. It enables interoperability between a wide range of devices and systems, making it suitable for various industrial applications.
5. **NMEA 2000:** Developed by the National Marine Electronics Association, NMEA 2000 defines the application layer for CAN-based communication in marine electronics and instrumentation systems. It standardizes communication protocols for devices such as GPS receivers, depth finders, and engine monitoring systems, ensuring compatibility and interoperability in marine applications.

By aligning with these established industry standards, designers and developers can ensure their CAN-based systems and components are compatible, scalable, and able to seamlessly integrate with a wide range of other CAN-enabled devices and applications. This level of standardization is critical for the widespread adoption and long-term viability of CAN bus technology across diverse industries and applications.

6.5.3. CAN Bus Architecture and Components

The CAN bus architecture is built to enable reliable, real-time communication between multiple electronic devices and systems. At its core lies the CAN bus, the physical medium for data transmission. Connected to the CAN bus are various CAN nodes, representing distinct devices or components capable of transmitting and receiving CAN messages.

1. **CAN Controller:** This component manages communication protocols and message formatting. It handles tasks such as message transmission, reception, and error detection. The CAN controller implements the CAN protocol and controls the flow of data on the bus.
2. **CAN Transceiver:** Responsible for converting digital signals from the CAN controller into the differential signaling used on the CAN bus and vice versa. It ensures proper electrical communication between the CAN controller and the physical bus medium.
3. **Termination Resistors:** These resistors are essential for maintaining signal integrity by providing proper impedance matching. Placed at each end of the CAN bus, they minimize signal reflections and prevent data corruption. Termination resistors are crucial for reliable communication on the CAN bus.
4. **Power Supplies:** CAN nodes require power to operate. Power supplies provide the necessary voltage and current to power the components of each node, ensuring their proper functioning within the CAN network.
5. **Wiring Harnesses:** Wiring harnesses connect the various components of the CAN bus system, including the CAN nodes, controllers, and transceivers. They

facilitate the transmission of data and power between different parts of the network.

6. **Interface Modules:** Interface modules provide connections between the CAN bus and external systems or devices. They may include features such as isolation, signal conditioning, or protocol conversion to interface with different types of equipment.

In addition to these components, the CAN bus architecture incorporates mechanisms for message arbitration, error handling, and fault tolerance. These mechanisms ensure that critical data is reliably delivered, even in the presence of disruptions or failures within the network. Understanding the roles of each component allows engineers to design and implement CAN-based systems that are scalable, maintainable, and capable of meeting the requirements of various industrial and automotive applications.

6.5.4. CAN Message Structure and Formatting

The CAN message structure and formatting are fundamental to the operation of the CAN bus application layer. Standardized by the ISO 11898 standard, CAN messages serve as the core units of communication, conveying crucial data and control information across the CAN network. Here's a breakdown of the key components of a CAN message:

1. **Arbitration Field:** This field contains the message identifier, which determines the priority of the message. Lower identifier values indicate higher priority. During transmission, nodes with lower identifiers have precedence over nodes with higher identifiers, resolving message conflicts.
2. **Control Field:** This field provides metadata about the message, including the data length code (DLC) and flags such as the remote transmission request (RTR). The DLC specifies the number of bytes in the data field, while the RTR flag indicates whether the message is a data frame or a remote frame.
3. **Data Field:** The data field carries the actual application-specific data, with a maximum length of 8 bytes in standard CAN and 64 bytes in CAN FD (Flexible Data Rate). This field holds the information to be transmitted between CAN nodes, such as sensor readings, control commands, or diagnostic information.
4. **CRC Field:** The cyclic redundancy check (CRC) field contains a checksum calculated based on the message content. This checksum allows receiving nodes to detect errors introduced during transmission, ensuring data integrity. If the calculated CRC does not match the received CRC, an error is flagged, and appropriate actions can be taken to address the issue.
5. **Start and Stop Delimiters:** These delimiters mark the beginning and end of the CAN frame, helping receiving nodes identify and parse incoming messages accurately.

6. **Interframe Space:** The interframe space provides a gap between successive transmissions, allowing nodes time to process received messages and prepare for the next transmission. This spacing ensures proper timing and coordination within the CAN network.

The specific formatting and encoding of CAN messages may vary depending on the application-layer protocol being used, such as SAE J1939, DeviceNet, or CANopen. These protocols define standardized data element definitions, message layouts, and communication procedures tailored to specific industries and applications. By adhering to these standardized message structures and formatting conventions, CAN-based systems can achieve reliable and consistent data exchange, facilitating interoperability and simplifying system integration and maintenance.

6.5.5. CAN Message Transmission and Reception

Message transmission and reception are fundamental processes in the CAN bus communication protocol, ensuring reliable and efficient data exchange between nodes within the network. Here's a detailed explanation of how message arbitration, transmission, and reception work in the CAN bus:

1. **Message Arbitration:**
    - When multiple nodes attempt to transmit messages simultaneously on the CAN bus, the arbitration mechanism determines which message gets priority.
    - Each message frame includes an arbitration field containing the identifier. Nodes compare the identifiers of the messages being transmitted.
    - The message with the lowest identifier value has the highest priority and wins the arbitration process. Lower identifiers take precedence over higher ones.
    - This ensures that critical messages are transmitted without interruption, maintaining the real-time responsiveness of the CAN network.
2. **Message Transmission:**
    - To transmit a message, a node first checks the bus to ensure it is idle. If the bus is idle, the node begins sending its message frame.
    - The message frame includes the arbitration field, control field, data field, and CRC field, formatted according to the CAN protocol.
    - The node transmits the message frame using differential signaling, where logical '0' is represented by a dominant (low) voltage and logical '1' by a recessive (high) voltage.
    - The transmitting node monitors the bus during transmission for any errors or collisions. If an error is detected, the node aborts the transmission and retries later.

3. **Message Reception:**
    - Nodes continuously monitor the CAN bus for incoming messages. When a message is detected, the receiving node reads the identifier from the arbitration field.
    - If the identifier matches the node's own identifier or the message is marked for broadcast, the node processes the message.
    - The node extracts the data from the message frame and performs any necessary validation or error checking, such as CRC verification.
    - If the message passes validation, the node acts on the data contained within it. If an error is detected, the node may request retransmission or take appropriate corrective action.

6.5.6. CAN Bus Error Handling and Diagnostics

1. **Comprehensive Error Detection:**
    - The CAN bus protocol employs various mechanisms for error detection to ensure data integrity. These include:
        - Cyclic Redundancy Check (CRC) field: Included in every CAN message frame, the CRC allows receiving nodes to verify the correctness of the transmitted data.
        - Bit stuffing monitoring: Nodes continuously monitor the bus for violations of the bit stuffing rule, indicating potential errors in the received data.
        - Form error detection: Nodes check for errors in the structure of received frames, such as incorrect frame length or format.
        - Acknowledgment error detection: Nodes verify that transmitted frames are acknowledged by other nodes, signaling successful reception.
2. **Error Handling Strategies:**
    - When errors are detected, CAN nodes employ various strategies to handle them and maintain network stability:
        - Automatic retransmission: Nodes may automatically retry transmitting messages if errors occur, ensuring data delivery.
        - Fault confinement: Nodes track error counts and enter error states (such as error passive or bus-off) if error thresholds are exceeded, isolating faulty nodes to prevent network disruption.
        - Communication parameter adjustment: Nodes dynamically adjust communication parameters, such as the bit rate or error detection settings, to adapt to changing network conditions and mitigate errors.
        - Error delimiter frames: Nodes use special error delimiter frames to signal the occurrence of errors and facilitate error recovery procedures.

3. **Diagnostic Capabilities:**
    - The CAN protocol provides comprehensive diagnostic features to facilitate troubleshooting and maintenance:
        - Standardized diagnostic trouble codes (DTCs): Nodes generate DTCs to indicate specific error conditions, aiding in the diagnosis of faults and failures.
        - Remote diagnostics: CAN nodes support remote diagnostic requests, allowing external systems or diagnostic tools to query node status and retrieve diagnostic information.
        - On-board diagnostic logging: Nodes may log diagnostic information, such as error counts, communication statistics, and event timestamps, for later analysis and troubleshooting.

These error handling and diagnostic capabilities ensure the reliability, resilience, and maintainability of CAN-based systems, enabling timely detection, isolation, and resolution of issues to minimize downtime and ensure continued operation.

6.5.7. CAN Bus Security and Access Control

Securing the CAN Bus

As a critical communication infrastructure, the CAN bus must be designed with robust security measures to protect against unauthorized access, data tampering, and other malicious activities. This is especially important in applications such as automotive, industrial automation, and medical devices, where the integrity and confidentiality of CAN communications can have significant safety and regulatory implications.

Security hardening for the CAN bus includes implementing encryption and authentication mechanisms to ensure that only authorized devices can access and transmit data over the network. This may involve the use of cryptographic keys, digital certificates, and secure message authentication codes to verify the legitimacy of CAN messages and nodes.

Access Control Strategies

Controlling access to the CAN bus is essential to maintain the overall security and reliability of the network. This can be achieved through a combination of physical, logical, and monitoring-based access control measures:

Physical access control: Restricting physical access to CAN hardware, such as ECUs, wiring harnesses, and diagnostic ports, to prevent unauthorized tampering or connection of rogue devices.

Logical access control: Implementing authentication and authorization mechanisms, such as user accounts, role-based permissions, and secure protocols, to regulate digital access to the CAN network and its resources.

Monitoring and auditing: Continuously monitoring CAN bus activity, detecting anomalies, and maintaining detailed logs to identify and respond to potential security breaches or unauthorized access attempts.

Layered Security Approach

Securing the CAN bus requires a layered security approach, with multiple defensive measures working in concert to mitigate risks and protect the network. This includes not only securing the CAN bus itself but also ensuring the security of the broader system that the CAN network is integrated into, such as the host ECUs, firmware, and software applications.

By adopting a holistic security strategy, CAN-based systems can better withstand emerging cyber threats, maintain the confidentiality and integrity of critical data, and comply with relevant industry regulations and standards, such as ISO/SAE 21434 for automotive cybersecurity.

Future Considerations

As the complexity and connectivity of CAN-based systems continue to grow, the importance of robust security and access control measures will only increase. Emerging technologies, such as blockchain-based solutions and quantum-resistant encryption, may play a crucial role in enhancing the security of CAN bus communications in the years to come.

Additionally, the development of CAN-specific security protocols and standards will be essential to ensure the interoperability and widespread adoption of secure CAN bus architectures across various industries and applications.

6.5.8. CAN Bus Message Formats

1. **Standard CAN Frame:** This format comprises an 11-bit message identifier (ID), an 8-byte data payload, and various control and error checking fields. It's commonly used and suitable for many CAN bus applications.
2. **Extended CAN Frame:** Expanding the message ID to 29 bits allows for a larger number of unique identifiers. This format is utilized in more complex systems requiring a greater number of ECUs or more detailed messages.

3. **Remote Transmission Request (RTR) Frame:** This special message type enables an ECU to request data from another node without continuously transmitting requests. It's helpful for scenarios where data retrieval is needed intermittently.
4. **Error Frame:** The error frame signals the detection of transmission errors. When an ECU identifies an error, it transmits this frame to notify all other nodes, facilitating rapid detection and retransmission of corrupted data.
5. **Overload Frame:** Used to indicate that an ECU needs additional time to process a received message before accepting the next frame. This prevents data loss and ensures proper timing of CAN bus communications.

Choosing the appropriate message format depends on factors like the number of ECUs, data exchange frequency, criticality, and the need for extended addressing or diagnostic capabilities. By adhering to these standardized formats, CAN bus systems achieve high interoperability, reliability, and real-time performance, making them indispensable in automotive, industrial, and transportation sectors

6.6. CAN Bus Error Handling

Error handling in a Controller Area Network (CAN) bus system is crucial for maintaining the integrity and reliability of communication between nodes. Here's an overview of the key aspects of CAN bus error handling:

1. **Error Detection:** CAN nodes continuously monitor the bus for errors during message transmission and reception. Various types of errors can occur, including bit errors, frame errors, stuff errors, acknowledge errors, and CRC errors.
2. **Error Signaling:** When a node detects an error, it immediately transmits an error frame onto the bus. This error frame indicates the presence of an error and interrupts the ongoing communication. All nodes on the bus receive and process the error frame, allowing them to take appropriate actions.
3. **Error Counters:** Each CAN node maintains error counters to track the number of errors encountered during transmission and reception. There are typically two types of error counters: transmit error counter (TEC) and receive error counter (REC). These counters are instrumental in diagnosing the health of the CAN network and identifying potential issues.
4. **Error Handling Modes:** CAN nodes can operate in different error handling modes based on the status of their error counters. These modes include:
    - **Error Active:** The node operates normally as long as the error counters are below a certain threshold.
    - **Error Passive:** If the error counters exceed the threshold, the node enters the error passive mode. In this mode, the node still participates in communication but takes additional precautions to minimize its impact on the network.

- **Bus Off:** If the error counters reach critical levels, the node enters the bus off state. In this state, the node is essentially removed from the network and cannot participate in communication until it is reset manually or automatically after a predefined recovery time.

5. **Error Recovery:** Nodes in error passive or bus off states can recover and rejoin the network once certain conditions are met. This recovery process typically involves decreasing the error counters over time and gradually transitioning back to the error active state.
6. **Fault Confinement:** The CAN protocol incorporates fault confinement mechanisms to isolate faulty nodes from the rest of the network. By entering error passive or bus off states, malfunctioning nodes prevent further disruption to the network and enable other nodes to continue operating normally.

6.6.1. Purpose and Scope of the Requirements Specifications

The comprehensive requirements specification document you've outlined is indeed critical for ensuring the effective development and implementation of robust CAN bus error handling mechanisms. By establishing clear guidelines and standards for identifying, classifying, and managing errors, this document lays the foundation for creating reliable and resilient CAN-based systems. Here's a breakdown of the purpose and scope you've outlined:

1. **Purpose:**
   - **Framework Establishment:** It sets up a standardized framework for handling errors within CAN-based communication networks, ensuring consistency and reliability across different implementations.
   - **Fault Tolerance and Reliability:** The document aims to prioritize fault tolerance and reliability, ensuring that CAN-based systems can continue to operate seamlessly even in the presence of errors or disruptions.
   - **Seamless Operation:** By providing guidance on error detection, reporting, and resolution, the document helps maintain seamless operation of CAN networks, minimizing downtime and disruptions.
2. **Scope:**
   - **Lifecycle Coverage:** It covers the entire lifecycle of CAN bus error handling, from detection to resolution, ensuring that all aspects of error management are addressed comprehensively.
   - **Wide Range of Topics:** The scope encompasses various topics related to error handling, including error classification, detection algorithms, logging protocols, handling strategies, and system-wide verification.

- **Applicability:** The document's guidelines are applicable to manufacturers, system integrators, and software developers working with CAN-based systems, ensuring broad applicability across the CAN ecosystem.

3. **Goals:**
   - **Comprehensive Guide:** The document aims to be a comprehensive guide that provides practical and actionable guidance for implementing effective error handling mechanisms in CAN-based systems.
   - **Resilience and Adaptability:** By adhering to the guidelines outlined in the document, organizations can create CAN-based products and solutions that are resilient, adaptable, and able to meet the evolving demands of modern embedded systems and Industry 4.0 applications.

6.6.2. Definitions and Abbreviations

The provided definitions and abbreviations offer a solid foundation for understanding the key concepts and terminology related to CAN bus error handling. They ensure clear and consistent communication throughout the requirements specification document. Here's a breakdown of each term:

1. **CAN (Controller Area Network):** The defined communication protocol used for efficient and reliable data exchange between connected devices in various industries.
2. **Error:** Any disruption to the normal operation of the CAN bus, resulting in data corruption, loss, or misinterpretation during transmission.
3. **Error Frame:** A specialized CAN frame transmitted by a node upon error detection, signaling the presence of an issue and triggering appropriate error handling mechanisms.
4. **Error Passive:** A state in which a CAN node exceeds a defined error threshold but can still participate in network communication, indicating a potential problem that requires attention.
5. **Error-Active:** The standard operational state of a CAN node, where it can transmit and receive frames without detecting errors.
6. **Bus Off:** A state in which a CAN node is temporarily or permanently removed from the network due to an excessive number of errors, requiring resolution before rejoining the communication.
7. **Error Counters:** Internal registers within a CAN node that track the number of detected errors, helping determine the node's error state and triggering appropriate actions.

6.6.3. CAN Bus Error Types and Classifications

Classifications:

1. **Bit Errors:** These errors occur due to discrepancies between the transmitted and observed bits, often caused by noise, signal degradation, or timing issues.
2. **Stuff Errors:** Occur when more than five consecutive bits of the same logical value are detected, violating the CAN protocol's bit stuffing rule.
3. **Form Errors:** Arise from detecting an invalid CAN frame structure, such as incorrect delimiters or unexpected message formats, potentially caused by hardware malfunctions or firmware bugs.
4. **Acknowledgment (ACK) Errors:** Happen when a transmitting node doesn't receive an ACK signal from the intended recipient(s) of a CAN frame, indicating potential issues like full receive buffers or disconnections.

These classifications provide a structured approach to understanding and addressing various types of errors encountered in CAN bus communication, aiding in the development of effective error detection and handling strategies.

6.6.4. CAN Bus Error Detection Mechanisms

Error Detection Mechanisms:

1. **Bit Monitoring:** Nodes continuously compare sent or expected bits with the actual bus state. Any discrepancies trigger bit errors, detected through continuous monitoring during transmission and reception.
2. **Stuff Bit Counting:** Controllers maintain a count of consecutive identical bits. If more than five are detected, violating the bit stuffing rule, a stuff error occurs, indicating potential issues with signal integrity or synchronization.
3. **Frame Structure Validation:** Nodes validate received frame structures, ensuring elements like delimiters, control fields, data fields, and CRCs are present and correct. Deviations trigger form errors, signaling potential problems with frame integrity.
4. **Acknowledgment (ACK) Monitoring:** Upon frame transmission, nodes listen for ACK signals from recipients. The absence of an ACK indicates an ACK error, suggesting issues like a recipient's inability to accept the message or physical disconnection.
5. **Error Counters and Thresholds:** Nodes maintain error counters for transmit and receive errors. Exceeding predefined thresholds updates the node's error state, leading to actions like transitioning to "Error Passive" or "Bus Off" states.

These mechanisms collectively ensure the timely detection and reporting of errors, enabling appropriate error handling actions to maintain CAN bus integrity and reliability.

6.6.5. CAN Bus Error Reporting and Logging Requirements

Error Reporting and Logging Requirements:

1. **Real-time Error Reporting:** Nodes must promptly detect, classify, and report errors by generating error frames transmitted onto the bus. These frames should contain detailed information about the error type, triggering conditions, and a timestamp for analysis.
2. **Comprehensive Error Logging:** Each node should maintain a detailed error log, recording the occurrence, severity, and contextual data for every detected error. This log must persist through node resets or power cycles to ensure critical diagnostic information is retained.
3. **Accessibility of Error Logs:** Error logs should be accessible to external diagnostic tools and interfaces, allowing engineers and technicians to analyze error history efficiently. This accessibility facilitates quick identification and resolution of issues.
4. **Standardized Error Classification:** Implement standardized error classification and severity levels across all nodes. Errors should be categorized based on their impact on network operation, distinguishing between minor, recoverable issues and critical faults requiring immediate attention.

By adhering to these requirements, CAN-based systems can establish robust error handling and diagnostic capabilities, ensuring the reliability and traceability of network communication throughout the system lifecycle.

6.6.6. CAN Bus Error Handling Strategies and Algorithms

Error Handling Strategies and Algorithms:

1. **Deterministic Error Handling:** Utilize deterministic algorithms based on standardized error detection mechanisms to swiftly identify and respond to errors. Algorithms should be rule-based and predictable, ensuring coordinated error management across the network.
2. **Prioritized Error Response:** Prioritize error response based on severity. Minor errors may be handled locally, with affected nodes retransmitting corrupted frames or entering an "Error Passive" state. Critical errors may necessitate transitioning into a "Bus Off" state to preserve network integrity.
3. **Coordinated Error Recovery:** Enable coordinated error recovery by leveraging standardized error reporting mechanisms. Nodes should broadcast error frames to alert peers, allowing them to adjust behavior accordingly and initiate recovery procedures like bus-off node reintegration.
4. **Dynamic Error Thresholds:** Implement adaptive error thresholds to accommodate changing network conditions. Thresholds can be dynamically adjusted based on real-time factors such as electromagnetic interference or

temporary disconnections, ensuring reliable communication while avoiding unnecessary node exclusions.

By implementing these strategies and algorithms, CAN-based systems can effectively detect, classify, and respond to errors, maintaining network stability and reliability in diverse operating environments.

6.6.7. CAN Bus Error Recovery and Reset Procedures

CAN Bus Error Recovery and Reset Procedures:

1. **Bus Off Recovery:**
    - When a CAN node enters the "Bus Off" state due to excessive errors, it waits for a defined period, known as the "Bus Off Recovery Time," to ensure the bus is idle.
    - After the recovery time elapses, the node automatically transitions back to the "Error Active" state, resuming normal communication.
    - This standardized process ensures coordinated reintegration, preventing immediate reconnection and further disruptions.
2. **Soft Reset:**
    - A Soft Reset procedure allows a node to quickly recover from less severe errors or transient issues without entering the Bus Off state.
    - It involves resetting the node's internal error counters while maintaining its connection to the bus.
    - Soft Reset is fast and efficient, minimizing downtime and network impact.
3. **Hard Reset:**
    - In cases of severe errors or persistent issues, a Hard Reset procedure may be necessary.
    - Hard Reset involves a complete power cycle or reinitialization of the node, clearing all internal state and error counters.
    - It may require manual intervention or commands from a supervisory controller.
    - Hard Reset should be a last resort due to its disruptive nature, used when other recovery mechanisms fail to resolve the issue.

6.7. CAN Bus Diagnostics

Diagnostic Tools

CAN bus diagnostics rely on specialized tools and software that can interface with the network and extract valuable data for troubleshooting and maintenance. These tools, often used by mechanics and technicians, are capable of reading diagnostic trouble codes, monitoring real-time data from sensors and ECUs (Electronic Control Units),

and performing advanced tests to identify the root cause of any issues within the CAN bus system.

Diagnostic Software

Alongside the hardware tools, CAN bus diagnostics also involve the use of sophisticated software applications that provide a comprehensive view of the network's behavior and performance. These software platforms allow technicians to analyze message traffic, monitor error conditions, and even reprogram or update ECU firmware as needed. By leveraging the data and insights provided by the diagnostic software, maintenance crews can quickly identify and resolve problems, minimizing downtime and ensuring the continued reliable operation of CAN bus-enabled systems.

Fault Code Diagnosis

One of the key features of CAN bus diagnostics is the ability to interpret diagnostic trouble codes (DTCs) reported by the various ECUs on the network. These fault codes provide valuable information about the nature and location of any issues, allowing technicians to quickly pinpoint the problem and take appropriate corrective actions. The standardized DTC format and the ability to access this data through CAN bus diagnostic tools are essential for maintaining the complex electronic systems found in modern vehicles, industrial equipment, and other CAN bus-enabled applications.

Chapter 7

Types of Attacks on CAN Bus

7.1. Types of Attacks:

When an attacker gains access to the Controller Area Network (CAN) bus, there are various types of attacks they can perform to compromise the integrity, confidentiality, or availability of the communication on the bus. Here are three broad types of CAN bus attacks:

1. Message Manipulation or Injection:

- *Spoofing:* The attacker sends messages on the CAN bus pretending to be a legitimate node. This can involve sending falsified sensor data or injecting malicious commands into the network.
- *Replay Attacks:* The attacker records and then retransmits legitimate messages to manipulate the system's behavior. This can be used to repeat certain actions or confuse the system.

2. Denial-of-Service (DoS) Attacks:

- *Flooding:* The attacker floods the CAN bus with a high volume of messages, overwhelming the network and causing legitimate messages to be delayed or lost.
- *Bus-off Attacks:* By repeatedly transmitting faulty or malformed messages, an attacker can force certain nodes on the CAN bus to go into a "bus-off" state, rendering them temporarily or permanently inoperable.

3.Physical Layer Attack:

- *Electromagnetic Interference (EMI):* Manipulating the physical environment to introduce electromagnetic interference that disrupts the CAN bus communication. This interference can lead to corruption or loss of messages.
- *Voltage Attacks:* Introducing abnormal voltage levels on the bus to disturb or damage connected electronic components. This can be achieved by manipulating power supplies or inducing voltage spikes.

It's important to note that these attacks can have serious consequences, particularly in automotive and industrial contexts where CAN bus is widely used. As a result, securing the CAN bus against unauthorized access and implementing measures to detect and mitigate potential attacks is crucial for maintaining the overall security and reliability of systems relying on this communication protocol

7.2. Attacks on CAN Bus:

Certainly, let's delve into explanations for each of the mentioned attacks on the Controller Area Network (CAN) bus:

1. Bus Flood Attack:

- ❑ Description**:** In a Bus Flood Attack, the attacker floods the CAN bus with a high volume of messages, overwhelming the network.
- ❑ Objective**:** The primary goal is to disrupt normal communication on the CAN bus by consuming the available bandwidth, causing legitimate messages to be delayed or lost.
- ❑ Impact**:** This can lead to performance degradation and potential malfunctions in systems that rely on timely and accurate communication.

2. Simple Frame Spoofing:

- ❑ Description**:** In Simple Frame Spoofing, the attacker sends unauthorized messages onto the CAN bus by mimicking the format of legitimate messages.
- ❑ Objective**:** The goal is to inject false information or commands into the network, potentially causing the receiving nodes to act on the malicious data.
- ❑ Impact**:** This type of attack can lead to incorrect system behavior, as nodes may respond to or act upon falsified information.

3. Adaptive Spoofing:

- ❑ Description**:** Adaptive Spoofing is a more sophisticated form of frame spoofing where the attacker dynamically adjusts their attack strategy based on the responses and behavior of the targeted system.
- ❑ Objective**:** The attacker adapts their spoofed messages to evade detection and more effectively manipulate the system over time.
- ❑ Impact**:** Adaptive Spoofing can be challenging to detect because the attacker modifies their tactics, making it harder for security measures to identify and prevent malicious activity.

4. Error Passive Spoofing Attack:

- ❑ Description**:** In an Error Passive Spoofing Attack, the attacker deliberately introduces errors into their spoofed messages to mimic the behavior of a node experiencing communication issues.
- ❑ Objective**:** By simulating error conditions, the attacker attempts to disrupt normal bus activity and potentially force other nodes into an error-passive state.

- ❑ Impact**:** This type of attack can lead to confusion and reduced reliability on the CAN bus, as other nodes may respond to the simulated error conditions by altering their behavior or even disconnecting from the network.

5. Wire-Cutting Spoofing Attack:

- ❑ Description**:** In a Wire-Cutting Spoofing Attack, the attacker physically cuts the CAN bus wires and connects their own device to one or both severed ends. This allows the attacker to inject malicious messages into the bus.
- ❑ Objective**:** The primary goal is to gain unauthorized access to the CAN bus by physically intercepting and manipulating the communication lines.
- ❑ Impact**:** This attack can lead to unauthorized control over the vehicle or system, enabling the attacker to inject false data, manipulate critical functions, or potentially compromise the safety of the system.

6. Double Receive Attack:

- ❑ Description**:** A Double Receive Attack occurs when an attacker injects a message into the CAN bus and also mimics the same message as if it originated from a legitimate node.
- ❑ Objective**:** The goal is to create confusion by introducing redundancy into the network, making it challenging for other nodes to distinguish between the legitimate and malicious messages.
- ❑ Impact**:** The impact of this attack includes potential errors in decision-making by nodes on the bus, as they may receive conflicting or redundant information.

7. Bus-Off Attack:

- ❑ Description**:** In a Bus-Off Attack, an attacker repeatedly sends faulty or malformed messages, leading to errors that can force specific nodes on the CAN bus into a "bus-off" state.
- ❑ Objective**:** The primary objective is to render targeted nodes temporarily or permanently inoperable, disrupting the overall communication on the bus.
- ❑ Impact**:** Nodes in a bus-off state are unable to participate in communication, leading to a loss of functionality and potential safety risks if critical nodes are affected.

8. Freeze-Frame (Doom Loop) Attack:

- ❑ Description**:** In a Freeze-Frame or Doom Loop Attack, an attacker sends a continuous stream of identical messages with a high-priority identifier, monopolizing the bus and preventing other messages from being transmitted.

- ❑ Objective**:** The goal is to create a denial-of-service situation by monopolizing the bus's bandwidth, causing disruption to normal communication.
- ❑ Impact**:** This attack can lead to delays or loss of critical messages, potentially affecting the performance and safety of the systems relying on the CAN bus.

9. DoS Attack:

All ECU nodes share the same bus resources, and no node is under the administrator's control. In this case, a malicious ECU can increase the bus occupancy without following the bus protocol, causing delays, or even suspending other messages. Therefore, hackers can implement DoS attacks by injecting high-priority messages into the bus in a short time, thus preventing the delivery of other CAN messages. The bus messages with higher priority (e.g., ID = 0x000) were randomly injected into the typical message traces at high speed. For example, a DoS attack can be implemented by injecting packets with an ID of 0x000 at a rate that is ten times the regular message interval.

10. Fuzzy Attack:

A fuzzy attack indicates that an attacker sends random compromised IDs and data to the CAN network at any time, even if the attacker does not have any specific information about the victim. The attacker is only required to send a malicious message in the same form as a normal CAN message all the time, and the vehicle may be successfully attacked. Unlike a DoS attack, the fuzzy attack paralyzes the vehicle's functionality or causes an abnormal ECU reaction instead of preventing regular message delivery by occupying the bus. Therefore, fuzzing attack messages consisting of random IDs and data are randomly inserted into the regular CAN bus traffic. When the fuzz attack was implemented on the test vehicle, the vehicle randomly performed abnormal reactions, such as abnormal engine noise, flashing lights, abnormal increase in power, and gear switching.

7.3. The CAN Packets are Contained in a Message

each message is composed by following values:
• Timestamp: recorded time (s).
• CAN ID: identifier of CAN message in HEX (i.e., 03B1).
• DL: number of data bytes, from 0 to 8.
• DATA [0 7]: data value (byte).

We extracted the feature vector from four dataset freely available for research purposes2 including normal real world CAN messages and four different kinds of injected messages caused by following attacks: dos attack (dos), fuzzy attack (fuzzy), spoofing the drive gear (gear) and spoofing the RPM gauge (rpm) [21]. Dataset were constructed by logging CAN traffic through the OBD-II (On-Board Diagnostics) port from a real vehicle while message injection attacks were performing. Datasets contain each 300 intrusions of message injection. Each intrusion performed for 3 to 5 seconds, and each dataset has total 30 to 40 minutes of CAN traffic. We describe in detail the four types of attacks:

• dos: it represents the denial-of-service attack, performed by injecting messages of '0000' CAN ID every 0.3 milliseconds. The '0000' CAN ID is the most dominant.

• fuzzy: injecting messages of totally random CAN ID and DATA values every 0.5 milliseconds.

• gear/rpm: injecting messages of certain CAN ID related to gear/rpm information every 1 millisecond. The rpm (i.e., revolutions per minute) measures the number of revolutions completed in one minute around a fixed axis. Running an engine at a high RPM may cause damage to the engine and reduce its expected lifespan.

7.4. Summary of Attacks

1. Bus Flood Attack
2. Simple frame spoofing
3. Adaptive spoofing
4. Error Passive Spoofing Attack
5. Wire-cutting spoofing attack
6. Double Receive Attack
7. Bus-off Attack
8. Freeze Doom Loop Attack

7.5. Attack Mitigation Techniques

There are several techniques for mitigating attacks:

• Intrusion detection: This is a technique where the traffic on the bus is inspected for abnormal behaviors. Without hardware support it cannot generally prevent an attack but even so has a use in intelligence gathering and in post-incident forensics.

• Security gateway: This is a hardware approach using a device with two (or more) CAN bus interfaces. The gateway copies only legitimate traffic between the trusted bus (typically a vehicle control network) and an untrusted bus that contains a device that is potentially compromised.

• Encryption: This is generally a software technique (sometimes with hardware assistance) where an ECU protects its CAN bus traffic using cryptographic methods. Only receivers with a key can decrypt a message and verify its legitimacy. There are several issues around practical use of encryption for protecting CAN.

• CAN security hardware: These approaches use a hardware device included on a PCB that monitors the CAN signals to and from the CAN bus and provides various levels of protection.

7.6. Intrusion Detection Systems

Intrusion Detection Systems (IDS) play a crucial role in securing the Controller Area Network (CAN) bus by monitoring and analyzing network traffic for signs of suspicious or malicious activities. Here are explanations for three components of IDS in the context of the CAN bus:

- ❖ Real-Time Traffic Analysis:

  - ➢ Description: Real-time traffic analysis involves monitoring the live data flowing on the CAN bus and identifying abnormal patterns or behaviors that may indicate an intrusion or attack.
  - ➢ How it works: The IDS continuously analyzes the CAN bus traffic, comparing the observed patterns to a baseline of normal behavior. Deviations from the baseline, such as an unexpected surge in message frequency or unusual message identifiers, can trigger alerts or alarms.
  - ➢ Benefits: Real-time traffic analysis enables rapid detection of anomalous behavior, allowing for timely responses to potential security threats on the CAN bus.

❖ Payload Analysis:

- Description: Payload analysis involves inspecting the content of CAN messages, focusing on the actual data (payload) being transmitted.
- How it works: The IDS examines the payload of CAN messages for patterns or signatures associated with known attacks. This can include identifying unauthorized commands, malicious data injection, or attempts to exploit vulnerabilities in the communication protocol.
- Benefits: Payload analysis enhances the ability to detect attacks that involve manipulating the content of CAN messages. It provides a deeper level of inspection beyond the structural aspects of the messages.

Chapter 8

Car-Hacking Dataset for the intrusion detection

8.1. Dataset of My Project

8.1.1. DATASET:

We provide car-hacking datasets which include DoS attack, fuzzy attack, spoofing the drive gear, and spoofing the RPM gauge. Datasets were constructed by logging CAN traffic via the OBD-II port from a real vehicle while message injection attacks were performing. Datasets contain each 300 intrusions of message injection. Each intrusion performed for 3 to 5 seconds, and each dataset has total 30 to 40 minutes of the CAN traffic.

1.DoS Attack : Injecting messages of '0000' CAN ID every 0.3 milliseconds. '0000' is the most dominant.

2.Fuzzy Attack : Injecting messages of totally random CAN ID and DATA values every 0.5 milliseconds.

3.Spoofing Attack (RPM/gear) : Injecting messages of certain CAN ID related to RPM/gear information every 1 millisecond.

8.1.2. DATA ATTRIBUTES

Timestamp, CAN ID, DLC, DATA[0], DATA[1], DATA[2], DATA[3], DATA[4], DATA[5], DATA[6], DATA[7], Flag

1.Timestamp : recorded time (s)
2.CAN ID : identifier of CAN message in HEX (ex. 043f)
3.DLC : number of data bytes, from 0 to 8
4.DATA[0~7] : data value (byte)
5.Flag : T or R, T represents injected message while R represents normal message

8.1.3. OVERVIEW OF DATASETS

ATTACK TYPE	# OF MESSAGES	# OF NORMAL MESSAGES	# OF INJECTED MESSAGES
DoS Attack	3,665,771	3,078,250	587,521
Fuzzy Attack	3,838,860	3,347,013	491,847
Spoofing the drive gear	4,443,142	3,845,890	597,252
Spoofing the RPM gauze	4,621,702	3,966,805	654,897
GIDS: Attack-free (normal)	988,987	988,872	-

Figure 14: OVERVIEW OF DATASETS

8.2.Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a powerful class of deep learning algorithms that have revolutionized the field of computer vision. These specialized neural networks are designed to process and analyze visual data, making them particularly well-suited for tasks such as image recognition, object detection, and image classification.

At the heart of a CNN lies the convolutional layer, which applies a set of learnable filters to the input image. These filters are capable of detecting low-level features like edges and shapes, and as the network goes deeper, they can recognize more complex patterns and high-level concepts. This hierarchical feature extraction process allows CNNs to build a comprehensive understanding of the visual input, making them highly effective for a wide range of visual recognition tasks.

8.2.1Convolutional Layers

Convolutional layers are the core building blocks of Convolutional Neural Networks (CNNs). These layers are designed to automatically extract and learn relevant features from input data, such as images or video frames. The convolutional operation is performed by applying a set of learnable filters, also known as kernels or feature detectors, to the input. Each filter is responsible for detecting a specific type of feature, such as edges, shapes, or patterns.

The convolutional layer operates by sliding the filter across the input, computing the dot product between the filter and the input at each location. This results in a feature map,

which represents the locations and strengths of the detected features. The filter weights are learned during the training process, allowing the network to adaptively identify the most relevant features for the given task.

Convolutional layers are often followed by activation functions, which introduce non-linearity and allow the network to learn more complex representations. Common activation functions used in CNNs include ReLU (Rectified Linear Unit), sigmoid, and tanh. The feature maps generated by the convolutional layers are then typically passed through pooling layers, which reduce the spatial dimensions of the feature maps while preserving the most important information.

8.2.2.Pooling Layers

Pooling layers are a fundamental component of convolutional neural networks (CNNs). They serve to reduce the spatial dimensions of the feature maps, allowing the network to capture more abstract and higher-level representations of the input data. The two most common pooling operations are max pooling and average pooling. Max Pooling selects the maximum value from a small region of the feature map, effectively retaining the most prominent features and discarding less relevant information. This helps the network focus on the most salient aspects of the input, making the model more robust to small translations and distortions in the input. Average Pooling, on the other hand, computes the average value from a small region of the feature map. This operation is often used to smooth out the feature representations, reducing the sensitivity to small variations in the input. Pooling layers are typically inserted periodically throughout the CNN architecture, following the convolutional layers. This hierarchical structure allows the network to learn increasingly complex and abstract representations of the input, which can be leveraged for tasks such as image classification, object detection, and segmentation.

8.2.3.Activation Functions

Activation functions are a crucial component of Convolutional Neural Networks (CNNs) and other deep learning models. These functions introduce non-linearity into the neural network, allowing it to learn and model complex, non-linear relationships in the input

data. The choice of activation function can have a significant impact on the performance and convergence of the model during training.

Some of the most commonly used activation functions in CNNs include the ReLU (Rectified Linear Unit), sigmoid, and tanh functions. The ReLU function is particularly popular due to its simplicity, computational efficiency, and ability to mitigate the vanishing gradient problem. It applies the function max(0, x), which sets all negative inputs to 0 and passes positive inputs unchanged. This sparsity in the activations can lead to faster and more efficient training.

The sigmoid function, on the other hand, maps the input to a value between 0 and 1, making it suitable for binary classification tasks. The tanh function is similar to the sigmoid function, but it maps the input to a range between -1 and 1, which can be more suitable for certain types of problems.

Activation functions can also be combined or modified to create more complex non-linear transformations, such as the Leaky ReLU or Parametric ReLU, which aim to address some of the limitations of the standard ReLU function. The choice of activation function is often an important hyperparameter to tune during the model development process.

8.2.4.Fully Connected Layers

Fully connected layers, also known as dense layers, are a crucial component of Convolutional Neural Networks (CNNs). These layers are responsible for transforming the high-level features extracted by the convolutional and pooling layers into the final output, such as class probabilities or regression values.

In a fully connected layer, every neuron in the layer is connected to every neuron in the previous layer, allowing the network to learn complex non-linear relationships between the input features and the output. These layers are typically placed at the end of a CNN architecture, after the convolutional and pooling layers, and their purpose is to aggregate the spatial and contextual information learned by the previous layers to make a final prediction.

The number of neurons in the fully connected layers, as well as the number of fully connected layers, are hyperparameters that can be tuned to optimize the performance of the CNN for a specific task. Larger fully connected layers can capture more complex relationships, but they also increase the model's complexity and risk of overfitting, especially when the training dataset is small.

8.3.5.CNN Architecture

Convolutional Neural Networks (CNNs) have a distinct architectural design that sets them apart from other neural network models. The core components of a CNN architecture include convolutional layers, pooling layers, and fully connected layers. These layers work together to extract and process features from the input data, ultimately leading to accurate predictions or classifications.

The convolutional layers are responsible for identifying local features in the input data, such as edges, shapes, and textures. These layers apply a set of learnable filters or kernels that slide across the input, performing element-wise multiplication and summing the results to produce feature maps. The size and number of these filters can be adjusted to capture different levels of detail.

Following the convolutional layers are the pooling layers, which aim to reduce the spatial dimensions of the feature maps while preserving the most important information. Common pooling operations include max pooling and average pooling, which extract the maximum or average values from a local neighborhood, respectively. This helps to make the network more robust to small variations in the input data.

8.2.6.Training CNNs

1-Data Preprocessing

Cleaning and normalizing the input data

2-Architectural Design

Selecting the right layers and hyperparameters

3-Optimization

Tuning the model for better performance

4-Validation

Evaluating the model on unseen data

5-Deployment

Integrating the trained model into production systems

Training a convolutional neural network (CNN) involves a multi-step process to ensure the model learns effective feature representations and generalizes well to new data. The first step is data preprocessing, where the input images are cleaned, normalized, and augmented to increase the diversity of the training set. Next, the CNN architecture is designed by selecting the appropriate number and types of convolutional, pooling, and fully connected layers, as well as tuning the hyperparameters such as learning rate, batch size, and regularization.

Once the model is configured, the training process begins, optimizing the network's weights and biases to minimize the loss function. This typically involves techniques like stochastic gradient descent, momentum, and adaptive learning rates. After the model has been trained, it's essential to validate its performance on a separate test set to ensure it generalizes well and doesn't overfit the training data. Finally, the trained model can be deployed in real-world applications, integrating it into production systems and continuously monitoring its performance.

8.2.7.Applications of CNNs

Image Recognition

Convolutional Neural Networks (CNNs) excel at image recognition tasks, such as object detection, image classification, and facial recognition. Their ability to automatically learn hierarchical features from raw pixel data makes them highly effective for a wide range of visual perception applications.

Medical Imaging

CNNs have demonstrated remarkable performance in medical imaging tasks, including the analysis of X-rays, CT scans, MRI images, and other medical data. They can be used for disease detection, tissue segmentation, and the identification of anomalies, helping healthcare professionals make more accurate diagnoses.

Self-Driving Cars

Autonomous vehicles rely heavily on computer vision, and CNNs are at the forefront of this technology. They are used for tasks like lane detection, traffic sign recognition, pedestrian detection, and object avoidance, enabling self-driving cars to navigate safely and efficiently.

8.2.8.Advantages and Limitations of CNNs

Advantages of CNNs

Convolutional Neural Networks (CNNs) offer several key advantages that have made them highly successful in a variety of computer vision and image recognition tasks. Their ability to automatically learn hierarchical features from raw image data, without the need for manual feature engineering, is a significant strength. CNNs are also highly effective at capturing spatial and local dependencies in images, making them well-suited for tasks like object detection, image segmentation, and image classification.

Limitations of CNNs

While CNNs have proven to be powerful tools, they also have some inherent limitations. They can be computationally intensive, especially when dealing with large-scale datasets or high-resolution images, requiring significant hardware resources for training and inference. Additionally, CNNs can be sensitive to hyperparameter tuning, and the process of finding the optimal architecture and hyperparameters can be time-consuming and challenging. Furthermore, the inner workings of CNNs can be difficult to interpret, making it challenging to understand the specific features and representations learned by the network.

8.2.9.CNN Code

1. Importing Libraries

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from sklearn.metrics import precision_score, recall_score, roc_auc_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout

```

Figure 15: Importing Libraries For CNN

This section imports the necessary libraries for data manipulation (Pandas and NumPy), data splitting and scaling (scikit-learn), and building and evaluating the deep learning model (TensorFlow/Keras). precision_score, recall_score, and roc_auc_score are used for evaluating the model's performance.

2. Data Loading and Preprocessing

```python
# Define column names
columns = ["Timestamp", "CAN_ID", "DLC", "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7", "Flag"]

# Load your dataset again, skipping the first row
data = pd.read_csv(r"/content/drive/MyDrive/Colab Notebooks/data/DoS_dataset.csv", names=columns, skiprows=1, low_memory=False)

# Convert hexadecimal values to integers
data['CAN_ID'] = data['CAN_ID'].apply(lambda x: int(x, 16))
data['DLC'] = data['DLC'].astype('int64')

# Define function to handle hexadecimal values
def b16(x):
    if isinstance(x, str):
        return int(x, 16) if all(c in '0123456789ABCDEFabcdef' for c in x) else None
    else:
        return None

# Apply conversion function to hexadecimal columns
for col in ['D0', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']:
    data[col] = data[col].apply(b16)

# Drop rows with missing values
data = data.dropna()
data = data.drop_duplicates()
data.fillna(0, inplace=True)

# Convert float columns to integers
data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']] = data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']].astype(int)

# Convert Flag to int (if applicable)
data['Flag'] = data['Flag'].apply(lambda x: 1 if x == 'R' else 0)

# Drop the 'Timestamp' column
data = data.drop('Timestamp', axis=1)

```

Figure 16: Data Loading and Preprocessing

Define Column Names: Specifies the column names for the dataset.
Load Dataset: Reads the dataset from the specified path, skipping the first row.
Hexadecimal to Integer Conversion: Converts the hexadecimal values in the CAN_ID column to integers.
Conversion Function: Defines a function b16 to handle conversion of hexadecimal string values to integers.
Apply Conversion: Applies the b16 function to the data columns D0 to D7.
Handling Missing Values: Drops rows with missing values and duplicates, and fills any remaining NaN values with 0.
Data Type Conversion: Converts columns D1 to D7 to integer type.
Convert Flag Column: Converts the Flag column values to binary (1 for 'R', 0 for others).
Drop Timestamp Column: Drops the Timestamp column as it is not needed for the analysis.

3. Data Balancing

```
# Separate normal and anomaly data
data_normal = data[data['Flag'] == 0]
data_anomaly = data[data['Flag'] == 1]

# Check which data is smaller
if len(data_normal) < len(data_anomaly):
    # Undersample anomaly data to match normal data size
    data_anomaly_sampled = data_anomaly.sample(n=len(data_normal), random_state=42)
    data_balanced = pd.concat([data_normal, data_anomaly_sampled])
else:
    # Undersample normal data to match anomaly data size
    data_normal_sampled = data_normal.sample(n=len(data_anomaly), random_state=42)
    data_balanced = pd.concat([data_normal_sampled, data_anomaly])

```

Figure 17:Data Balancing

Separate Normal and Anomaly Data: Splits the data into two subsets based on the Flag column.
Balancing Data: Checks which class (normal or anomaly) has fewer samples. It then undersamples the larger class to match the size of the smaller class to create a balanced dataset.
Shuffle Data: Shuffles the balanced dataset and resets the index.
Data Inspection: Prints the data types of the balanced dataset and the first few rows for inspection.

4. Feature and Target Separation

```
# Preprocess the balanced data
X = data_balanced.drop(data_balanced.columns[-1], axis=1)  # Drop the target column from the features
y = data_balanced[data_balanced.columns[-1]].values  # Assign the target column to y

# Convert string columns to numerical format (if any)
X = X.apply(pd.to_numeric, errors='coerce')

```

Figure 18: Feature and Target Separation

Feature-Target Split: Separates the features (X) from the target variable (y).
Convert Strings to Numbers: Converts any remaining string columns in X to numeric format, handling any errors by setting non-convertible entries to NaN.

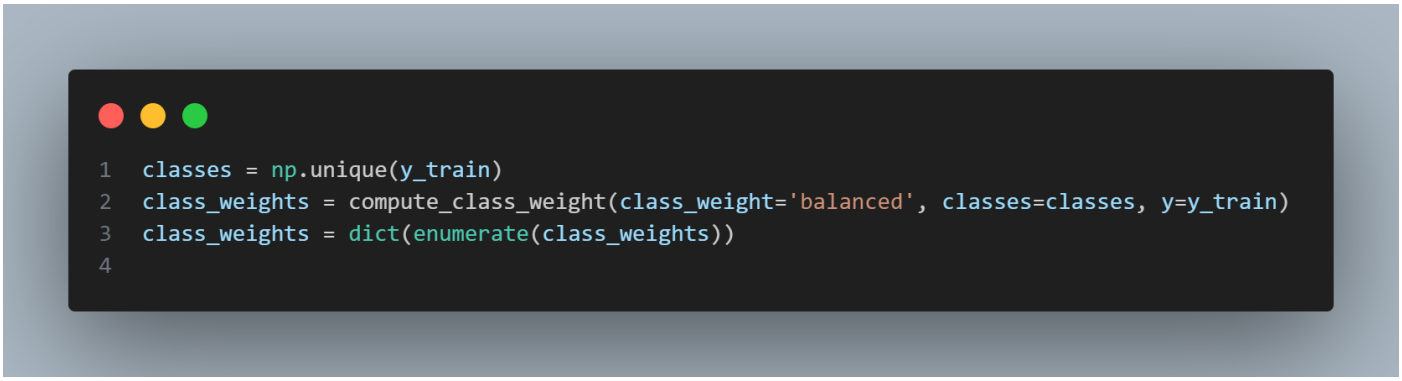
5. Train-Test Split and Normalization

```
# Split the balanced data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalization (only for X)
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for LSTM input
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1],1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1],1)
```

Figure 19: Train-Test Split and Normalization

Train-Test Split: Splits the balanced dataset into training and testing sets, maintaining the class distribution (stratification) with an 80-20 split.
Normalization: Scales the feature data using StandardScaler to have a mean of 0 and a standard deviation of 1.
Reshape Data: Reshapes the feature data to be compatible with the input requirements of a Conv1D model (3D shape: samples, timesteps, features).

6. Compute Class Weights



```
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=y_train)
class_weights = dict(enumerate(class_weights))

```

Figure 20: Compute Class Weights

Computes class weights to handle class imbalance during model training, ensuring that the model gives appropriate importance to both classes.

7.Model Definition

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(x_train.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Conv1D(filters=128, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Dense(1, activation='sigmoid'))

# Print the model summary
model.summary()
```

Figure 21: Model Definition For CNN

Model Definition: Defines a Sequential model for anomaly detection.
Conv1D Layers: Adds two 1D convolutional layers with 64 and 128 filters respectively, each followed by a max pooling layer.
Dropout Layers: Adds dropout layers with a rate of 0.5 after the convolutional layers and the dense layer to prevent overfitting.
Flatten Layer: Flattens the output of the convolutional layers.
Dense Layers: Adds a dense (fully connected) layer with 128 units and a dropout layer, followed by an output layer with a sigmoid activation function for binary classification.
Model Summary: Prints the summary of the model architecture.

8.Model Compilation and Training with Noise

```
loss = 'binary_crossentropy'
optim = 'Adam'
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optim, metrics=metrics)

# Adding noise to training data
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, -1.0, 1.0)

# Training
batch_size = 512
epochs = 5
history = model.fit(x_train_noisy, y_train, batch_size=batch_size,
                    epochs=epochs, validation_data=(x_test, y_test),
                      shuffle=True, verbose=2, class_weight=class_weights)

```

Figure 22: Model Compilation and Training

Model Compilation: Compiles the model using binary cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric.
Adding Noise: Introduces Gaussian noise to the training data to improve the model's robustness.
Training: Trains the model using the noisy training data with a batch size of 512 and 10 epochs, validating on the test set, and applying class weights.

9. Model Evaluation

```python
# Evaluate on the test set
y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Calculate additional metrics
accuracy = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)[1]
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"ROC-AUC: {roc_auc:.4f}")
```

Figure 23: Model Evaluation

Prediction: Makes predictions on the test set and converts the probabilities to binary class labels.
Evaluation: Computes the accuracy, precision, recall, and ROC-AUC score to evaluate the model's performance.
Print Results: Prints the calculated evaluation metrics.

8.2.9.1. Accuracy

1.DOS Attack

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_12 (Conv1D)          (None, 8, 64)             256

 max_pooling1d_12 (MaxPooli  (None, 4, 64)             0
 ng1D)

 dropout_18 (Dropout)        (None, 4, 64)             0

 conv1d_13 (Conv1D)          (None, 2, 128)            24704

 max_pooling1d_13 (MaxPooli  (None, 1, 128)            0
 ng1D)

 dropout_19 (Dropout)        (None, 1, 128)            0

 flatten_6 (Flatten)         (None, 128)               0

 dense_12 (Dense)            (None, 128)               16512

 dropout_20 (Dropout)        (None, 128)               0

 dense_13 (Dense)            (None, 1)                 129

=================================================================
Total params: 41601 (162.50 KB)
Trainable params: 41601 (162.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1837/1837 - 42s - loss: 0.0210 - accuracy: 0.9947 - val_loss: 0.0078 - val_accuracy: 0.9986 - 42s/epoch - 23ms/step
Epoch 2/10
1837/1837 - 37s - loss: 0.0109 - accuracy: 0.9978 - val_loss: 0.0072 - val_accuracy: 0.9984 - 37s/epoch - 20ms/step
Epoch 3/10
1837/1837 - 39s - loss: 0.0106 - accuracy: 0.9979 - val_loss: 0.0073 - val_accuracy: 0.9984 - 39s/epoch - 21ms/step
Epoch 4/10
1837/1837 - 36s - loss: 0.0104 - accuracy: 0.9979 - val_loss: 0.0066 - val_accuracy: 0.9984 - 36s/epoch - 20ms/step
Epoch 5/10
1837/1837 - 39s - loss: 0.0102 - accuracy: 0.9979 - val_loss: 0.0073 - val_accuracy: 0.9984 - 39s/epoch - 21ms/step
Epoch 6/10
1837/1837 - 37s - loss: 0.0102 - accuracy: 0.9979 - val_loss: 0.0075 - val_accuracy: 0.9984 - 37s/epoch - 20ms/step
Epoch 7/10
1837/1837 - 38s - loss: 0.0101 - accuracy: 0.9979 - val_loss: 0.0072 - val_accuracy: 0.9984 - 38s/epoch - 21ms/step
Epoch 8/10
1837/1837 - 37s - loss: 0.0101 - accuracy: 0.9979 - val_loss: 0.0069 - val_accuracy: 0.9984 - 37s/epoch - 20ms/step
Epoch 9/10
1837/1837 - 40s - loss: 0.0101 - accuracy: 0.9979 - val_loss: 0.0064 - val_accuracy: 0.9984 - 40s/epoch - 22ms/step
Epoch 10/10
1837/1837 - 42s - loss: 0.0100 - accuracy: 0.9980 - val_loss: 0.0072 - val_accuracy: 0.9984 - 42s/epoch - 23ms/step
7345/7345 [==============================] - 14s 2ms/step
Test Accuracy: 0.9984
Precision: 1.0000
Recall: 0.9967
ROC-AUC: 0.9984
```

Figure 24:Accuracy DoS Attack For CNN

2.Fuzzy Attack

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_6 (Conv1D)           (None, 8, 64)             256

 max_pooling1d_6 (MaxPoolin  (None, 4, 64)             0
 g1D)

 dropout_9 (Dropout)         (None, 4, 64)             0

 conv1d_7 (Conv1D)           (None, 2, 128)            24704

 max_pooling1d_7 (MaxPoolin  (None, 1, 128)            0
 g1D)

 dropout_10 (Dropout)        (None, 1, 128)            0

 flatten_3 (Flatten)         (None, 128)               0

 dense_6 (Dense)             (None, 128)               16512

 dropout_11 (Dropout)        (None, 128)               0

 dense_7 (Dense)             (None, 1)                 129

=================================================================
Total params: 41601 (162.50 KB)
Trainable params: 41601 (162.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
3075/3075 - 44s - loss: 0.0455 - accuracy: 0.9850 - val_loss: 0.0141 - val_accuracy: 0.9965 - 44s/epoch - 14ms/step
Epoch 2/10
3075/3075 - 46s - loss: 0.0173 - accuracy: 0.9950 - val_loss: 0.0134 - val_accuracy: 0.9972 - 46s/epoch - 15ms/step
Epoch 3/10
3075/3075 - 38s - loss: 0.0129 - accuracy: 0.9964 - val_loss: 0.0190 - val_accuracy: 0.9966 - 38s/epoch - 12ms/step
Epoch 4/10
3075/3075 - 40s - loss: 0.0114 - accuracy: 0.9968 - val_loss: 0.0229 - val_accuracy: 0.9948 - 40s/epoch - 13ms/step
Epoch 5/10
3075/3075 - 40s - loss: 0.0104 - accuracy: 0.9971 - val_loss: 0.0128 - val_accuracy: 0.9974 - 40s/epoch - 13ms/step
Epoch 6/10
3075/3075 - 38s - loss: 0.0099 - accuracy: 0.9973 - val_loss: 0.0207 - val_accuracy: 0.9962 - 38s/epoch - 12ms/step
Epoch 7/10
3075/3075 - 41s - loss: 0.0092 - accuracy: 0.9975 - val_loss: 0.0201 - val_accuracy: 0.9952 - 41s/epoch - 13ms/step
Epoch 8/10
3075/3075 - 40s - loss: 0.0090 - accuracy: 0.9976 - val_loss: 0.0396 - val_accuracy: 0.9924 - 40s/epoch - 13ms/step
Epoch 9/10
3075/3075 - 38s - loss: 0.0087 - accuracy: 0.9976 - val_loss: 0.0271 - val_accuracy: 0.9936 - 38s/epoch - 12ms/step
Epoch 10/10
3075/3075 - 39s - loss: 0.0082 - accuracy: 0.9977 - val_loss: 0.0176 - val_accuracy: 0.9969 - 39s/epoch - 13ms/step
6149/6149 [==============================] - 12s 2ms/step
Test Accuracy: 0.9969
Precision: 0.9985
Recall: 0.9953
ROC-AUC: 0.9969
```

Figure 25: Accuracy Fuzzy Attack For CNN

visualization methods, To ensure that my model is not overfitting

Figure 26: Visualization Fuzzy For CNN

3.Gear Attack

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_8 (Conv1D)           (None, 8, 64)             256

 max_pooling1d_8 (MaxPoolin  (None, 4, 64)             0
 g1D)

 dropout_12 (Dropout)        (None, 4, 64)             0

 conv1d_9 (Conv1D)           (None, 2, 128)            24704

 max_pooling1d_9 (MaxPoolin  (None, 1, 128)            0
 g1D)

 dropout_13 (Dropout)        (None, 1, 128)            0

 flatten_4 (Flatten)         (None, 128)               0

 dense_8 (Dense)             (None, 128)               16512

 dropout_14 (Dropout)        (None, 128)               0

 dense_9 (Dense)             (None, 1)                 129

=================================================================
Total params: 41601 (162.50 KB)
Trainable params: 41601 (162.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1867/1867 - 40s - loss: 0.0305 - accuracy: 0.9920 - val_loss: 0.0088 - val_accuracy: 0.9959 - 40s/epoch - 21ms/step
Epoch 2/10
1867/1867 - 38s - loss: 0.0198 - accuracy: 0.9947 - val_loss: 0.0078 - val_accuracy: 0.9958 - 38s/epoch - 21ms/step
Epoch 3/10
1867/1867 - 41s - loss: 0.0194 - accuracy: 0.9948 - val_loss: 0.0081 - val_accuracy: 0.9958 - 41s/epoch - 22ms/step
Epoch 4/10
1867/1867 - 42s - loss: 0.0191 - accuracy: 0.9948 - val_loss: 0.0087 - val_accuracy: 0.9959 - 42s/epoch - 23ms/step
Epoch 5/10
1867/1867 - 40s - loss: 0.0189 - accuracy: 0.9949 - val_loss: 0.0109 - val_accuracy: 0.9955 - 40s/epoch - 21ms/step
Epoch 6/10
1867/1867 - 38s - loss: 0.0188 - accuracy: 0.9949 - val_loss: 0.0093 - val_accuracy: 0.9960 - 38s/epoch - 20ms/step
Epoch 7/10
1867/1867 - 38s - loss: 0.0187 - accuracy: 0.9950 - val_loss: 0.0105 - val_accuracy: 0.9958 - 38s/epoch - 20ms/step
Epoch 8/10
1867/1867 - 40s - loss: 0.0188 - accuracy: 0.9949 - val_loss: 0.0096 - val_accuracy: 0.9957 - 40s/epoch - 22ms/step
Epoch 9/10
1867/1867 - 37s - loss: 0.0185 - accuracy: 0.9949 - val_loss: 0.0100 - val_accuracy: 0.9957 - 37s/epoch - 20ms/step
Epoch 10/10
1867/1867 - 40s - loss: 0.0186 - accuracy: 0.9949 - val_loss: 0.0103 - val_accuracy: 0.9956 - 40s/epoch - 21ms/step
7466/7466 [==============================] - 14s 2ms/step
Test Accuracy: 0.9956
Precision: 1.0000
Recall: 0.9911
ROC-AUC: 0.9956
```

Figure 27: Accuracy Gear Attack For CNN

4.RPM Attack

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_10 (Conv1D)          (None, 9, 64)             256

 max_pooling1d_10 (MaxPooli  (None, 4, 64)             0
 ng1D)

 dropout_15 (Dropout)        (None, 4, 64)             0

 conv1d_11 (Conv1D)          (None, 2, 128)            24704

 max_pooling1d_11 (MaxPooli  (None, 1, 128)            0
 ng1D)

 dropout_16 (Dropout)        (None, 1, 128)            0

 flatten_5 (Flatten)         (None, 128)               0

 dense_10 (Dense)            (None, 128)               16512

 dropout_17 (Dropout)        (None, 128)               0

 dense_11 (Dense)            (None, 1)                 129

=================================================================
Total params: 41601 (162.50 KB)
Trainable params: 41601 (162.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1900/1900 - 42s - loss: 0.2815 - accuracy: 0.9019 - val_loss: 0.1216 - val_accuracy: 0.9567 - 42s/epoch - 22ms/step
Epoch 2/10
1900/1900 - 40s - loss: 0.1568 - accuracy: 0.9448 - val_loss: 0.0191 - val_accuracy: 0.9924 - 40s/epoch - 21ms/step
Epoch 3/10
1900/1900 - 38s - loss: 0.1543 - accuracy: 0.9463 - val_loss: 0.0262 - val_accuracy: 0.9925 - 38s/epoch - 20ms/step
Epoch 4/10
1900/1900 - 40s - loss: 0.1413 - accuracy: 0.9506 - val_loss: 0.0203 - val_accuracy: 0.9915 - 40s/epoch - 21ms/step
Epoch 5/10
1900/1900 - 38s - loss: 0.1392 - accuracy: 0.9507 - val_loss: 0.0168 - val_accuracy: 0.9953 - 38s/epoch - 20ms/step
Epoch 6/10
1900/1900 - 43s - loss: 0.1514 - accuracy: 0.9481 - val_loss: 0.0112 - val_accuracy: 0.9936 - 43s/epoch - 23ms/step
Epoch 7/10
1900/1900 - 40s - loss: 0.1462 - accuracy: 0.9515 - val_loss: 0.0083 - val_accuracy: 0.9979 - 40s/epoch - 21ms/step
Epoch 8/10
1900/1900 - 49s - loss: 0.1408 - accuracy: 0.9518 - val_loss: 0.0058 - val_accuracy: 0.9980 - 49s/epoch - 26ms/step
Epoch 9/10
1900/1900 - 40s - loss: 0.1397 - accuracy: 0.9560 - val_loss: 0.0121 - val_accuracy: 0.9939 - 40s/epoch - 21ms/step
Epoch 10/10
1900/1900 - 39s - loss: 0.1341 - accuracy: 0.9510 - val_loss: 0.0112 - val_accuracy: 0.9931 - 39s/epoch - 21ms/step
7598/7598 [==============================] - 15s 2ms/step
Test Accuracy: 0.9931
Precision: 1.0000
Recall: 0.9931
ROC-AUC: 0.9966
```

Figure 28: Accuracy RPM Attack For CNN

8.2.9.2.Code For Merge Four Dataset

```
merged_data = pd.concat([ DoS, Fuzzy, gear, RPM ], ignore_index=True)
df = merged_data.sample(frac=1).reset_index(drop=True)
df['flag'].replace(['R','T'], [[0],[1]], inplace=True)
df['flag'].value_counts(normalize = True)
print(df.duplicated().sum())
df.drop_duplicates(inplace=True)
df = df.dropna()
df['flag'] = df['flag'].astype(int)
# Define columns to convert from object to integer
columns_to_convert = ['Timestamp', 'CAN_ID', 'DLC', 'DATA[0]', 'DATA[1]', 'DATA[2]', 'DATA[3]', 'DATA[4]', 'DATA[5]', 'DATA[6]', 'DATA[7]']

# Convert each column from object to integer using a for loop
for column in columns_to_convert:
    df[column] = df[column].apply(lambda x: int(x, 16) if isinstance(x, str) else x)
df = df.drop('Timestamp', axis=1)
encode=LabelEncoder()
df["type"]=encode.fit_transform(df["type"])
# Class distribution
# Define colors for each attack type
colors = ['blue', 'orange', 'green', 'red']
# Count the occurrences of each type of attack
attack_counts = df['type'].value_counts()
# Undersample the majority class (normal messages) to balance the dataset
normal_messages = df[df['flag'] == 0]
injected_messages = df[df['flag'] == 1]

# Downsample the majority class to match the number of samples in the minority class (injected messages)
normal_downsampled = resample(normal_messages, replace=False, n_samples=len(injected_messages), random_state=42)

# Combine the downsampled normal messages with the injected messages
undersampled_df = pd.concat([normal_downsampled, injected_messages])
sdf = undersampled_df.sample(frac=1).reset_index(drop=True)
display(sdf)

```

Figure 29: Code For Merge Four Dataset For CNN

```python
# Preprocess the balanced data
X = sdf.drop(sdf.columns[-1], axis=1)  # Drop the target column from the features
y = sdf[sdf.columns[-1]].values  # Assign the target column to y

# Convert string columns to numerical format (if any)
X = X.apply(pd.to_numeric, errors='coerce')

# Split the balanced data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalization (only for X)
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for CNN input
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1],1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1],1)

classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=y_train)
class_weights = dict(enumerate(class_weights))

model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=(x_train.shape[1], 1)))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Conv1D(filters=128, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Add dropout layer
model.add(Dense(1, activation='sigmoid'))

# Print the model summary
model.summary()

# Compile the model
loss = 'binary_crossentropy'
optim = 'Adam'
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optim, metrics=metrics)

# Adding noise to training data
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, -1.0, 1.0)

# Training
batch_size = 512
epochs = 10
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True, verbose=2, class_weight=class_weight
s)
# Evaluate on the test set
y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Calculate additional metrics
accuracy = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)[1]
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"ROC-AUC: {roc_auc:.4f}")

```

Figure 30: Code For Merge Four Dataset For CNN 2

1.ACCURACY:

```
Epoch 1/10
7286/7286 - 58s - 8ms/step - accuracy: 0.9903 - loss: 0.0299 - val_accuracy: 0.9987 - val_loss: 0.0040
Epoch 2/10
7286/7286 - 56s - 8ms/step - accuracy: 0.9976 - loss: 0.0083 - val_accuracy: 0.9996 - val_loss: 0.0015
Epoch 3/10
7286/7286 - 51s - 7ms/step - accuracy: 0.9982 - loss: 0.0065 - val_accuracy: 0.9997 - val_loss: 0.0014
Epoch 4/10
7286/7286 - 50s - 7ms/step - accuracy: 0.9984 - loss: 0.0057 - val_accuracy: 0.9997 - val_loss: 0.0011
Epoch 5/10
7286/7286 - 49s - 7ms/step - accuracy: 0.9985 - loss: 0.0053 - val_accuracy: 0.9997 - val_loss: 0.0012
Epoch 6/10
7286/7286 - 50s - 7ms/step - accuracy: 0.9986 - loss: 0.0048 - val_accuracy: 0.9997 - val_loss: 0.0012
Epoch 7/10
7286/7286 - 49s - 7ms/step - accuracy: 0.9987 - loss: 0.0046 - val_accuracy: 0.9998 - val_loss: 9.8340e-04
Epoch 8/10
7286/7286 - 49s - 7ms/step - accuracy: 0.9988 - loss: 0.0043 - val_accuracy: 0.9998 - val_loss: 9.4912e-04
Epoch 9/10
7286/7286 - 49s - 7ms/step - accuracy: 0.9989 - loss: 0.0039 - val_accuracy: 0.9998 - val_loss: 8.2949e-04
Epoch 10/10
7286/7286 - 49s - 7ms/step - accuracy: 0.9989 - loss: 0.0040 - val_accuracy: 0.9998 - val_loss: 6.8576e-04
29144/29144 ━━━━━━━━━━━━━━━━━━━━ 34s 1ms/step
Test Accuracy: 0.9998
Precision: 1.0000
Recall: 0.9997
ROC-AUC: 0.9998
```

Figure 31: Accuracy Merge Data For CNN

8.3. Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) that is particularly adept at processing and predicting sequential data. Unlike traditional RNNs, LSTMs have the ability to selectively remember and forget information, allowing them to maintain long-term dependencies and avoid the vanishing gradient problem that can plague standard RNNs.

At the core of an LSTM are specialized memory cells that can store and retrieve information over long periods of time. These cells are equipped with gates that control the flow of information, enabling the LSTM to decide what to remember, what to forget, and what to output at each time step. This unique architecture makes LSTMs highly effective in a wide range of applications, from natural language processing and speech recognition to time series forecasting and machine translation.

8.3.1. What is LSTM?

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that is particularly well-suited for processing and predicting sequential data, such as text, audio, or time series data. Unlike traditional RNNs, which can suffer from the vanishing gradient problem, LSTMs are designed to selectively remember and forget information, allowing them to capture long-term dependencies in data more effectively.

At the core of an LSTM network are special units called LSTM cells, which are composed of several gates that control the flow of information. These gates include the forget gate, input gate, and output gate, which work together to decide what information to keep, what to add, and what to output. This unique structure enables LSTMs to maintain a "memory" of past inputs, allowing them to make more accurate predictions and understand the context of the data better than traditional RNNs.

8.3.2. History and Development of LSTM

The Long Short-Term Memory (LSTM) neural network architecture was first introduced in the late 1990s by Sepp Hochreiter and Jürgen Schmidhuber. LSTM was developed as a solution to the vanishing gradient problem that plagued earlier recurrent neural network (RNN) models, which struggled to learn long-term dependencies in sequential data. The key innovation of LSTM was the introduction of memory cells and gate functions, which allowed the network to selectively remember and forget information as it processed sequential inputs.

Over the years, LSTM has undergone various refinements and improvements. Researchers have explored different gate structures, such as the gated recurrent unit (GRU), and have developed variants of LSTM that are tailored for specific tasks or domains. Additionally, the widespread adoption of deep learning has led to the integration of LSTM into larger deep neural network architectures, further expanding its capabilities and applications.

The success of LSTM can be attributed to its ability to effectively model long-term dependencies in sequential data, making it a valuable tool for a wide range of applications, including natural language processing, speech recognition, time series forecasting, and machine translation. As deep learning continues to evolve, LSTM remains a fundamental building block in the field of recurrent neural networks, with ongoing research exploring new ways to enhance its performance and expand its capabilities.

8.3.3.Key Components of LSTM

1-Forget Gate

The forget gate is a critical component of the LSTM architecture. It determines what information from the previous hidden state and current input should be retained or forgotten. This selective memory management allows the LSTM to focus on the most relevant information, improving its ability to learn and remember long-term dependencies in sequence data.

2-Input Gate

The input gate controls what new information from the current input and previous hidden state will be added to the cell state. It decides which values in the cell state will be updated, allowing the LSTM to selectively update its internal memory with relevant data from the input sequence.

3-Output Gate

The output gate determines what information from the current input and previous hidden state will be used to produce the output for the current time step. It filters the cell state to produce the hidden state, which can then be used for downstream tasks or passed to the next time step of the LSTM.

4-Cell State

The cell state is the long-term memory of the LSTM, carrying information from one time step to the next. It is a crucial component that allows the LSTM to maintain and update its internal state, enabling it to learn and remember long-term dependencies in the input sequence.

8.3.4.How LSTM Works

1-Sequence Input

LSTM models take a sequence of inputs, such as words in a sentence or frames in a video. These inputs are processed one at a time, with the LSTM unit maintaining an internal memory state that is updated with each new input.

2-Gating Mechanism

The key to LSTM's success is its gating mechanism, which consists of several gates that control the flow of information into and out of the cell state. These gates decide what

information from the current input and previous hidden state should be remembered, forgotten, or added to the cell state.

3-Cell State Updates

The cell state acts as the LSTM's long-term memory, allowing it to capture and remember important information from the input sequence. The gates carefully modify the cell state, selectively remembering or forgetting information as needed to make accurate predictions or decisions.

4-Output Generation

Based on the updated cell state and the current input, the LSTM generates an output, which can be used for tasks like language modeling, machine translation, or speech recognition. The output is influenced by the gating mechanism and the information stored in the cell state.

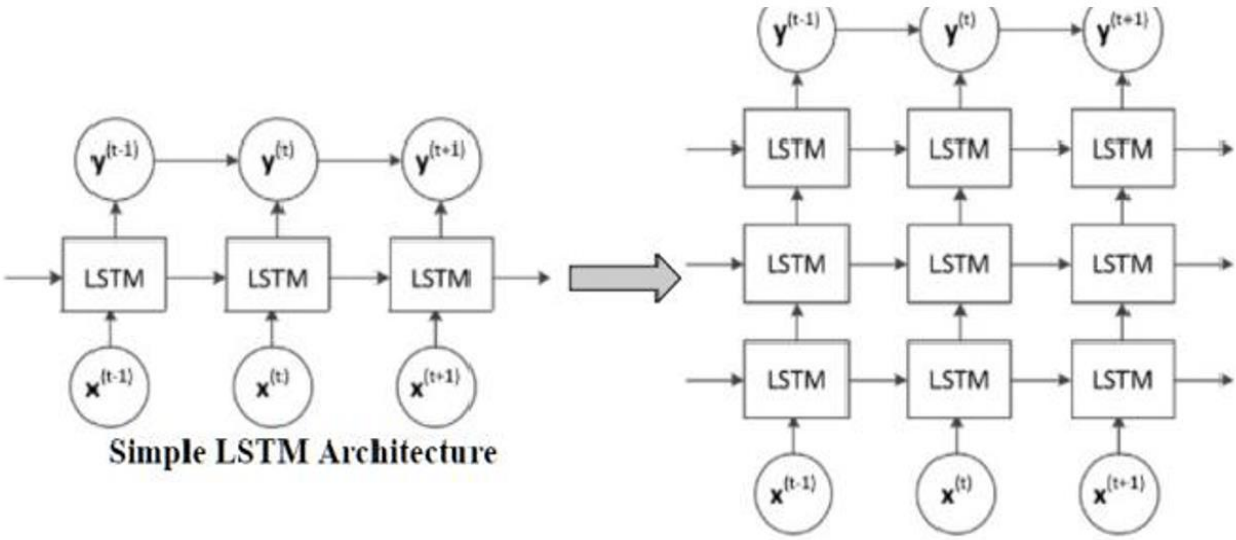


Figure 32: How LSTM Works

8.3.5.Advantages of LSTM

1-Superior Memory Retention

LSTM models excel at retaining relevant information over long sequences, thanks to their unique gating mechanism. This allows them to effectively capture and utilize long-term dependencies, which is crucial for tasks like language modeling, machine translation, and speech recognition.

2-Ability to Handle Vanishing Gradients

The vanishing gradient problem, which can cause traditional recurrent neural networks to struggle, is effectively mitigated by LSTM's architecture. LSTM's gates and cell states help preserve gradients during backpropagation, enabling the model to learn long-term dependencies.

3-Flexibility and Adaptability

LSTM models can be tailored to a wide range of tasks and domains by adjusting the network architecture and hyperparameters. This flexibility allows LSTM to be applied to diverse problems, from language processing to time series forecasting and beyond.

8.3.6.Applications of LSTM

Natural Language Processing (NLP)

LSTM models have become a staple in natural language processing tasks such as language modeling, text generation, machine translation, and sentiment analysis. Their ability to capture long-term dependencies in sequential data makes them particularly well-suited for understanding and generating human language, which often exhibits complex temporal and contextual patterns.

Speech Recognition

LSTM networks have been widely used in automatic speech recognition (ASR) systems, where they excel at modeling the temporal dynamics of speech signals. By learning to predict the next phoneme or word in a sequence, LSTM-based ASR models can accurately transcribe spoken language, even in the presence of noise or variations in accents and speech patterns.

8.3.7.Limitations and Challenges of LSTM

While Long Short-Term Memory (LSTM) networks have proven to be highly effective in various applications, they do face certain limitations and challenges that need to be addressed. One key limitation is their increased complexity compared to simpler recurrent neural network (RNN) architectures. The additional gates and parameters in LSTM models can make them more computationally expensive and slower to train, especially on large-scale datasets.

Another challenge is the difficulty in interpreting the internal workings of LSTM models, as their memory cells and gates can be difficult to interpret and understand. This can make it challenging to diagnose and troubleshoot issues with LSTM models, as well as to explain their decision-making processes to stakeholders or end-users.

LSTM models can also be sensitive to the quality and quantity of training data, and may struggle to generalize well to new, unseen data that deviates significantly from the training distribution. This can be a particular challenge in applications where the input data is highly variable or noisy.

Additionally, LSTM models can be prone to overfitting, which can lead to poor performance on new, unseen data. Careful regularization techniques and monitoring of model performance during training are often required to mitigate this issue.

8.3.8.LSTM Variants and Improvements

Since its introduction, the LSTM architecture has undergone various modifications and improvements to enhance its performance and expand its applications. These LSTM variants have aimed to address specific limitations or challenges faced by the original LSTM model, making it more efficient, versatile, and capable of handling complex tasks.

One notable LSTM variant is the Bidirectional LSTM (Bi-LSTM), which incorporates information from both the forward and backward directions of the input sequence. This allows the model to capture more contextual information and improve its understanding of the data, particularly in tasks such as language modeling and sentiment analysis. Another variant, the Convolutional LSTM (ConvLSTM), integrates convolutional layers into the LSTM architecture, enabling the model to effectively process spatial and temporal data, making it suitable for applications like video processing and weather forecasting.

Additionally, researchers have explored the incorporation of attention mechanisms into LSTM models, resulting in the Attention-based LSTM (Att-LSTM). This approach allows the model to focus on the most relevant parts of the input sequence, enhancing its performance in tasks like machine translation and text summarization.

8.3.9. LSTM Code

1. Importing Libraries

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import precision_score, recall_score, roc_auc_score
from sklearn.utils import resample
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
from keras.utils import to_categorical
```

Figure 33: Importing Libraries For LSTM

This section imports the necessary libraries for data manipulation (Pandas and NumPy), data splitting and scaling (scikit-learn), and building and evaluating the deep learning model (TensorFlow/Keras). precision_score, recall_score, and roc_auc_score are used for evaluating the model's performance.

2.Data Loading and Preprocessing

```python

# Define column names
columns = ["Timestamp", "CAN_ID", "DLC", "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7", "Flag"]

# Load your dataset again, skipping the first row
data = pd.read_csv("/content/DoS_dataset.csv", names=columns, skiprows=1, low_memory=False)

# Convert hexadecimal values to integers
data['CAN_ID'] = data['CAN_ID'].apply(lambda x: int(x, 16))
data['DLC'] = data['DLC'].astype('int64')

# Define function to handle hexadecimal values
def b16(x):
    if isinstance(x, str):
        return int(x, 16) if all(c in '0123456789ABCDEFabcdef' for c in x) else None
    else:
        return None

# Apply conversion function to hexadecimal columns
for col in ['D0', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']:
    data[col] = data[col].apply(b16)

# Drop rows with missing values
data = data.dropna()
data = data.drop_duplicates()
data.fillna(0, inplace=True)

# Convert float columns to integers
data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']] = data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']].astype(int)

# Convert Flag to int (if applicable)
data['Flag'] = data['Flag'].apply(lambda x: 1 if x == 'R' else 0)

# Drop the 'Timestamp' column
data = data.drop('Timestamp', axis=1)

```

Figure 34:Data Loading and Preprocessing

Define Column Names: Specifies the column names for the dataset.
Load Dataset: Reads the dataset from the specified path, skipping the first row.
Hexadecimal to Integer Conversion: Converts the hexadecimal values in the CAN_ID column to integers.
Conversion Function: Defines a function b16 to handle conversion of hexadecimal string values to integers.
Apply Conversion: Applies the b16 function to the data columns D0 to D7.
Handling Missing Values: Drops rows with missing values and duplicates, and fills any remaining NaN values with 0.
Data Type Conversion: Converts columns D1 to D7 to integer type.
Convert Flag Column: Converts the Flag column values to binary (1 for 'R', 0 for others).
Drop Timestamp Column: Drops the Timestamp column as it is not needed for the analysis.

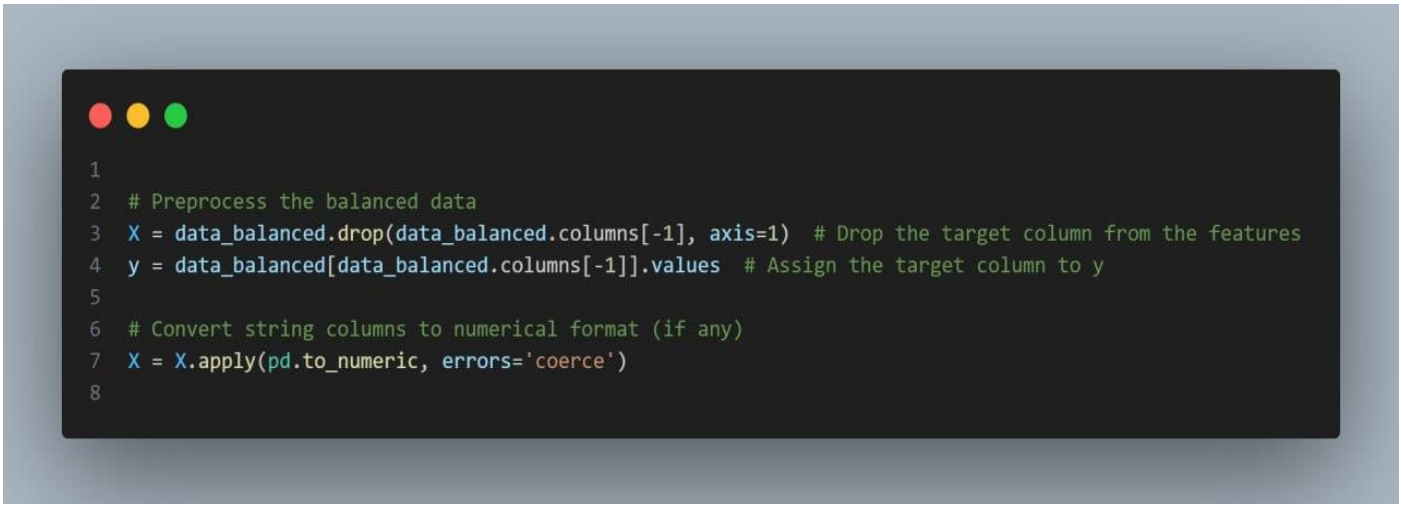
3.Data Balancing

```
# Separate normal and anomaly data
data_normal = data[data['Flag'] == 0]
data_anomaly = data[data['Flag'] == 1]

# Check which data is smaller
if len(data_normal) < len(data_anomaly):
    # Undersample anomaly data to match normal data size
    data_anomaly_sampled = data_anomaly.sample(n=len(data_normal), random_state=42)
    data_balanced = pd.concat([data_normal, data_anomaly_sampled])
else:
    # Undersample normal data to match anomaly data size
    data_normal_sampled = data_normal.sample(n=len(data_anomaly), random_state=42)
    data_balanced = pd.concat([data_normal_sampled, data_anomaly])

```

Figure 35: Data Balancing

Separate Normal and Anomaly Data: Splits the data into two subsets based on the Flag column.
Balancing Data: Checks which class (normal or anomaly) has fewer samples. It then undersamples the larger class to match the size of the smaller class to create a balanced dataset.
Shuffle Data: Shuffles the balanced dataset and resets the index.
Data Inspection: Prints the data types of the balanced dataset and the first few rows for inspection

4. Feature and Target Separation



```
# Preprocess the balanced data
X = data_balanced.drop(data_balanced.columns[-1], axis=1)  # Drop the target column from the features
y = data_balanced[data_balanced.columns[-1]].values  # Assign the target column to y

# Convert string columns to numerical format (if any)
X = X.apply(pd.to_numeric, errors='coerce')
```

Figure 36: Feature and Target Separation

Feature-Target Split: Separates the features (X) from the target variable (y).
Convert Strings to Numbers: Converts any remaining string columns in X to numeric format, handling any errors by setting non-convertible entries to NaN.

5. Train-Test Split and Normalization

```
# Split the balanced data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalization (only for X)
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for LSTM input
x_train = x_train.reshape(x_train.shape[0], 1, x_train.shape[1])
x_test = x_test.reshape(x_test.shape[0], 1, x_test.shape[1])
```

Figure 37: Train-Test Split and Normalization

Train-Test Split: Splits the balanced dataset into training and testing sets, maintaining the class distribution (stratification) with an 80-20 split.
Normalization: Scales the feature data using StandardScaler to have a mean of 0 and a standard deviation of 1.
Reshape Data: Reshapes the feature data to be compatible with the input requirements of a Conv1D model (3D shape: samples, timesteps, features).

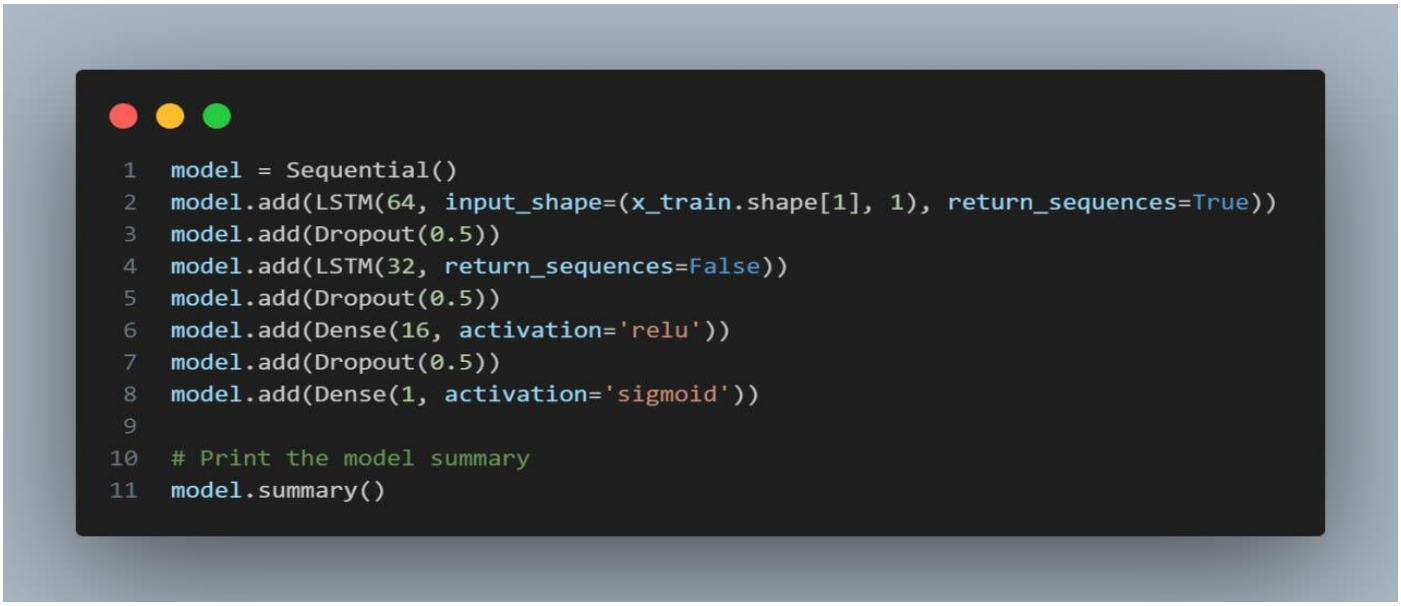
6. Compute Class Weights

```
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=y_train)
class_weights = dict(enumerate(class_weights))
```

Figure 38: Compute Class Weights

Computes class weights to handle class imbalance during model training, ensuring that the model gives appropriate importance to both classes.

7.Model Definition



```
model = Sequential()
model.add(LSTM(64, input_shape=(x_train.shape[1], 1), return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(32, return_sequences=False))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Print the model summary
model.summary()
```

Figure 39: Model Definition For LSTM

Model Definition: Defines a Sequential model for anomaly detection.
Conv1D Layers: Adds two 1D convolutional layers with 64 and 128 filters respectively, each followed by a max pooling layer.
Dropout Layers: Adds dropout layers with a rate of 0.5 after the convolutional layers and the dense layer to prevent overfitting.
Flatten Layer: Flattens the output of the convolutional layers.
Dense Layers: Adds a dense (fully connected) layer with 128 units and a dropout layer, followed by an output layer with a sigmoid activation function for binary classification.
Model Summary: Prints the summary of the model architecture.

8.Model Compilation and Training with Noise

```python
# Compile the model
loss = 'binary_crossentropy'
optim = 'Adam'
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optim, metrics=metrics)

# Adding noise to training data
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, -1.0, 1.0)

# Training
batch_size = 512
epochs = 10
history = model.fit(x_train_noisy, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True, verbose=2, class_weight=class_weights)

```

Figure 40: Model Compilation and Training

Model Compilation: Compiles the model using binary cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric.
Adding Noise: Introduces Gaussian noise to the training data to improve the model's robustness.
Training: Trains the model using the noisy training data with a batch size of 512 and 10 epochs, validating on the test set, and applying class weights.

9. Model Evaluation

```python
# Evaluate on the test set
y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Calculate additional metrics
accuracy = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)[1]
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"ROC-AUC: {roc_auc:.4f}")
```

Figure 41: Model Evaluation

Prediction: Makes predictions on the test set and converts the probabilities to binary class labels.
Evaluation: Computes the accuracy, precision, recall, and ROC-AUC score to evaluate the model's performance.
Print Results: Prints the calculated evaluation metrics.

8.3.9.1 Accuracy

1.Dos Attack

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 1, 128)            71168

 lstm_1 (LSTM)               (None, 128)               131584

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 202881 (792.50 KB)
Trainable params: 202881 (792.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
Epoch 1/10
1434/1434 - 43s - loss: 0.0371 - accuracy: 0.9921 - val_loss: 0.0065 - val_accuracy: 0.9984 - 43s/epoch - 30ms/step
Epoch 2/10
1434/1434 - 36s - loss: 0.0096 - accuracy: 0.9980 - val_loss: 0.0075 - val_accuracy: 0.9984 - 36s/epoch - 25ms/step
Epoch 3/10
1434/1434 - 40s - loss: 0.0093 - accuracy: 0.9980 - val_loss: 0.0063 - val_accuracy: 0.9986 - 40s/epoch - 28ms/step
Epoch 4/10
1434/1434 - 41s - loss: 0.0092 - accuracy: 0.9981 - val_loss: 0.0073 - val_accuracy: 0.9984 - 41s/epoch - 28ms/step
Epoch 5/10
1434/1434 - 40s - loss: 0.0092 - accuracy: 0.9981 - val_loss: 0.0091 - val_accuracy: 0.9982 - 40s/epoch - 28ms/step
Epoch 6/10
1434/1434 - 40s - loss: 0.0091 - accuracy: 0.9981 - val_loss: 0.0074 - val_accuracy: 0.9984 - 40s/epoch - 28ms/step
Epoch 7/10
1434/1434 - 37s - loss: 0.0092 - accuracy: 0.9981 - val_loss: 0.0072 - val_accuracy: 0.9984 - 37s/epoch - 26ms/step
Epoch 8/10
1434/1434 - 37s - loss: 0.0091 - accuracy: 0.9981 - val_loss: 0.0078 - val_accuracy: 0.9984 - 37s/epoch - 26ms/step
Epoch 9/10
1434/1434 - 40s - loss: 0.0091 - accuracy: 0.9981 - val_loss: 0.0070 - val_accuracy: 0.9984 - 40s/epoch - 28ms/step
Epoch 10/10
1434/1434 - 36s - loss: 0.0091 - accuracy: 0.9981 - val_loss: 0.0085 - val_accuracy: 0.9982 - 36s/epoch - 25ms/step
5734/5734 [==============================] - 17s 3ms/step
Test Accuracy: 0.9982
Precision: 1.0000
Recall: 0.9963
ROC-AUC: 0.9982
```

Figure 42: Accuracy DoS Attack For LSTM

visualization methods, To ensure that my model is not overfitting

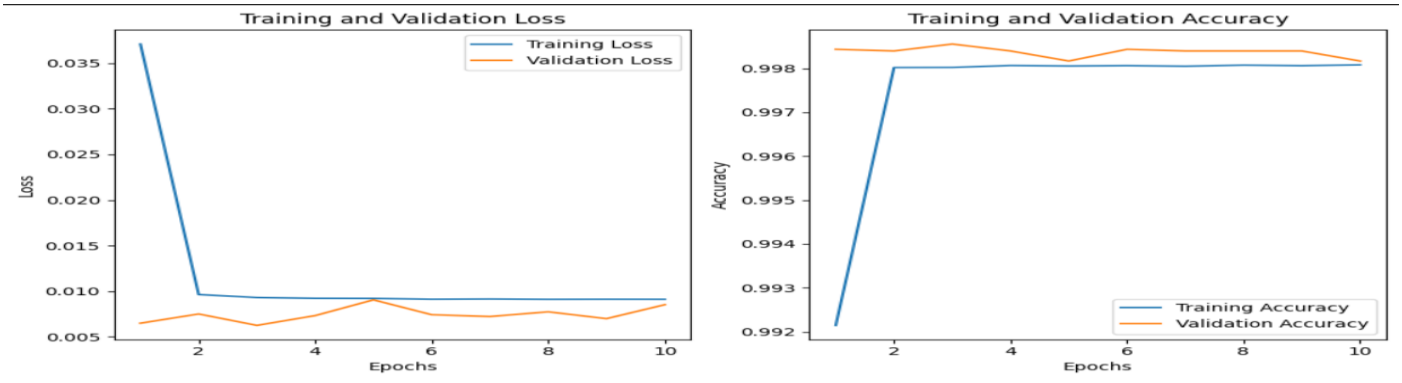


Figure 43: Visualization Dos For LSTM

2.Fuzzy Attack

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 1, 128)            71168

 lstm_3 (LSTM)               (None, 128)               131584

 dense_1 (Dense)             (None, 1)                 129

=================================================================
Total params: 202881 (792.50 KB)
Trainable params: 202881 (792.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
Epoch 1/10
1538/1538 - 48s - loss: 0.0427 - accuracy: 0.9906 - val_loss: 0.0073 - val_accuracy: 0.9988 - 48s/epoch - 31ms/step
Epoch 2/10
1538/1538 - 50s - loss: 0.0038 - accuracy: 0.9991 - val_loss: 0.0030 - val_accuracy: 0.9993 - 50s/epoch - 33ms/step
Epoch 3/10
1538/1538 - 48s - loss: 0.0023 - accuracy: 0.9995 - val_loss: 0.0070 - val_accuracy: 0.9994 - 48s/epoch - 31ms/step
Epoch 4/10
1538/1538 - 48s - loss: 0.0017 - accuracy: 0.9996 - val_loss: 0.1557 - val_accuracy: 0.9697 - 48s/epoch - 31ms/step
Epoch 5/10
1538/1538 - 52s - loss: 0.0015 - accuracy: 0.9996 - val_loss: 0.1782 - val_accuracy: 0.9725 - 52s/epoch - 34ms/step
Epoch 6/10
1538/1538 - 46s - loss: 0.0013 - accuracy: 0.9997 - val_loss: 0.2637 - val_accuracy: 0.9655 - 46s/epoch - 30ms/step
Epoch 7/10
1538/1538 - 41s - loss: 0.0012 - accuracy: 0.9997 - val_loss: 0.2808 - val_accuracy: 0.9708 - 41s/epoch - 27ms/step
Epoch 8/10
1538/1538 - 50s - loss: 0.0011 - accuracy: 0.9997 - val_loss: 0.3197 - val_accuracy: 0.9724 - 50s/epoch - 33ms/step
Epoch 9/10
1538/1538 - 46s - loss: 0.0010 - accuracy: 0.9997 - val_loss: 0.2615 - val_accuracy: 0.9713 - 46s/epoch - 30ms/step
Epoch 10/10
1538/1538 - 45s - loss: 9.2706e-04 - accuracy: 0.9998 - val_loss: 0.4141 - val_accuracy: 0.9691 - 45s/epoch - 29ms/step
6149/6149 [==============================] - 36s 6ms/step
Test Accuracy: 0.9691
```

Figure 44 Accuracy Fuzzy Attack For LSTM

3.Gear Attack

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 1, 128)            71168

 lstm_5 (LSTM)               (None, 128)               131584

 dense_2 (Dense)             (None, 1)                 129

=================================================================
Total params: 202881 (792.50 KB)
Trainable params: 202881 (792.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
Epoch 1/10
1867/1867 - 59s - loss: 0.0366 - accuracy: 0.9931 - val_loss: 0.0095 - val_accuracy: 0.9955 - 59s/epoch - 32ms/step
Epoch 2/10
1867/1867 - 55s - loss: 0.0165 - accuracy: 0.9953 - val_loss: 0.0086 - val_accuracy: 0.9959 - 55s/epoch - 30ms/step
Epoch 3/10
1867/1867 - 56s - loss: 0.0164 - accuracy: 0.9953 - val_loss: 0.0084 - val_accuracy: 0.9960 - 56s/epoch - 30ms/step
Epoch 4/10
1867/1867 - 61s - loss: 0.0163 - accuracy: 0.9954 - val_loss: 0.0089 - val_accuracy: 0.9960 - 61s/epoch - 32ms/step
Epoch 5/10
1867/1867 - 56s - loss: 0.0162 - accuracy: 0.9953 - val_loss: 0.0111 - val_accuracy: 0.9954 - 56s/epoch - 30ms/step
Epoch 6/10
1867/1867 - 54s - loss: 0.0163 - accuracy: 0.9953 - val_loss: 0.0095 - val_accuracy: 0.9957 - 54s/epoch - 29ms/step
Epoch 7/10
1867/1867 - 62s - loss: 0.0162 - accuracy: 0.9954 - val_loss: 0.0093 - val_accuracy: 0.9957 - 62s/epoch - 33ms/step
Epoch 8/10
1867/1867 - 53s - loss: 0.0162 - accuracy: 0.9953 - val_loss: 0.0105 - val_accuracy: 0.9956 - 53s/epoch - 29ms/step
Epoch 9/10
1867/1867 - 62s - loss: 0.0162 - accuracy: 0.9954 - val_loss: 0.0094 - val_accuracy: 0.9957 - 62s/epoch - 33ms/step
Epoch 10/10
1867/1867 - 53s - loss: 0.0162 - accuracy: 0.9954 - val_loss: 0.0101 - val_accuracy: 0.9956 - 53s/epoch - 28ms/step
7466/7466 [==============================] - 28s 4ms/step
Test Accuracy: 0.9956
```

Figure 45: Accuracy Gear Attack For LSTM

4.RPM

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_6 (LSTM)               (None, 11, 64)            16896

 dropout_3 (Dropout)         (None, 11, 64)            0

 lstm_7 (LSTM)               (None, 32)                12416

 dropout_4 (Dropout)         (None, 32)                0

 dense_4 (Dense)             (None, 16)                528

 dropout_5 (Dropout)         (None, 16)                0

 dense_5 (Dense)             (None, 1)                 17

=================================================================
Total params: 29857 (116.63 KB)
Trainable params: 29857 (116.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
1900/1900 - 146s - loss: 0.5128 - accuracy: 0.5989 - val_loss: 0.2623 - val_accuracy: 0.9328 - 146s/epoch - 77ms/step
Epoch 2/10
1900/1900 - 138s - loss: 0.2345 - accuracy: 0.9020 - val_loss: 0.2460 - val_accuracy: 0.9300 - 138s/epoch - 73ms/step
Epoch 3/10
1900/1900 - 145s - loss: 0.1105 - accuracy: 0.9627 - val_loss: 0.0472 - val_accuracy: 0.9767 - 145s/epoch - 76ms/step
Epoch 4/10
1900/1900 - 145s - loss: 0.0869 - accuracy: 0.9745 - val_loss: 0.0082 - val_accuracy: 0.9976 - 145s/epoch - 76ms/step
Epoch 5/10
1900/1900 - 138s - loss: 0.0668 - accuracy: 0.9791 - val_loss: 0.0331 - val_accuracy: 0.9880 - 138s/epoch - 73ms/step
Epoch 6/10
1900/1900 - 147s - loss: 0.0739 - accuracy: 0.9745 - val_loss: 0.0064 - val_accuracy: 0.9987 - 147s/epoch - 77ms/step
Epoch 7/10
1900/1900 - 146s - loss: 0.0570 - accuracy: 0.9824 - val_loss: 0.0039 - val_accuracy: 0.9982 - 146s/epoch - 77ms/step
Epoch 8/10
1900/1900 - 145s - loss: 0.0524 - accuracy: 0.9830 - val_loss: 0.0013 - val_accuracy: 0.9993 - 145s/epoch - 76ms/step
Epoch 9/10
1900/1900 - 139s - loss: 0.0413 - accuracy: 0.9846 - val_loss: 0.0062 - val_accuracy: 0.9974 - 139s/epoch - 73ms/step
Epoch 10/10
1900/1900 - 139s - loss: 0.0468 - accuracy: 0.9801 - val_loss: 0.0059 - val_accuracy: 0.9964 - 139s/epoch - 73ms/step
7598/7598 [==============================] - 36s 5ms/step
Test Accuracy: 0.9964
```

Figure 46: Accuracy RPM Attack For LSTM

8.3.9.2.Code For Merge Four Dataset

```python
merged_data = pd.concat([ DoS, Fuzzy, gear, RPM ], ignore_index=True)
df = merged_data.sample(frac=1).reset_index(drop=True)
df['flag'].replace(['R','T'], [[0],[1]], inplace=True)
df['flag'].value_counts(normalize = True)
print(df.duplicated().sum())
df.drop_duplicates(inplace=True)
df = df.dropna()
df['flag'] = df['flag'].astype(int)
# Define columns to convert from object to integer
columns_to_convert = ['Timestamp', 'CAN_ID', 'DLC', 'DATA[0]', 'DATA[1]', 'DATA[2]', 'DATA[3]', 'DATA[4]', 'DATA[5]', 'DATA[6]', 'DATA[7]']

# Convert each column from object to integer using a for loop
for column in columns_to_convert:
    df[column] = df[column].apply(lambda x: int(x, 16) if isinstance(x, str) else x)
df = df.drop('Timestamp', axis=1)
encode=LabelEncoder()
df["type"]=encode.fit_transform(df["type"])
# Class distribution
# Define colors for each attack type
colors = ['blue', 'orange', 'green', 'red']
# Count the occurrences of each type of attack
attack_counts = df['type'].value_counts()
# Undersample the majority class (normal messages) to balance the dataset
normal_messages = df[df['flag'] == 0]
injected_messages = df[df['flag'] == 1]

# Downsample the majority class to match the number of samples in the minority class (injected messages)
normal_downsampled = resample(normal_messages, replace=False, n_samples=len(injected_messages), random_state=42)

# Combine the downsampled normal messages with the injected messages
undersampled_df = pd.concat([normal_downsampled, injected_messages])
sdf = undersampled_df.sample(frac=1).reset_index(drop=True)
display(sdf)

```

Figure 47: Code For Merge Four Dataset For LSTM

```python
X = sdf.drop(sdf.columns[-1], axis=1)  # Drop the target column
y = sdf[sdf.columns[-1]].values  # Target column

# Convert string columns to numerical format
X = X.apply(pd.to_numeric, errors='coerce')

# Split the balanced data
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalization
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for LSTM input
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

# Compute class weights
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=y_train)
class_weights = dict(enumerate(class_weights))

# Build the LSTM model
model = Sequential()
model.add(LSTM(64, input_shape=(x_train.shape[1], 1), return_sequences=True))
model.add(Dropout(0.5))
model.add(LSTM(128, return_sequences=False))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# Print the model summary
model.summary()

# Compile the model
loss = 'binary_crossentropy'
optim = 'Adam'
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optim, metrics=metrics)

# Adding noise to training data
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, -1.0, 1.0)

# Training
batch_size = 512
epochs = 10
history = model.fit(x_train_noisy, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True, verbose=2, class_weight=class_weights)

# Evaluate on the test set
y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5).astype(int)

# Calculate additional metrics
accuracy = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)[1]
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

print(f"Test Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"ROC-AUC: {roc_auc:.4f}")

```

Figure 48: Code For Merge Four Dataset For LSTM 2

1. ACCURACY

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_4 (LSTM)               (None, 11, 64)            16896

 dropout_3 (Dropout)         (None, 11, 64)            0

 lstm_5 (LSTM)               (None, 128)               98816

 dropout_4 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 128)               16512

 dropout_5 (Dropout)         (None, 128)               0

 dense_4 (Dense)             (None, 1)                 129

=================================================================
Total params: 132353 (517.00 KB)
Trainable params: 132353 (517.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/10
4278/4278 - 847s - loss: 0.0481 - accuracy: 0.9846 - val_loss: 0.0275 - val_accuracy: 0.9924 - 847s/epoch - 198ms/step
Epoch 2/10
4278/4278 - 868s - loss: 0.0107 - accuracy: 0.9974 - val_loss: 0.0112 - val_accuracy: 0.9969 - 868s/epoch - 203ms/step
Epoch 3/10
4278/4278 - 862s - loss: 0.0085 - accuracy: 0.9980 - val_loss: 0.0120 - val_accuracy: 0.9972 - 862s/epoch - 201ms/step
Epoch 4/10
4278/4278 - 930s - loss: 0.0074 - accuracy: 0.9982 - val_loss: 0.0068 - val_accuracy: 0.9986 - 930s/epoch - 217ms/step
Epoch 5/10
4278/4278 - 860s - loss: 0.0069 - accuracy: 0.9983 - val_loss: 0.0077 - val_accuracy: 0.9987 - 860s/epoch - 201ms/step
Epoch 6/10
4278/4278 - 860s - loss: 0.0066 - accuracy: 0.9984 - val_loss: 0.0356 - val_accuracy: 0.9914 - 860s/epoch - 201ms/step
Epoch 7/10
4278/4278 - 960s - loss: 0.0063 - accuracy: 0.9985 - val_loss: 0.0357 - val_accuracy: 0.9896 - 960s/epoch - 224ms/step
Epoch 8/10
4278/4278 - 1036s - loss: 0.0061 - accuracy: 0.9985 - val_loss: 0.0185 - val_accuracy: 0.9938 - 1036s/epoch - 242ms/step
Epoch 9/10
4278/4278 - 1046s - loss: 0.0060 - accuracy: 0.9986 - val_loss: 0.0263 - val_accuracy: 0.9896 - 1046s/epoch - 244ms/step
Epoch 10/10
4278/4278 - 924s - loss: 0.0060 - accuracy: 0.9986 - val_loss: 0.0085 - val_accuracy: 0.9983 - 924s/epoch - 216ms/step
17110/17110 [==============================] - 212s 12ms/step
Test Accuracy: 0.9983
Precision: 0.9980
Recall: 0.9986
ROC-AUC: 0.9983
```

Figure 49: Accuracy Merge Data For LSTM

visualization methods, To ensure that my model is not overfitting

Figure 50: Accuracy Merge Data For LSTM

8.4. Bidirectional Recurrent Neural Networks (BRNN)

Bidirectional Recurrent Neural Networks (BRNNs) are a powerful class of deep learning models that excel at processing sequential data, such as natural language, speech, and time series. Unlike traditional recurrent neural networks (RNNs) that only process information in a forward direction, BRNNs leverage two separate RNN components to analyze the input sequence in both forward and backward directions. This bidirectional approach allows the model to capture richer contextual information, leading to enhanced performance on a wide range of sequence-to-sequence tasks.

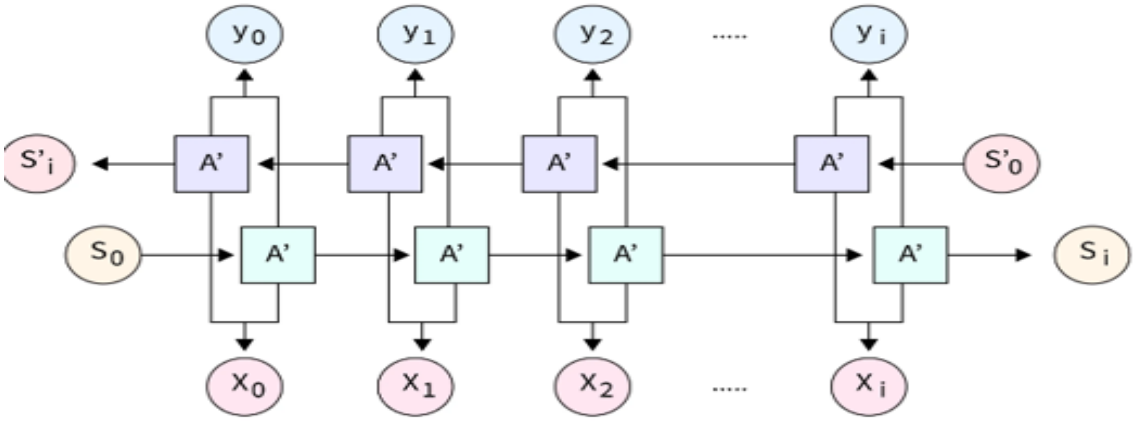


Figure 51: How BRNNs Work

8.4.1.Motivation and Applications of BRNN

Bidirectional Recurrent Neural Networks (BRNNs) have gained significant attention in the field of deep learning due to their ability to capture both forward and backward dependencies in sequential data. This unique architecture allows BRNNs to leverage information from the entire input sequence, rather than just the preceding or subsequent elements, enabling more comprehensive and accurate modeling of complex patterns.

The primary motivation for using BRNNs lies in their ability to improve performance on a wide range of tasks, particularly in areas where the context and dependencies within the sequence are crucial. Some of the key applications of BRNNs include natural language processing (NLP) tasks such as text classification, sentiment analysis, named entity recognition, and machine translation. In these domains, BRNNs excel at capturing the nuanced relationships between words and their surrounding context, leading to better understanding and more accurate predictions.

8.4.2.Architectural Overview of BRNN

A Bidirectional Recurrent Neural Network (BRNN) is an extension of the standard Recurrent Neural Network (RNN) architecture, designed to capture information from both the past and future context of a sequence. In a traditional RNN, the network processes the input sequence in a unidirectional manner, where the hidden state at each time step is influenced only by the current input and the previous hidden state. In contrast, a BRNN consists of two separate RNN layers, one processing the input sequence in the forward direction and the other in the reverse direction. The final output of the BRNN is then a combination of the outputs from both the forward and backward RNN layers, allowing the network to leverage information from both the past and future contexts of the input sequence.

The key architectural component of a BRNN is the bidirectional layer, which is composed of two separate RNN layers, one processing the input sequence in the forward direction and the other in the reverse direction. The forward RNN layer takes the input sequence in its original order and computes the hidden states and outputs, while the backward RNN layer processes the input sequence in the reverse order, generating its own set of hidden states and outputs. The final output of the BRNN is then obtained by concatenating or combining the outputs from both the forward and backward RNN layers, effectively capturing information from both the past and future contexts of the input sequence.

8.4.3.Forward and Backward Passes in BRNN

Bidirectional Input

The input sequence is processed in both the forward and backward directions.

Forward Pass

The forward RNN encodes the input sequence from beginning to end.

Backward Pass

The backward RNN encodes the input sequence from end to beginning.

Concatenation

The forward and backward hidden states are concatenated to produce the final output.

The defining characteristic of a Bidirectional Recurrent Neural Network (BRNN) is the use of two separate RNN modules to process the input sequence in both the forward and backward directions. During the forward pass, the first RNN encodes the input from

the beginning to the end, while the second RNN simultaneously encodes the input from the end to the beginning. The final output is then produced by concatenating the hidden states from both the forward and backward RNNs at each time step.

This bidirectional processing allows the BRNN to capture both past and future context for each element in the input sequence, leading to more powerful and expressive representations. The forward and backward passes can use the same or different RNN architectures, such as vanilla RNNs, LSTMs, or GRUs, depending on the specific requirements of the task.

8.4.4.Training Algorithms for BRNN

Training a bidirectional recurrent neural network (BRNN) requires specialized algorithms that can effectively capture the bidirectional flow of information. Unlike traditional unidirectional RNNs, BRNNs have two separate hidden state sequences, one for the forward pass and one for the backward pass. This added complexity necessitates different training approaches.

A common training algorithm for BRNNs is the backpropagation through time (BPTT) algorithm. BPTT computes the gradients of the loss function with respect to the network parameters by unrolling the recurrent connections and applying the chain rule. This allows the network to learn the dependencies between the forward and backward hidden states, enabling it to effectively model bidirectional relationships in the data.

Another approach is the truncated BPTT algorithm, which limits the number of time steps used in the backpropagation, reducing the memory requirements and allowing for more efficient training of BRNNs, especially on long sequences.

Additionally, reinforcement learning techniques have been explored for training BRNNs, where the network is taught to make decisions based on the bidirectional context, leading to improved performance in tasks such as sequence labeling and machine translation.

8.4.5.Advantages of BRNN over Unidirectional RNNs

Bidirectional Information Flow

Bidirectional Recurrent Neural Networks (BRNNs) have a distinct advantage over their unidirectional counterparts in their ability to process information in both forward and backward directions. This bidirectional flow of information allows BRNNs to capture

contextual cues from the past and future, leading to a more comprehensive understanding of the input sequence.

Enhanced Performance

The bidirectional nature of BRNNs often results in improved performance across a variety of tasks, such as language modeling, speech recognition, and text classification. By leveraging information from both past and future contexts, BRNNs can make more informed and accurate predictions, leading to better overall performance compared to unidirectional RNNs.

Robustness to Noise

BRNNs have been shown to be more robust to noise and missing information in the input data. The backward pass allows the network to compensate for gaps or inconsistencies in the forward sequence, making BRNNs a more reliable choice for applications where the input data may be noisy or incomplete.

8.4.6.Challenges and Limitations of BRNN

Complexity and Computational Cost

Bidirectional Recurrent Neural Networks (BRNNs) are more complex and computationally expensive compared to their unidirectional counterparts. The need to process the input sequence in both forward and backward directions increases the overall computational requirements, which can be a limiting factor, especially for real-time applications or resource-constrained environments.

Dependency on Full Sequence

BRNNs require the entire input sequence to be available before the model can generate the output. This can be a limitation in scenarios where the input data is generated incrementally or in a streaming fashion, as the 0model cannot make predictions until the complete sequence is available.

Lack of Causal Relationships

Unlike unidirectional RNNs, BRNNs do not maintain a strict causal relationship between the input and output sequences. This can be a concern in applications where the order of events or the causal dependencies are crucial, such as in speech recognition or time-series forecasting.

Vanishing/Exploding Gradient Issues

BRNNs, like other types of recurrent neural networks, are susceptible to the vanishing or exploding gradient problem, which can make training the model challenging. This

issue arises due to the recursive nature of the network and the repeated application of the same weight matrices over long sequences.

8.4.7.Variants and Extensions of BRNN

While the basic Bidirectional Recurrent Neural Network (BRNN) architecture has proven to be highly effective in various applications, researchers have proposed numerous variants and extensions to further enhance its capabilities. One notable extension is the Bidirectional Long Short-Term Memory (BLSTM) network, which incorporates LSTM units to better capture long-range dependencies in the input data. BLSTM models have demonstrated superior performance in tasks such as speech recognition, handwriting recognition, and machine translation, where the ability to leverage both past and future context is crucial.

Another variant is the Bidirectional Gated Recurrent Unit (BGRU), which replaces the LSTM units with Gated Recurrent Units (GRUs). GRUs are a simpler and more computationally efficient alternative to LSTMs, making BGRU models a compelling choice for applications with limited computational resources. Bidirectional Transformer models, such as BERT, have also emerged as powerful language models that leverage the BRNN architecture to capture bidirectional dependencies in text data, enabling state-of-the-art performance on a wide range of natural language processing tasks.

8.4.8.BRNN in Natural Language Processing

Bidirectional Recurrent Neural Networks (BRNNs) have become a powerful tool in the field of natural language processing (NLP). In contrast to traditional unidirectional RNNs, BRNNs can learn and process sequential data by integrating information from both the past and the future context. This unique capability makes them particularly well-suited for a variety of NLP tasks, such as language modeling, part-of-speech tagging, named entity recognition, and text summarization.

One of the key advantages of BRNNs in NLP is their ability to capture long-range dependencies in text. By processing the input sequence in both forward and backward directions, BRNNs can learn to understand the semantic and syntactic relationships between words, even when they are separated by large distances. This is particularly important in tasks like sentiment analysis and question answering, where the meaning of a word or phrase can be heavily influenced by the surrounding context.

Moreover, BRNNs can also be effectively combined with other deep learning architectures, such as convolutional neural networks (CNNs) and attention

mechanisms, to further enhance their performance in various NLP applications. For example, the integration of BRNNs with attention-based models has led to significant advancements in machine translation and text generation.

Overall, the bidirectional nature of BRNNs and their ability to capture long-range dependencies make them a valuable tool in the NLP researcher's toolkit, contributing to breakthroughs in a wide range of language-related tasks and applications.

8.4.9.BRNN in Other Domains and Future Directions

While Bidirectional Recurrent Neural Networks (BRNNs) have found widespread success in natural language processing, their applications extend far beyond this field. BRNNs have shown promising results in various other domains, including speech recognition, handwriting recognition, and even genomic sequence analysis.

In the realm of speech recognition, BRNNs can leverage the contextual information from both past and future input sequences to improve accuracy and robustness. Similarly, in handwriting recognition, the bidirectional nature of BRNNs allows them to better capture the temporal dependencies and spatial features present in handwritten text.

Looking to the future, researchers are exploring the potential of BRNNs in emerging areas such as video processing, financial time series analysis, and biomedical signal processing. The ability of BRNNs to model complex, non-linear relationships and capture both forward and backward dependencies makes them well-suited for these applications.

As the field of deep learning continues to evolve, the versatility and adaptability of BRNNs will likely lead to further advancements and novel applications. Researchers are investigating ways to extend the BRNN architecture, such as incorporating attention mechanisms or combining it with other neural network models, to tackle even more challenging problems and push the boundaries of what's possible with deep learning.

8.4.9.1 BRNN Code

1. Importing Libraries

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Bidirectional, SimpleRNN, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

Figure 52: Importing Libraries For BRNN

This section imports the necessary libraries for data manipulation (Pandas and NumPy), data splitting and scaling (scikit-learn), and building and evaluating the deep learning model (TensorFlow/Keras).

2.Data Loading and Preprocessing

```
# Define column names
columns = ["Timestamp", "CAN_ID", "DLC", "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7", "Flag"]

# Load your dataset again, skipping the first row
data = pd.read_csv("/content/DoS_dataset (1).csv", names=columns, skiprows=1, low_memory=False)

# Convert hexadecimal values to integers
data['CAN_ID'] = data['CAN_ID'].apply(lambda x: int(x, 16))
data['DLC'] = data['DLC'].astype('int64')

# Define function to handle hexadecimal values
def b16(x):
    if isinstance(x, str):
        return int(x, 16) if all(c in '0123456789ABCDEFabcdef' for c in x) else None
    else:
        return None

# Apply conversion function to hexadecimal columns
for col in ['D0', 'D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']:
    data[col] = data[col].apply(b16)

# Drop rows with missing values
data = data.dropna()
data = data.drop_duplicates()
data.fillna(0, inplace=True)

# Convert float columns to integers
data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']] = data[['D1', 'D2', 'D3', 'D4', 'D5', 'D6', 'D7']].astype(int)
```

Figure 53: Data Loading and Preprocessing

Define Column Names: Specifies the column names for the dataset.
Load Dataset: Reads the dataset from the specified path, skipping the first row.
Hexadecimal to Integer Conversion: Converts the hexadecimal values in the CAN_ID column to integers.
Conversion Function: Defines a function b16 to handle conversion of hexadecimal string values to integers.
Apply Conversion: Applies the b16 function to the data columns D0 to D7.
Handling Missing Values: Drops rows with missing values and duplicates, and fills any remaining NaN values with 0.
Data Type Conversion: Converts columns D1 to D7 to integer type.
Convert Flag Column: Converts the Flag column values to binary (1 for 'R', 0 for others).
Drop Timestamp Column: Drops the Timestamp column as it is not needed for the analysis.

3.Data Balancing

```
# Convert Flag to int (if applicable)
data['Flag'] = data['Flag'].apply(lambda x: 1 if x == 'R' else 0)

# Drop the 'Timestamp' column
data = data.drop('Timestamp', axis=1)

# Separate normal and anomaly data
data_normal = data[data['Flag'] == 0]
data_anomaly = data[data['Flag'] == 1]

# Check which data is smaller
if len(data_normal) < len(data_anomaly):
    # Undersample anomaly data to match normal data size
    data_anomaly_sampled = data_anomaly.sample(n=len(data_normal), random_state=42)
    data_balanced = pd.concat([data_normal, data_anomaly_sampled])
else:
    # Undersample normal data to match anomaly data size
    data_normal_sampled = data_normal.sample(n=len(data_anomaly), random_state=42)
    data_balanced = pd.concat([data_normal_sampled, data_anomaly])
```

Figure 54: Data Balancing

Separate Normal and Anomaly Data: Splits the data into two subsets based on the Flag column.
Balancing Data: Checks which class (normal or anomaly) has fewer samples. It then undersamples the larger class to match the size of the smaller class to create a balanced dataset.
Shuffle Data: Shuffles the balanced dataset and resets the index.

Data Inspection: Prints the data types of the balanced dataset and the first few rows for inspection.

4. Feature and Target Separation

```
# Preprocess the balanced data
X = data_balanced.drop(data_balanced.columns[-1], axis=1)  # Drop the target column from the features
y = data_balanced[data_balanced.columns[-1]].values  # Assign the target column to y

# Convert string columns to numerical format (if any)
X = X.apply(pd.to_numeric, errors='coerce')
```

Figure 55: Feature and Target Separation

Feature-Target Split: Separates the features (X) from the target variable (y).
Convert Strings to Numbers: Converts any remaining string columns in X to numeric format, handling any errors by setting non-convertible entries to NaN.

5. Train-Test Split and Normalization

```
# Split the balanced data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalization (only for X)
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for BRNN input
x_train = x_train.reshape(x_train.shape[0], 1, x_train.shape[1])
x_test = x_test.reshape(x_test.shape[0], 1, x_test.shape[1])
```

Figure 56: Train-Test Split and Normalization

Train-Test Split: Splits the balanced dataset into training and testing sets, maintaining the class distribution (stratification) with an 80-20 split.
Normalization: Scales the feature data using StandardScaler to have a mean of 0 and a standard deviation of 1.
Reshape Data: Reshapes the feature data to be compatible with the input requirements of a Conv1D model (3D shape: samples, timesteps, features).

6.Model Definition

```
# Model
model = Sequential()
model.add(Bidirectional(SimpleRNN(128, return_sequences=True, activation='relu'), input_shape=(1, x_train.shape[2])))
model.add(Dropout(0.2))  # Add dropout layer
model.add(Bidirectional(SimpleRNN(128, return_sequences=False, activation='relu')))
model.add(Dropout(0.2))  # Add dropout layer
model.add(Dense(1, activation='sigmoid'))  # Output units = 1 (binary classification)
print(model.summary())
```

Figure 57: Model Definition For BRNN

Model Definition: Defines a Sequential model for anomaly detection.
Conv1D Layers: Adds two 1D convolutional layers with 64 and 128 filters respectively, each followed by a max pooling layer.
Dropout Layers: Adds dropout layers with a rate of 0.5 after the convolutional layers and the dense layer to prevent overfitting.
Flatten Layer: Flattens the output of the convolutional layers.
Dense Layers: Adds a dense (fully connected) layer with 128 units and a dropout layer, followed by an output layer with a sigmoid activation function for binary classification.
Model Summary: Prints the summary of the model architecture.

8.Model Compilation and Training with Noise

```
# Compile the model
loss = 'binary_crossentropy'
optim = 'RMSprop'  # Try a different optimizer
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optim, metrics=metrics)

# Adding noise to training data
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_train_noisy = np.clip(x_train_noisy, -1.0, 1.0)

# Training
batch_size = 512
epochs = 10
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)  # Early stopping callback

history = model.fit(x_train_noisy, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True, verbose=2,
```

Figure 58: Model Compilation and Training

Model Compilation: Compiles the model using binary cross-entropy loss, Adam optimizer, and accuracy as the evaluation metric.

Adding Noise: Introduces Gaussian noise to the training data to improve the model's robustness.
Training: Trains the model using the noisy training data with a batch size of 512 and 10 epochs, validating on the test set, and applying class weights.

9. Model Evaluation

```
# Evaluate on the test set
y_pred = model.predict(x_test)
y_pred = (y_pred > 0.5).astype(int)  # Convert probabilities to binary predictions

# Calculate additional metrics
accuracy = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)[1]

print(f"Test Accuracy: {accuracy:.4f}")
```

Figure 59: Model Evaluation

Prediction: Makes predictions on the test set and converts the probabilities to binary class labels.
Evaluation: Computes the accuracy to evaluate the model's performance.
Print Results: Prints the calculated evaluation metrics.

8.4.9.2 Accuracy

1.Dos Attack

```
_________________________________________________________________
None
Epoch 1/10
1040/1040 - 27s - loss: 0.0150 - accuracy: 0.9968 - val_loss: 0.0082 - val_accuracy: 0.9981 - 27s/epoch - 26ms/step
Epoch 2/10
1040/1040 - 21s - loss: 0.0098 - accuracy: 0.9980 - val_loss: 0.0074 - val_accuracy: 0.9984 - 21s/epoch - 21ms/step
Epoch 3/10
1040/1040 - 24s - loss: 0.0096 - accuracy: 0.9981 - val_loss: 0.0077 - val_accuracy: 0.9983 - 24s/epoch - 24ms/step
Epoch 4/10
1040/1040 - 22s - loss: 0.0095 - accuracy: 0.9981 - val_loss: 0.0072 - val_accuracy: 0.9984 - 22s/epoch - 21ms/step
Epoch 5/10
1040/1040 - 21s - loss: 0.0093 - accuracy: 0.9981 - val_loss: 0.0071 - val_accuracy: 0.9984 - 21s/epoch - 21ms/step
Epoch 6/10
1040/1040 - 23s - loss: 0.0093 - accuracy: 0.9981 - val_loss: 0.0077 - val_accuracy: 0.9983 - 23s/epoch - 22ms/step
Epoch 7/10
1040/1040 - 22s - loss: 0.0093 - accuracy: 0.9981 - val_loss: 0.0067 - val_accuracy: 0.9984 - 22s/epoch - 21ms/step
Epoch 8/10
1040/1040 - 24s - loss: 0.0092 - accuracy: 0.9982 - val_loss: 0.0069 - val_accuracy: 0.9984 - 24s/epoch - 23ms/step
Epoch 9/10
1040/1040 - 22s - loss: 0.0092 - accuracy: 0.9981 - val_loss: 0.0065 - val_accuracy: 0.9984 - 22s/epoch - 21ms/step
Epoch 10/10
1040/1040 - 23s - loss: 0.0092 - accuracy: 0.9981 - val_loss: 0.0067 - val_accuracy: 0.9984 - 23s/epoch - 22ms/step
4158/4158 [==============================] - 11s 2ms/step
Test Accuracy: 0.9984
```

Figure 60: Accuracy DoS Attack For BRNN

2.Fuzzy Attack

```
_________________________________________________________________
None
Epoch 1/10
1538/1538 - 42s - loss: 0.0189 - accuracy: 0.9950 - val_loss: 0.0069 - val_accuracy: 0.9983 - 42s/epoch - 27ms/step
Epoch 2/10
1538/1538 - 41s - loss: 0.0056 - accuracy: 0.9986 - val_loss: 0.0033 - val_accuracy: 0.9994 - 41s/epoch - 27ms/step
Epoch 3/10
1538/1538 - 41s - loss: 0.0039 - accuracy: 0.9990 - val_loss: 0.0028 - val_accuracy: 0.9995 - 41s/epoch - 27ms/step
Epoch 4/10
1538/1538 - 42s - loss: 0.0031 - accuracy: 0.9992 - val_loss: 0.0093 - val_accuracy: 0.9978 - 42s/epoch - 28ms/step
Epoch 5/10
1538/1538 - 43s - loss: 0.0027 - accuracy: 0.9993 - val_loss: 0.0091 - val_accuracy: 0.9981 - 43s/epoch - 28ms/step
Epoch 6/10
1538/1538 - 36s - loss: 0.0025 - accuracy: 0.9994 - val_loss: 0.0426 - val_accuracy: 0.9720 - 36s/epoch - 23ms/step
Epoch 7/10
1538/1538 - 32s - loss: 0.0023 - accuracy: 0.9994 - val_loss: 0.0557 - val_accuracy: 0.9721 - 32s/epoch - 21ms/step
Epoch 8/10
1538/1538 - 33s - loss: 0.0022 - accuracy: 0.9995 - val_loss: 0.0985 - val_accuracy: 0.9715 - 33s/epoch - 22ms/step
6149/6149 [==============================] - 15s 2ms/step
Test Accuracy: 0.9995
```

Figure 61: Accuracy Fuzzy Attack For BRNN

3.Gear Attack

```
_________________________________________________________________
None
Epoch 1/10
497/497 - 7s - loss: 0.3151 - accuracy: 0.8692 - val_loss: 0.0734 - val_accuracy: 0.9735 - 7s/epoch - 14ms/step
Epoch 2/10
497/497 - 3s - loss: 0.1938 - accuracy: 0.9339 - val_loss: 0.0608 - val_accuracy: 0.9823 - 3s/epoch - 6ms/step
Epoch 3/10
497/497 - 5s - loss: 0.1759 - accuracy: 0.9397 - val_loss: 0.0558 - val_accuracy: 0.9841 - 5s/epoch - 9ms/step
Epoch 4/10
497/497 - 3s - loss: 0.1670 - accuracy: 0.9425 - val_loss: 0.0531 - val_accuracy: 0.9849 - 3s/epoch - 6ms/step
Epoch 5/10
497/497 - 3s - loss: 0.1610 - accuracy: 0.9447 - val_loss: 0.0526 - val_accuracy: 0.9854 - 3s/epoch - 5ms/step
Epoch 6/10
497/497 - 3s - loss: 0.1578 - accuracy: 0.9457 - val_loss: 0.0525 - val_accuracy: 0.9857 - 3s/epoch - 6ms/step
Epoch 7/10
497/497 - 3s - loss: 0.1544 - accuracy: 0.9469 - val_loss: 0.0543 - val_accuracy: 0.9855 - 3s/epoch - 7ms/step
Epoch 8/10
497/497 - 5s - loss: 0.1522 - accuracy: 0.9476 - val_loss: 0.0548 - val_accuracy: 0.9856 - 5s/epoch - 9ms/step
Epoch 9/10
497/497 - 3s - loss: 0.1514 - accuracy: 0.9479 - val_loss: 0.0554 - val_accuracy: 0.9854 - 3s/epoch - 6ms/step
Epoch 10/10
497/497 - 4s - loss: 0.1496 - accuracy: 0.9488 - val_loss: 0.0569 - val_accuracy: 0.9838 - 4s/epoch - 8ms/step
1985/1985 [==============================] - 5s 2ms/step
Test Accuracy: 0.9838
```

Figure 62: Accuracy Gear Attack For BRNN

visualization methods, To ensure that my model is not overfitting:

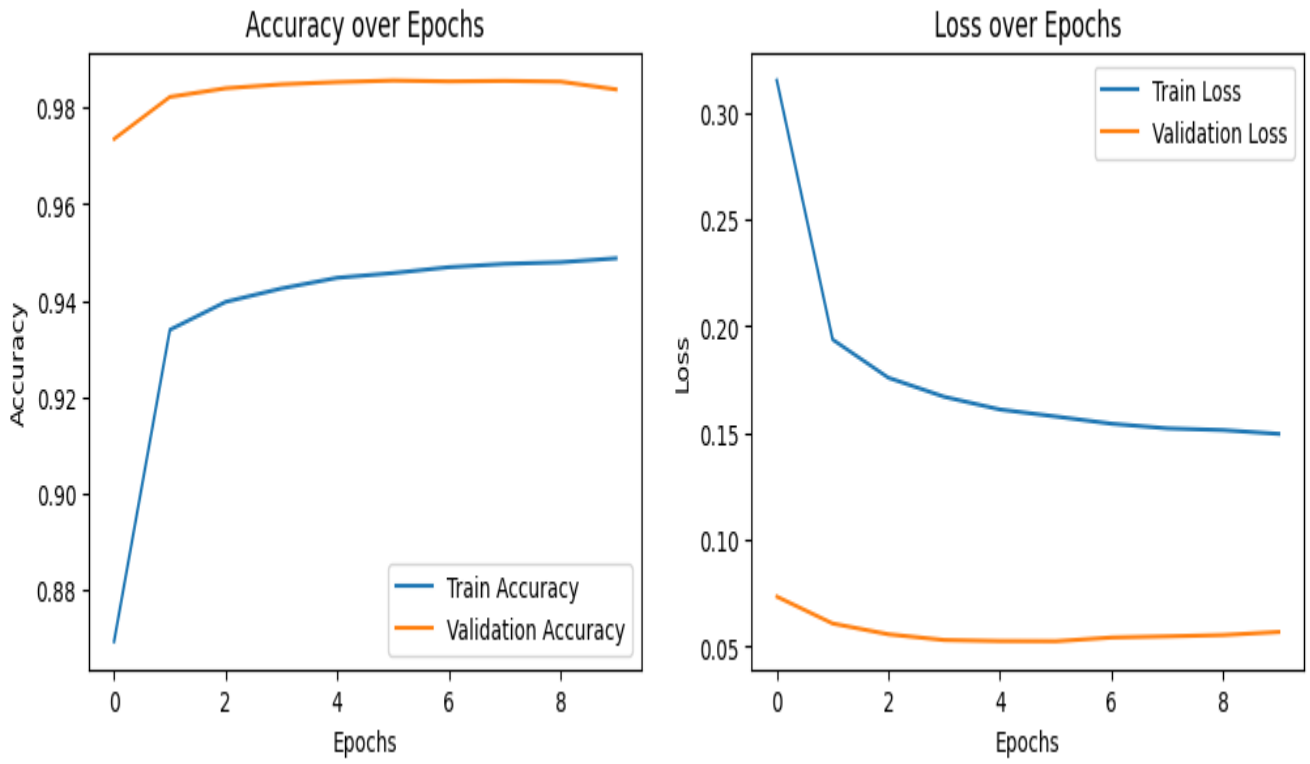


Figure 63: Visualization Gear For BRNN

4.RPM Attack

```
_________________________________________________________________
None
Epoch 1/10
344/344 - 8s - loss: 0.3646 - accuracy: 0.8331 - val_loss: 0.0128 - val_accuracy: 0.9977 - 8s/epoch - 23ms/step
Epoch 2/10
344/344 - 2s - loss: 0.1906 - accuracy: 0.9342 - val_loss: 0.0046 - val_accuracy: 0.9997 - 2s/epoch - 5ms/step
Epoch 3/10
344/344 - 2s - loss: 0.1643 - accuracy: 0.9448 - val_loss: 0.0028 - val_accuracy: 0.9999 - 2s/epoch - 6ms/step
Epoch 4/10
344/344 - 2s - loss: 0.1500 - accuracy: 0.9496 - val_loss: 0.0021 - val_accuracy: 1.0000 - 2s/epoch - 5ms/step
Epoch 5/10
344/344 - 2s - loss: 0.1401 - accuracy: 0.9521 - val_loss: 0.0025 - val_accuracy: 1.0000 - 2s/epoch - 5ms/step
Epoch 6/10
344/344 - 2s - loss: 0.1323 - accuracy: 0.9545 - val_loss: 0.0021 - val_accuracy: 1.0000 - 2s/epoch - 5ms/step
Epoch 7/10
344/344 - 2s - loss: 0.1275 - accuracy: 0.9560 - val_loss: 0.0018 - val_accuracy: 1.0000 - 2s/epoch - 6ms/step
Epoch 8/10
344/344 - 3s - loss: 0.1249 - accuracy: 0.9573 - val_loss: 0.0025 - val_accuracy: 1.0000 - 3s/epoch - 9ms/step
Epoch 9/10
344/344 - 2s - loss: 0.1213 - accuracy: 0.9574 - val_loss: 0.0030 - val_accuracy: 1.0000 - 2s/epoch - 6ms/step
Epoch 10/10
344/344 - 2s - loss: 0.1198 - accuracy: 0.9575 - val_loss: 0.0021 - val_accuracy: 1.0000 - 2s/epoch - 6ms/step
1376/1376 [==============================] - 3s 2ms/step
Test Accuracy: 1.0000
```

Figure 64: Accuracy RPM Attack For BRNN

8.4.9.3.Code For Merge Four Dataset

```
[1] import tensorflow as tf
    from tensorflow import keras
    from keras.models import Sequential
    from keras.layers import Dense, Dropout, BatchNormalization, Bidirectional, SimpleRNN
    from keras.callbacks import EarlyStopping, ReduceLROnPlateau
    from keras.optimizers import Adam
    import pandas as pd
    import numpy as np
    from sklearn.preprocessing import StandardScaler, LabelEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.utils import compute_class_weight
    from keras.utils import to_categorical

    # Load the dataset
    columns = ['Timestamp', 'CAN_ID', 'DLC', 'DATA[0]', 'DATA[1]', 'DATA[2]', 'DATA[3]', 'DATA[4]', 'DATA[5]', 'DATA[6]', 'DATA[7]', 'type', 'flag']
    file_path = r"/content/Data.csv"
    df = pd.read_csv(file_path, header=None, names=columns, skiprows=1)

    # Preprocess the data
    df['flag'].replace(['R','T'], [0, 1], inplace=True)

    # Remove non-finite values and drop NaNs
    df.dropna(subset=['flag'], inplace=True)

    # Ensure 'flag' is in integer format
    df['flag'] = df['flag'].astype(int)

    df.drop_duplicates(inplace=True)
```

Figure 65: Code For Merge Four Dataset For BRNN

```
    # Convert hexadecimal columns to integers
    columns_to_convert = ['Timestamp', 'CAN_ID', 'DLC', 'DATA[0]', 'DATA[1]', 'DATA[2]', 'DATA[3]', 'DATA[4]', 'DATA[5]', 'DATA[6]', 'DATA[7]']
    for column in columns_to_convert:
        df[column] = df[column].apply(lambda x: int(x, 16) if isinstance(x, str) else x)

    df = df.drop('Timestamp', axis=1)
    encode = LabelEncoder()
    df["type"] = encode.fit_transform(df["type"])


    # Balance the dataset
    from sklearn.utils import resample

    normal_messages = df[df['flag'] == 0]
    injected_messages = df[df['flag'] == 1]
    normal_downsampled = resample(normal_messages, replace=False, n_samples=len(injected_messages), random_state=42)
    undersampled_df = pd.concat([normal_downsampled, injected_messages]).sample(frac=1).reset_index(drop=True)

    # Prepare features and labels
    X = undersampled_df.drop(['flag'], axis=1)
    y = undersampled_df['flag'].values

    # Convert string columns to numerical format
    X = X.apply(pd.to_numeric, errors='coerce')
```

Figure 66

```
[1]
# Split the data
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalize the features
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

# Reshape the data for BRNN input
x_train = x_train.reshape(x_train.shape[0], 1, x_train.shape[1])
x_test = x_test.reshape(x_test.shape[0], 1, x_test.shape[1])

# Compute class weights
classes = np.unique(y_train)
class_weights = compute_class_weight(class_weight='balanced', classes=classes, y=y_train)
class_weights = dict(enumerate(class_weights))

# Build the BRNN model
model = Sequential()
model.add(Bidirectional(SimpleRNN(64, return_sequences=True, activation='tanh'), input_shape=(1, x_train.shape[2])))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Bidirectional(SimpleRNN(32, return_sequences=True, activation='tanh')))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Bidirectional(SimpleRNN(16, return_sequences=False, activation='tanh')))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))
```

```
# Print the model summary
model.summary()

# Compile the model
loss = 'binary_crossentropy'
optimizer = Adam(learning_rate=0.001)
metrics = ['accuracy']
model.compile(loss=loss, optimizer=optimizer, metrics=metrics)

# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, min_lr=1e-6)

# Training
batch_size = 64
epochs = 10
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test, y_test), shuffle=True, verbose=2, class_weight=class_

# Evaluate on the test set
test_loss, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size, verbose=2)
print(f"Test Accuracy: {test_acc:.4f}")
```

Figure 67: Code For Merge Four Dataset For BRNN

1. ACCURACY

```
_________________________________________________________________
Epoch 1/10
267/267 - 10s - loss: 0.4415 - accuracy: 0.7811 - val_loss: 0.1783 - val_accuracy: 0.9159 - lr: 0.0010 - 10s/epoch - 39ms/step
Epoch 2/10
267/267 - 2s - loss: 0.2425 - accuracy: 0.8991 - val_loss: 0.0883 - val_accuracy: 0.9843 - lr: 0.0010 - 2s/epoch - 9ms/step
Epoch 3/10
267/267 - 3s - loss: 0.1652 - accuracy: 0.9416 - val_loss: 0.0536 - val_accuracy: 0.9892 - lr: 0.0010 - 3s/epoch - 10ms/step
Epoch 4/10
267/267 - 2s - loss: 0.1379 - accuracy: 0.9547 - val_loss: 0.0470 - val_accuracy: 0.9897 - lr: 0.0010 - 2s/epoch - 7ms/step
Epoch 5/10
267/267 - 2s - loss: 0.1183 - accuracy: 0.9624 - val_loss: 0.0400 - val_accuracy: 0.9902 - lr: 0.0010 - 2s/epoch - 6ms/step
Epoch 6/10
267/267 - 2s - loss: 0.1071 - accuracy: 0.9677 - val_loss: 0.0392 - val_accuracy: 0.9899 - lr: 0.0010 - 2s/epoch - 6ms/step
Epoch 7/10
267/267 - 2s - loss: 0.0947 - accuracy: 0.9722 - val_loss: 0.0383 - val_accuracy: 0.9902 - lr: 0.0010 - 2s/epoch - 7ms/step
Epoch 8/10
267/267 - 2s - loss: 0.0912 - accuracy: 0.9757 - val_loss: 0.0329 - val_accuracy: 0.9913 - lr: 0.0010 - 2s/epoch - 7ms/step
Epoch 9/10
267/267 - 3s - loss: 0.0796 - accuracy: 0.9784 - val_loss: 0.0335 - val_accuracy: 0.9923 - lr: 0.0010 - 3s/epoch - 10ms/step
Epoch 10/10
267/267 - 3s - loss: 0.0874 - accuracy: 0.9766 - val_loss: 0.0322 - val_accuracy: 0.9916 - lr: 0.0010 - 3s/epoch - 10ms/step
67/67 - 0s - loss: 0.0322 - accuracy: 0.9916 - 181ms/epoch - 3ms/step
Test Accuracy: 0.9916
```

Figure 68: Accuracy Merge Data For BRNN

Chapter 9

Application

9.1.PROTOTYPE OF APPLICATION

1.Welcom Page

Figure 69:Home Page

the welcome page contains 2 buttons.

the first button is login button which directs you to login page.

if you don't have an account, you can choose the sign it is second button.

2.Log In

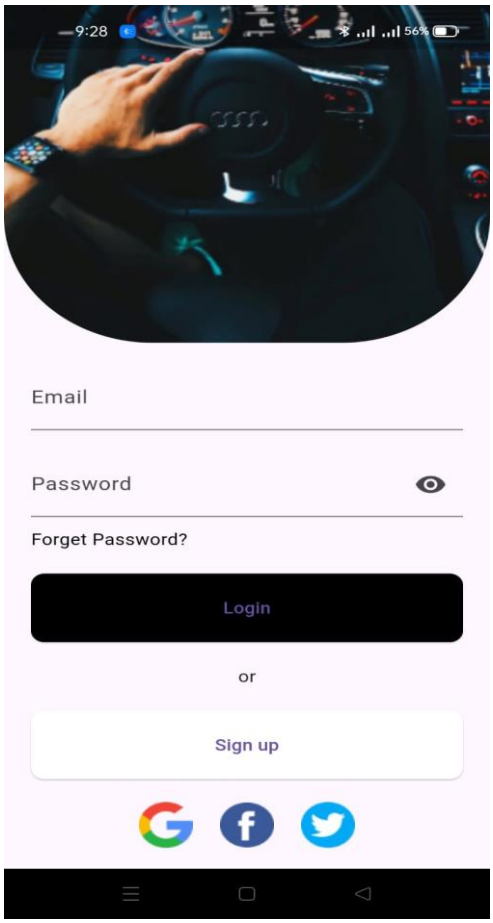


Figure 70:Log In

the user goes to the login page and enters his credentials to get access to the application. If the user entered his credentials wrong, the application wouldn't open for him. If the user entered his credentials correctly and pressed the login button, then he is routed to the home page.

3.Sign Up

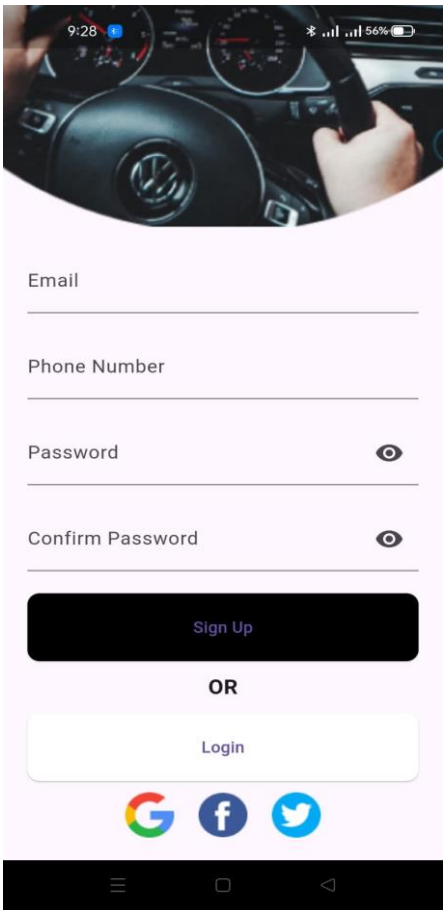


Figure 71:sign Up

- **This is Sign Up page, when driver not have any account, driver must sign up on application ,How? The first step Enter username or email, Second step Phone Number, Third step Password, Fourth step confirm password**

4.Data Trip

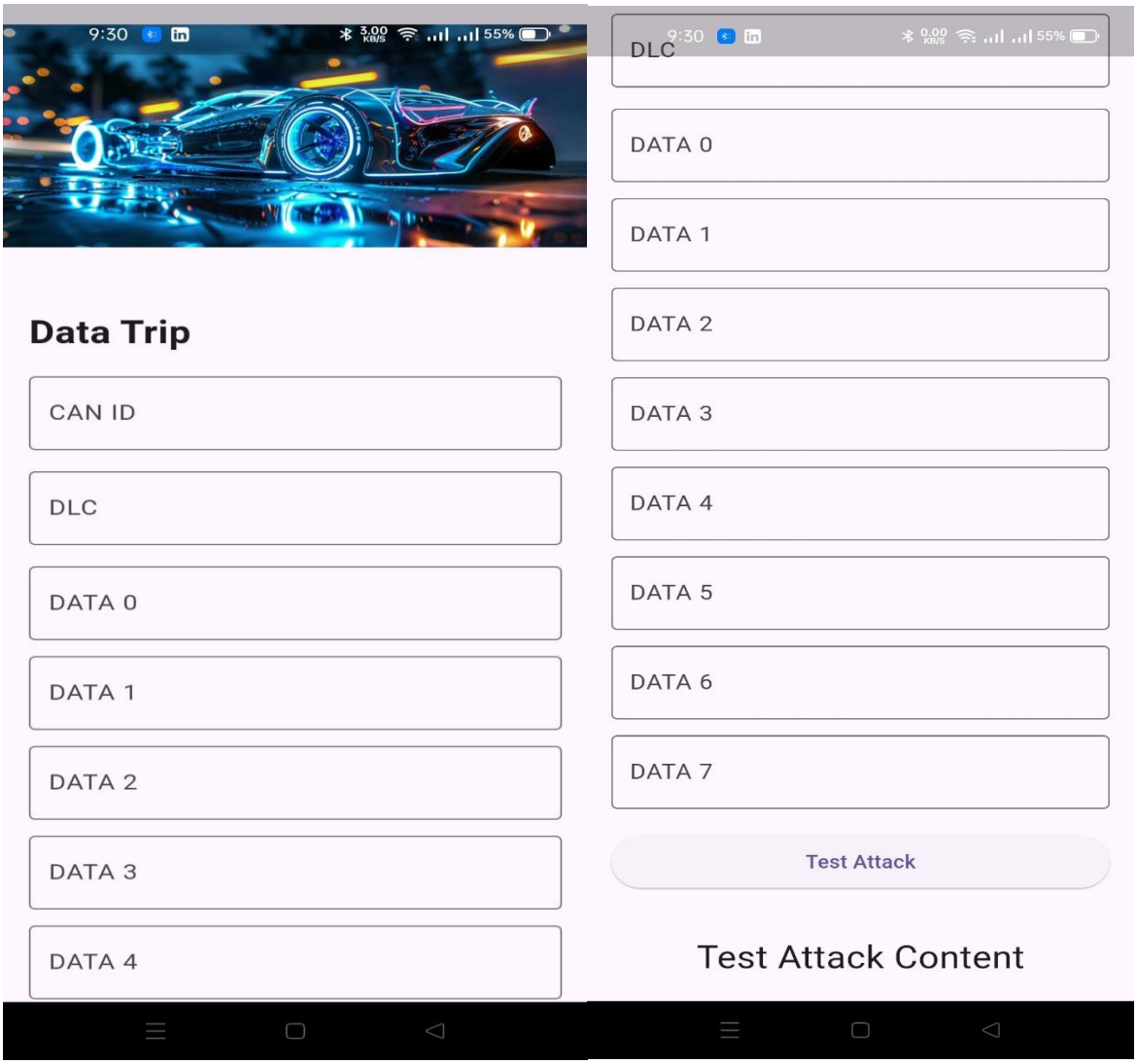


Figure 72 Trip Data

- **In the page, Enter Can ID and data 0 to data 7 and click on Test Attack ,wait the result after few minute if take attack or no attack**

9.2.Coding:

1.Main

```dart
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'dart:io';
import 'package:flutter_application_1/welcome_page.dart';
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  try {
    if (Platform.isAndroid) {
      await Firebase.initializeApp(
        options: FirebaseOptions(
          apiKey: 'AIzaSyB2hq91sWvZHZCMLAUaKIPB_D8lcI1ixgc',
          appId: '1:389387411447:android:2ff2517683f9792a0f98f2',
          messagingSenderId: '389387411447',
          projectId: 'self-driving-intrusions',
          storageBucket: 'self-driving-intrusions.appspot.com',
        ),
      );
    } else {
      await Firebase.initializeApp();
    }
  } catch (e) {
    // Handle initialization error
    print('Firebase initialization error: $e');
  }
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const WelcomePage(),
      debugShowCheckedModeBanner: false,
    );
  }
}

```

Figure 73 Main code

2.Welcome Page

```dart
import 'package:flutter/material.dart';
import 'login_page.dart';
import 'signup_page.dart';

class WelcomePage extends StatelessWidget {
  const WelcomePage({super.key});

  void _navigateToLoginPage(BuildContext context) {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const LoginPage()),
    );
  }

  void _navigateToSignupPage(BuildContext context) {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => const SignupPage()),
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Stack(
        fit: StackFit.expand,
        children: [
          Image.asset(
            'assets/background.png', // Ensure you have this image in your assets folder
            fit: BoxFit.cover,
          ),
          Column(
            mainAxisAlignment: MainAxisAlignment.start,
            children: [
              Padding(
                padding: const EdgeInsets.symmetric(vertical: 60.0),
                child: const Text(
                  'Welcome',
                  style: TextStyle(
                    fontSize: 48, // Increased font size
                    fontWeight: FontWeight.bold,
                    color: Colors.black, // Changed color to black
                  ),
                ),
              ),
            ],
          ),
          Positioned(
            bottom: 30,
            left: 20,
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                ElevatedButton(
                  onPressed: () => _navigateToLoginPage(context),
                  style: ElevatedButton.styleFrom(
                    foregroundColor: Colors.black,
                    backgroundColor: Colors.white,
                    shape: RoundedRectangleBorder(
                      borderRadius: BorderRadius.circular(30.0), // Sphere shape
                    ), // Text color black
                    padding: const EdgeInsets.symmetric(horizontal: 40, vertical: 15), // Increased padding
                    textStyle: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold), // Increased font size and bold
                  ),
                  child: const Text('Login'),
                ),
                const SizedBox(height: 10),
                ElevatedButton(
                  onPressed: () => _navigateToSignupPage(context),
                  style: ElevatedButton.styleFrom(
                    foregroundColor: Colors.black,
                    backgroundColor: Colors.white,
                    shape: RoundedRectangleBorder(
                      borderRadius: BorderRadius.circular(30.0), // Sphere shape
                    ), // Text color black
                    padding: const EdgeInsets.symmetric(horizontal: 40, vertical: 15), // Increased padding
                    textStyle: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold), // Increased font size and bold
                  ),
                  child: const Text('Sign Up'),
                ),
              ],
            ),
          ),
        ],
      ),
    );
  }
}

```

Figure 74 Welcome page code

3.Login Page

```dart

class LoginPage extends StatefulWidget {
  const LoginPage({Key? key}) : super(key: key);

  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
  bool _isObscure = true; // Manage password visibility
  String? _emailError;
  String? _passwordError;

  void _login(BuildContext context) async {
    // Validate form fields before proceeding
    if (_validateEmail(_emailController.text) == null &&
        _validatePassword(_passwordController.text) == null) {
      try {
        // Sign in with email and password
        UserCredential userCredential =
        await FirebaseAuth.instance.signInWithEmailAndPassword(
          email: _emailController.text,
          password: _passwordController.text,
        );

        // login successful, handle the next step (e.g., navigate to another page)
        print('User logged in: ${userCredential.user?.email}');
        Navigator.pushReplacement(
          context,
          MaterialPageRoute(builder: (context) => const HomePage()),
        );
      } on FirebaseAuthException catch (e) {
        // Handle specific Firebase Auth exceptions
        print('Error logging in: $e');
        showDialog(
          context: context,
          builder: (context) => AlertDialog(
            title: Text('Error'),
            content: Text(e.message ?? 'An error occurred while logging in.'),
            actions: <Widget>[
              TextButton(
                child: Text('OK'),
                onPressed: () {
                  Navigator.of(context).pop();
                },
              ),
            ],
          ),
        );
      } catch (e) {
        // Handle any other exceptions
        print('Error logging in: $e');
        showDialog(
          context: context,
          builder: (context) => AlertDialog(
            title: Text('Error'),
            content: Text('An error occurred while logging in.'),
            actions: <Widget>[
              TextButton(
                child: Text('OK'),
                onPressed: () {
                  Navigator.of(context).pop();
                },
              ),
            ],
          ),
        );
      }
    } else {
      // Show error dialog or message for invalid inputs
      showDialog(
        context: context,
        builder: (context) => AlertDialog(
          title: Text('Error'),
          content: Text('Please fix the errors in the form.'),
          actions: <Widget>[
            TextButton(
              child: Text('OK'),
              onPressed: () {
                Navigator.of(context).pop();
              },
            ),
          ],
        ),
      );
    }
  }

  String? _validateEmail(String? value) {
    if (value == null || value.isEmpty) {
      return 'Email is required';
    }
    // Simple email validation example, adjust as needed
    if (!value.contains('@')) {
      return 'Please enter a valid email';
    }
    return null;
  }

  String? _validatePassword(String? value) {
    if (value == null || value.isEmpty) {
      return 'Password is required';
    }
    if (value.length < 8) {
      return 'Password must be at least 8 characters';
    }
    // Check for uppercase letters
    if (!value.contains(RegExp(r'[A-Z]'))) {
      return 'Password must contain at least one uppercase letter';
    }
    // Check for lowercase letters
    if (!value.contains(RegExp(r'[a-z]'))) {
      return 'Password must contain at least one lowercase letter';
    }
    // Check for numbers
    if (!value.contains(RegExp(r'[0-9]'))) {
      return 'Password must contain at least one number';
    }
    // Check for special characters
    if (!value.contains(RegExp(r'[!@#$%^&*(),.?":{}|<>]'))) {
      return 'Password must contain at least one special character';
    }
    return null;
  }
```

Figure 75 Login page code

4.Sign Up

```dart
class SignupPage extends StatefulWidget {
  const SignupPage({Key? key}) : super(key: key);

  @override
  _SignupPageState createState() => _SignupPageState();
}

class _SignupPageState extends State<SignupPage> {
  TextEditingController _emailController = TextEditingController();
  TextEditingController _phoneNumberController = TextEditingController();
  TextEditingController _passwordController = TextEditingController();
  TextEditingController _confirmPasswordController = TextEditingController();
  bool _isObscure = true;
  bool _isObscuree = true;

  String? _emailError;
  String? _phoneNumberError;
  String? _passwordError;
  String? _confirmPasswordError;

  void _signup(BuildContext context) async {
    // Validate form fields before proceeding
    if (_validateEmail(_emailController.text) == null &&
        _validatePhoneNumber(_phoneNumberController.text) == null &&
        _validatePassword(_passwordController.text) == null &&
        _validateConfirmPassword(_confirmPasswordController.text) == null) {
      try {
        // Create a new user with email and password
        UserCredential userCredential =
            await FirebaseAuth.instance.createUserWithEmailAndPassword(
          email: _emailController.text,
          password: _passwordController.text,
        );

        // User registration successful, handle the next step (e.g., navigate to another page)
        print('User registered: ${userCredential.user?.email}');
        Navigator.pushReplacement(
          context,
          MaterialPageRoute(builder: (context) => const LoginPage()),
        );
      } on FirebaseAuthException catch (e) {
        // Handle specific FirebaseAuth exceptions
        String errorMessage = _handleFirebaseAuthError(e);
        showDialog(
          context: context,
          builder: (context) => AlertDialog(
            title: Text('Error'),
            content: Text(errorMessage),
            actions: <Widget>[
              TextButton(
                child: Text('OK'),
                onPressed: () {
                  Navigator.of(context).pop();
                },
              ),
            ],
          ),
        );
      } catch (e) {
        // Handle any other exceptions
        print('Error registering user: $e');
        showDialog(
          context: context,
          builder: (context) => AlertDialog(
            title: Text('Error'),
            content: Text('An error occurred while registering the user.'),
            actions: <Widget>[
              TextButton(
                child: Text('OK'),
                onPressed: () {
                  Navigator.of(context).pop();
                },
              ),
            ],
          ),
        );
      }
    } else {
      // Show error dialog or message for invalid inputs
      showDialog(
        context: context,
        builder: (context) => AlertDialog(
          title: Text('Error'),
          content: Text('Please fix the errors in the form.'),
          actions: <Widget>[
            TextButton(
              child: Text('OK'),
              onPressed: () {
                Navigator.of(context).pop();
              },
            ),
          ],
        ),
      );
    }
  }

  String _handleFirebaseAuthError(FirebaseAuthException e) {
    switch (e.code) {
      case 'weak-password':
        return 'The password provided is too weak.';
      case 'email-already-in-use':
        return 'The account already exists for that email.';
      case 'invalid-email':
        return 'The email address is not valid.';
      default:
        return 'Error: ${e.message ?? 'An error occurred while registering the user.'}';
    }
  }

  String? _validateEmail(String? value) {
    if (value == null || value.isEmpty) {
      return 'Email is required';
    }
    // Simple email validation example, adjust as needed
    if (!value.contains('@')) {
      return 'Please enter a valid email';
    }
    return null;
  }

  String? _validatePhoneNumber(String? value) {
    if (value == null || value.isEmpty) {
      return 'Phone number is required';
    }
    // Simple phone number validation example, adjust as needed
    if (value.length != 11) {
      return 'Phone number must be 11 digits';
    }
    return null;
  }

  String? _validatePassword(String? value) {
    if (value == null || value.isEmpty) {
      return 'Password is required';
    }
    if (value.length < 8) {
      return 'Password must be at least 8 characters';
    }
    // Check for uppercase letters
    if (!value.contains(RegExp(r'[A-Z]'))) {
      return 'Password must contain at least one uppercase letter';
    }
    // Check for lowercase letters
    if (!value.contains(RegExp(r'[a-z]'))) {
      return 'Password must contain at least one lowercase letter';
    }
    // Check for numbers
    if (!value.contains(RegExp(r'[0-9]'))) {
      return 'Password must contain at least one number';
    }
    // Check for special characters
    if (!value.contains(RegExp(r'[!@#$%^&*(),.?":{}|<>]'))) {
      return 'Password must contain at least one special character';
    }
    return null;
  }

  String? _validateConfirmPassword(String? value) {
    if (value != _passwordController.text) {
      return 'Passwords do not match';
    }
    return null;
  }
```

Figure 76 sign Up page code

5.Trip Data page

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Data Trip',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const HomePage(),
    );
  }
}

class HomePage extends StatefulWidget {
  const HomePage({Key? key}) : super(key: key);

  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  int _attackResult = -1;
  int _selectedIndex = 0;
  String _selectedCanId = '';
  String _selectedDLC = '1'; // Default DLC value
  final List<Widget> _pages = [
    Center(child: Text('Test Attack Content', style: TextStyle(fontSize: 24))),
    Center(child: Text('Attack Content', style: TextStyle(fontSize: 24))),
    Center(child: Text('No Attack Content', style: TextStyle(fontSize: 24))),
  ];

  // Controllers for the text fields
  List<TextEditingController> _textControllers =
      List.generate(8, (index) => TextEditingController());

  @override
  void dispose() {
    // Clean up the controllers when the widget is disposed
    _textControllers.forEach((controller) => controller.dispose());
    super.dispose();
  }

```

Figure 77 Trip Data page code

6.Flask API

```python
from flask import Flask , request
import numpy as np
import keras
from keras.models import load_model
import joblib

print(keras.__version__)

app = Flask(__name__)
model = load_model("./model_inception.hdf5")
scaler = joblib.load("./scaler.save")

# python -m flask --app index run

@app.route("/predict", methods=['POST'])
def predict():
    # recive data from the user
    request_data = request.get_json()

    can_id = int(request_data['can_id'], 16)  # Convert hex to int
    dlc = int(request_data['dlc'])
    data = [int(request_data['data'][i], 16) for i in range(8)]  # Convert hex to int

    # Preprocess the input data as necessary
    input_data = np.array([[can_id, dlc] + data ])

    input_data = scaler.transform(input_data)


    print(input_data)
    ## test data
    # can_id = -1.58416054
    # dlc = 0
    # d0=-0.67226711
    # d1= -0.90624512
    # d2=-0.82496983
    # d3=-0.99551724
    # d4=-0.83802093
    # d5=-0.71949151
    # d6=-0.44731662
    # d7=-0.73740763
    # Preprocess the input data as necessary

    prediction = model.predict(input_data)
    return str(prediction[0][0])
```

Figure 78 Flask API

7.connect flutter and flask

```dart

Future<void> fetchAlbum() async {
  try {
    final result = await http.post(
      Uri.parse('http://10.0.2.2:5000/predict'),
      headers: {
        'Content-Type': 'application/json; charset=UTF-8',
      },
      body: jsonEncode({
        "can_id": _selectedCanId,
        "dlc": _selectedDLC,
        "data": [
          _textControllers[0].text,
          _textControllers[1].text,
          _textControllers[2].text,
          _textControllers[3].text,
          _textControllers[4].text,
          _textControllers[5].text,
          _textControllers[6].text,
          _textControllers[7].text,
        ],
      }),
    );

    if (result.statusCode == 200) {
      final String modelOutput = result.body;
      setState(() {
        _attackResult = double.parse((modelOutput)).round();
      });
      // Update UI or display modelOutput as needed
      print('Response from Flask: $modelOutput');
    } else {
      // Handle other status codes here
      print('Error response: ${result.statusCode}');
    }
  } catch (e) {
    print('Error sending request: $e');
    // Handle errors from request
  }
}

```

Figure 79 code connect Flutter For Flask

8.Firebase Authentication

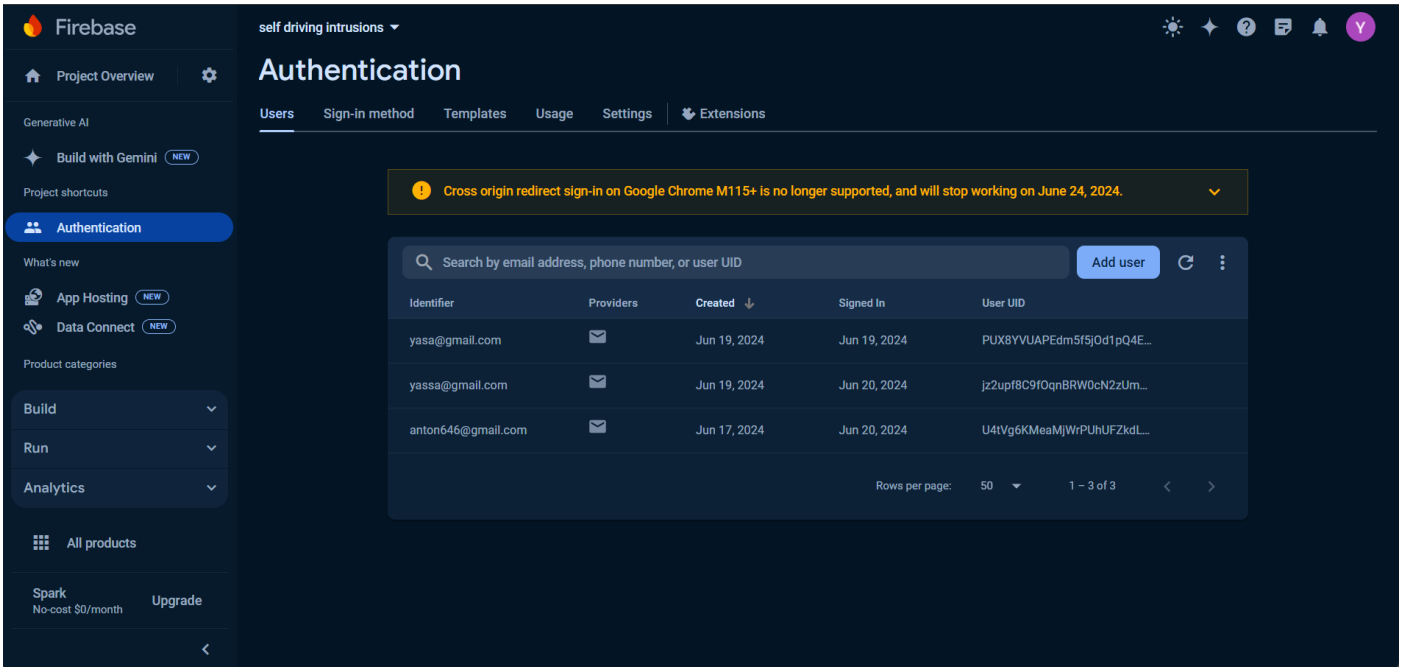


Figure 80 Firebase Authentication

Chapter 10

Conclousion

10. Conclousion

10.1.Summary

Self-driving cars are the future of transportation, and deep learning is playing a crucial role in making this technology a reality. Deep learning algorithms, powered by artificial neural networks, are enabling cars to perceive their surroundings, recognize objects, and make complex decisions in real-time, all while ensuring the safety of passengers and pedestrians. This overview will explore the key components of deep learning systems that are driving the development of autonomous vehicles, from computer vision and sensor fusion to end-to-end learning and reinforcement learning.

10.2.Conclusion

The deep learning system for discovering self-driving intrusion has shown promising results in identifying potential vulnerabilities and threats to autonomous vehicle systems. By leveraging advanced neural network architectures and large datasets, this system can effectively detect anomalies, unauthorized access attempts, and other suspicious activities that could compromise the safety and security of self-driving vehicles. The ability to proactively identify and mitigate these risks is crucial for ensuring the reliable and trustworthy deployment of autonomous driving technologies.

Key Takeaways and Future Considerations

Robust Threat Detection

The deep learning system has demonstrated its ability to detect a wide range of potential threats, from software vulnerabilities to physical tampering, with a high degree of accuracy. This capability is essential for maintaining the integrity of self-driving systems and protecting against malicious actors.

Adaptive Learning

The system's ability to continuously learn and adapt to new threat patterns is a crucial advantage. As cybersecurity threats evolve, this adaptability ensures the system can stay ahead of the curve, providing ongoing protection for self-driving vehicles.

Scalability and Deployment

The scalability of the deep learning system allows it to be deployed across large fleets of autonomous vehicles, ensuring comprehensive coverage and consistent protection. This is essential for the widespread adoption and trust in self-driving technologies.

Future Enhancements

As the field of autonomous driving continues to advance, there are opportunities to further refine and enhance the deep learning system, such as integrating it with other security measures, improving real-time response capabilities, and exploring the use of emerging technologies like edge computing.

The Crucial Role of CAN Bus in Self-Driving Cars

CAN (Controller Area Network) bus is the backbone of self-driving car technology, enabling seamless communication and coordination between the various electronic control units (ECUs) that monitor and control a vehicle's functions. As autonomous driving capabilities continue to advance, the importance of a robust and efficient CAN bus system cannot be overstated. This critical component ensures the safe and reliable operation of self-driving cars, allowing for the precise control and synchronization of essential systems, from steering and braking to engine management and environmental sensors.

10.3 Future Work

The rapid advancements in deep learning technology have revolutionized the field of autonomous vehicle development. As self-driving cars continue to evolve, the integration of deep learning algorithms is poised to play a pivotal role in shaping their future. By leveraging the power of deep neural networks, researchers and engineers are tackling the complex challenges associated with perceiving the environment, making real-time decisions, and ultimately, ensuring the safe and seamless operation of autonomous vehicles.

1. Perception and Sensing

One of the key challenges in autonomous vehicle development is creating robust perception systems that can accurately detect and classify objects, pedestrians, and road conditions in real-time. Deep learning-based computer vision and sensor fusion techniques are crucial in addressing these challenges, enabling self-driving cars to navigate complex environments safely.

2.Decision-Making and Planning

Autonomous vehicles must be equipped with advanced decision-making algorithms that can navigate complex traffic scenarios, anticipate potential hazards, and make split-second decisions to ensure the safety of passengers and other road users. Deep

learning models are instrumental in this process, helping self-driving cars adapt to dynamic situations and optimize their responses.

2. Ethical Considerations

As self-driving cars become more prevalent, ethical questions around liability, privacy, and the prioritization of human life in the event of an unavoidable accident have emerged. Deep learning models can be trained to incorporate ethical frameworks and decision-making principles, addressing these complex issues and building public trust in autonomous vehicle technology.

4.Sensor Integration

Autonomous vehicles rely on a complex network of sensors, including cameras, radar, and lidar, to perceive their environment and make real-time decisions. The CAN bus plays a crucial role in seamlessly integrating these sensors and ensuring reliable data exchange, enabling the vehicle's central control unit to process information and navigate safely.

5. Cybersecurity Challenges

As self-driving cars become more prevalent, the need for robust cybersecurity measures becomes paramount. The CAN bus, as a critical communication hub, must be fortified against cyber threats to protect the vehicle's systems and the safety of passengers. Addressing these security concerns is essential for building public trust and widespread adoption.

6.Regulatory Frameworks

The development and deployment of autonomous vehicles require the establishment of comprehensive regulatory frameworks. Policymakers must address issues such as liability, insurance, and traffic management to ensure a smooth transition and the safe integration of self-driving cars into existing transportation infrastructure.

References

Industrial & Automotive Buses CAN Bus : ENG. HASSAN Al-IRAQE

Attacks on the CAN Bus and their mitigations, Dr. Ken Tindell, CTO Canis Automotive Labs

Hindawi Journal of Advanced Transportation Volume 2020, Article ID 3035741, 9 pages https://doi.org/10.1155/2020/3035741

YouTube channels

Chat GPT

Mehmet Bozdal , Mohammad Samie , Ian jennions IVHM Centre, Cranfield University, Bedford, MK43 0AL, UK

1) https://onlinelibrary.wiley.com/doi/10.1111/exsy.12754
2) https://www.mdpi.com/1424-8220/19/24/5370
3) https://www.mdpi.com/1424-8220/21/12/3993
4) https://ocslab.hksecurity.net/Datasets/car-hacking-dataset
5) https://github.com/mehrotrasan16/CS581-CAN-DO-Project/blob/main/papers/AnomDet-CANBusMessages-DNN.pdf
6) https://0xsam.com/road/
7) https://arxiv.org/abs/1812.03243
8) https://ieeexplore.ieee.org/document/8997640
9) https://ieeexplore.ieee.org/document/8584950