Software Security in Autonomous Vehicles

Taylor Adams

***Abstract*—Autonomous vehicles are becoming more commonplace throughout the world. Vehicle manufacturers are focusing on integrating autonomous systems into their vehicles. While manufacturers race to compete in the blooming autonomous vehicle market, they must ensure sound security practices in the software developed for their autonomous vehicles. This paper aims to outline the importance of software security in autonomous vehicles, the challenges of creating autonomous vehicle software, and past attacks against vehicles and autonomous vehicle subsystems. This is followed by proposed methods on how to develop secure software in autonomous vehicles and a proposition for vehicle manufactures to utilize the Rust programming language. This paper also reviews the life cycle phases of secure vehicle software engineering. This includes analyzing, scanning, and testing vehicle software for vulnerabilities. Proposed frameworks for utilizing deep learning for intrusion detection of autonomous vehicle systems are also investigated.**

I. Introduction To Autonomous Vehicle Security

IN, recent years Autonomous Vehicles have become mainstream and accepted by society. In October 2020, Tesla released their Full Self-Driving Beta to the public. Today, Tesla's Full Self-Driving Beta software has been released to nearly 100,000 vehicles currently operating on public roads. While Tesla appears to be the market leader, in terms of the number of "autonomous vehicles on the road" most other vehicle manufacturers and many startup companies are following suit to develop autonomous systems. It is no longer a question of if fully autonomous vehicles will become viable, but when.

A. *Defining Autonomy*

Autonomous vehicles do not fit a single category. There are several levels of autonomous vehicles described by the U.S. Department of Transportation: Driver only, Driver assistance, Partial automation, Conditional automation, High automation, and Full automation [1]. Currently, production autonomous vehicles such as Tesla's vehicles sit between partial automation and conditional automation. Most modern vehicles have some form of autonomy between level 1 and level 2. Many modern vehicles include features like lane assist, parking assistance and many other features included in the driver assistance category. (For this paper autonomous vehicles and vehicles will be used interchangeably depending on the application of the method described.)

II. Importance of software security in autonomous vehicles

A. *The need for secure software in autonomous vehicles*

Software security behind autonomous vehicle systems is paramount to the technology's success. Vehicles have become

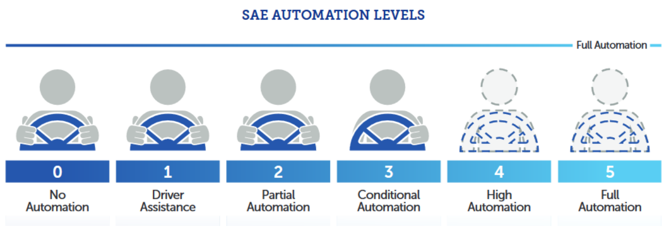


Fig. 1. Autonomy Levels [1]

more connected, with many featuring "over-the-air" updates when in proximity to a Wi-Fi connection or with an onboard cellular connection. This network connectivity also enables smartphone control over certain systems in a vehicle such as remote start and air conditioning controls. While this network functionality can provide end users with a more polished experience, it also opens additional vectors of attack where a malicious actor could tamper with vehicle operations remotely.

Another major software security concern in autonomous vehicles is dealing with the extremely large code bases autonomous vehicles use to operate. Jaguar Land Rover estimated it would take nearly 1 billion lines of code for level 5 autonomy (fully autonomous) in vehicles. Not only do software developers need to be concerned with autonomous systems, but also the security of the software used in the vehicle itself. In order for an autonomous system to have control over a vehicle the driving controls systems of the vehicle must be opened up to receive digital input. If an attacker can bypass an autonomous vehicle's software or inject a malicious payload into it, they could gain remote access to control the vehicle.

Autonomous software systems are still relatively new and are highly complex. At this point, most research surrounding autonomous vehicle systems is about functionality rather than security. Wyglinski, et al. on this topic have stated, "unlike complex networks such as the Internet, where the issue of security has been extensively researched and funded, security issues surrounding complex networks of autonomous automotive systems haven't been as readily studied. Moreover, these systems' security vulnerabilities are increasingly being discovered and exploited" [2].

III. Challenges of Creating Secure Software In Autonomous Vehicles

There are many design challenges when it comes to automotive security. In their paper "Towards a Secure Software Life-cycle for Autonomous Vehicles", Moukahal, et al. list the major development challenges of automotive software as: complex and large systems, various signals, outsourcing, security decay, open-source code, and maintenance ad incident

T. Adams is with The University of Alabama, Tuscaloosa, AL, 35401 USA e-mail: tkadams1@crimson.ua.edu.
Manuscript received December 01, 2022

response monitoring. Each of these concepts can be explained in greater depth:

A. *Complex and Large Systems*

It is well known automotive code bases are some of the largest programs in the world. Many contain over ”100 million lines of code, and future connected autonomous vehicles are expected to have more than 300 million lines of code” [3]. The size of these code bases creates a monumental challenge to ensure the overall security of the system. At the same time, vehicle manufacturers must remain competitive in the software features their vehicles offer. For autonomous vehicles, the point of competitive software becomes exponentially more important. This is especially true today, with production-level autonomous vehicles being in their infancy.

This inherent necessity for competitive software leads to a common mentality as seen throughout all software development: functionality over security. Vehicle manufacturers must also take extra care to assure security is a paramount consideration in their autonomous vehicle software production. If they fail, it is inevitable the security of their autonomous vehicle systems will be prone to vulnerabilities.

B. *Various Signals*

Various Signals describe the connected technologies implemented in vehicles and autonomous vehicles. This includes ”sensors, cameras, and radars enable vehicles to gather information” [3]. Each of these sensors can open up potential flaws in the overall vehicle system should an attack find a way to manipulate the sensor data being output by the sensor. Autonomous vehicle manufacturers must ensure all sensor packages they include in their vehicles are hardened from attack.

C. *Outsourcing*

Outsourcing, in this case, refers to the development of software where ”automotive subsystems require particular expertise and specialized skills” [3]. The manufacturer of a vehicle may not have the talent or time to create the software for a particular subsystem of their vehicle, leading them to outsource. This adds an additional level of complexity for security engineers. In this case, security engineers need to independently verify the security of the outsourced software received from a third party.

D. *Open-source Code*

Open-source code is vital to the automotive industry. Open source code allows for industry-wide standards, this typically offers the end user greater functionality and reduces software development costs. Open-source code poses a potential threat to the security of vehicles utilizing that code. By nature, open-source software allows attackers ”to identify vulnerabilities and possible entry points” [3] by reviewing the base code outright. One example of open-source automotive architecture commonly used in the automotive industry today is AUTOSAR. If an attacker managed to find a vulnerability in the AUTOSAR architecture, this could put any vehicles utilizing the architecture at risk of being exploited.

E. *Security Decay*

Security Decay is described by Moukahal, et al. as evolving security requirements during the development life cycle of a connected autonomous vehicle. The development of autonomous vehicle software is not a quick endeavor. It is likely to take years of development for a production system to be deployed. In this time, security requirements can change based on a variety of factors. These factors can include more stringent security requirements, new vulnerabilities being discovered, increased attacker capabilities, etc. These changes can necessitate new security reviews of existing software. Existing software can be legacy code inherited from past designs, creating additional challenges for security engineers to maintain the security of the overall code base.

F. *Maintenance and Incident response monitoring*

This security consideration describes the ongoing security improvements which must be made during the life cycle of the connected autonomous vehicle. New vehicles recently began receiving over-the-air updates from the manufacturer. Security engineers must ensure updates do not add vulnerabilities to the existing systems. Each update to the vehicle must be validated. This adds additional costs and strains manufacturer resources.

IV. Overview of possible attacks, exploits, and vulnerabilities in autonomous vehicle software

A. *Overview of primary sensors added to autonomous vehicles*

In order to understand any piece of software, a developer must first understand the inputs given to the software. This section will cover common sensor packages included in autonomous vehicles today. Sensor data is critical to vehicle systems’ functionality. Because sensor data is so critical, it can also be a primary threat vector for attackers to try to manipulate. As stated by Rasheed, et al. ”Sensors and communication links have great security and safety concerns as they can be attacked by an adversary to take the control of an autonomous vehicle by influencing their data” [4]. See figure 2 for an example of several sensors and computing devices located in autonomous vehicles.

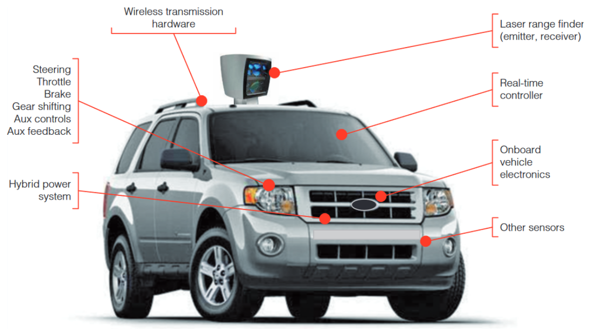


Fig. 2. Autonomous Automotive Systems [2]

The ”key sensors in autonomous vehicles: camera, radar, and lidar” [5].

1) Radar: Radar is typically used for ”different purposes like adaptive cruise control, blind spot warning collision warning, and collision avoidance” [5].

2) Cameras: Currently, computer vision is one of the primary technologies driving autonomous software. Computer vision relies on cameras positioned around the vehicle to navigate the vehicle in a safe and effective manner.

3) LiDAR: Lidar is an exteroceptive sensor used to determine the distance of surrounding objects. Lidar transmits a laser light which is reflected by surrounding objects. It records a time of flight which is used to calculate the distance of objects. This sensor data is used to create a 3D map of its surroundings.

B. Overview of commonly implemented Automotive Control systems

While sensors play a major role in the operation of modern vehicles, it is also important to understand the control systems that utilize sensor data. Automotive control systems are the heart of a vehicle. Essentially an automotive control system is a network of components that integrates all of a vehicle's sensors and systems together to function as a cohesive unit. The primary units that make up the automotive control system are electronic control units (ECU), control area networks (CAN), and Local Interconnect Networks (LIN). ECUs typically control the mechanical functions of vehicles. CANs allow for the transfer of data between ECU modules and act as the highway between different vehicle functions. LINs are ”a low-speed single-master network commonly used for door locks, climate controls, seat belts, sunroof, and mirror controls” [6].

In order for an attacker to gain control over the core functionality of a vehicle, typically, they must find a way to control the vehicle ECUs. Security engineers must ensure, at all costs that these ECU models and the interconnect networks between them are hardened from attack.

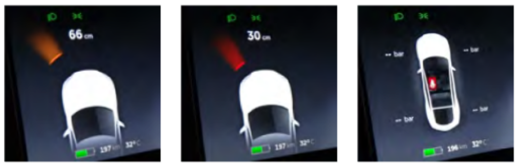
C. Sensor-based attacks

Sensors provide the core data on which an autonomous vehicle makes decisions. Without sensors, autonomous vehicles would not be possible. There would be no perception of the vehicle's surroundings. Because autonomous vehicles rely so heavily on their sensors, sensors become a sought-after target for attackers. Wyglinski, et al. illustrate an attack against AV sensors, ”By attacking the sensors, a malicious user can cause autonomous platform suicides and vandalism, as well as perform denial-of-service (DoS) attacks, and can even gain full control of the autonomous platform itself” [2].

Diving in deeper, more potential vulnerabilities begin to show in autonomous vehicles. Close proximity attacks to manipulate sensor data on passing autonomous vehicles can have dire consequences. An example of this type of attack is provided by Wyglinski, et al. ”For instance, a malicious user could just keep sending the vehicle false signals to disorient, hijack, or even restrict it to within a virtual fence” [2].

Below are examples of a few attacks conducted by researchers against autonomous vehicle sensors:

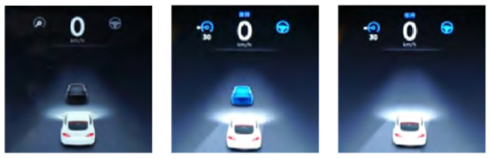
1) Attack on Ultrasonic Sensor: In 2016, several researchers conducted sensor jamming attacks against a Tesla Model S with autonomous vehicle features. From this attack the researchers claim, ”the security issues of sensors have not earned their due attention.” The goal of their attack was to, ”perform jamming and spoofing attacks, which caused the Tesla's blindness and malfunction” [7]. In one of their attacks the researchers were able to jam an ultrasonic sensor to make objects undetectable. They created their ultrasonic jammer with a low-cost Arduino unit, which would be readily available to real attackers. The goal of their attack was to continuously generate ultrasonic noise at an ultrasonic sensor. The researchers ended up successfully spoofing and jamming a Tesla Model S' ultrasonic sensor as can be seen in figure 2. This could force an autonomous vehicle to stop or cause it to get into a collision.



(a) Normal. (b) Spoofed. (c) Jammed.

Fig. 3. Ultrasonic Sensor on Model S jammed [7]

2) Attack on vehicle Radar: The same researchers who performed the attack on the ultrasonic sensor also performed an attack against a Tesla Model S' MRR radar sensor. This attack required more skill and resources to accomplish than the aforementioned ultrasonic sensor attack. However, the researchers were able to successfully launch a Radar jamming attack against the Tesla Model S' Radar by determining the frequency at which the radar operated. They then launched a jamming attack by spamming the radar with the same frequency band and a second attack with a sweeping frequency approach [7].



(a) Drive gear. (b) Autopilot. (c) Jammed.

Fig. 4. Successful Radar jamming attack against Tesla Model S [7]

3) Attack on Camera Systems: The third attack conducted by the researchers was a simulated blinding attack against the camera systems of an autonomous vehicle. This appears to have been a relatively straightforward attack compared to the radar and ultrasonic sensor attacks. For this blinding attack, the researchers used several different light sources: LED, visible laser, and infrared LEDs. The results are as follows: The LED light led to the blinding of the camera system. The laser light, pointed directly at the camera, ”lead to complete blindness for approximately 3 seconds” [7]. It also caused permanent damage to the sensor. The infrared light showed no effect on the camera sensor.

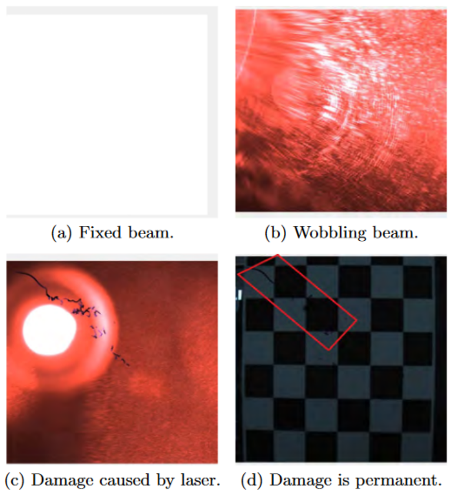


Fig. 5. Attacks against Camera systems in Autonomous Vehicles [7]

As demonstrated by Yan, et al. in reference [7] autonomous vehicle sensors are vulnerable to proximity attacks against them. Bad sensor input could cause software systems in autonomous vehicles to behave in unexpected ways. This data proves autonomous vehicles should not rely on any one sensor type alone where security is concerned. Redundancy must be built into autonomous vehicle software to account for sensor input manipulation by attackers. [2], [6]

D. *Network-based attacks*

As stated previously, vehicles have become more connected as time passes. This connectivity brought on many changes to the automotive industry with features such as, over-the-air updates, mobile phone applications for vehicles, and even subscription services for vehicle features such as heated seats. The push for network connectivity in vehicles opens up a host of new security issues new to the automotive industry. Evidence shows, ”attacks on autonomous vehicles will increasingly target vehicle-to-everything (V2X) technology related to communication rather than other simpler elements of the vehicles” [6]. By being networked to the outside world, vehicles open themselves up to an incredible amount of outside threats. Network based attacks have the added benefit to the attack in that they can be conducted remotely. This allows the hacker additional anonymity as they have the potential to be anywhere in the world conducting a network based attack against a vehicle.

Below is an example of an attack conducted against a connected vehicle through a diagnostic application:

Network based attacks can be devastating because they can allow an attacker to control a vehicle from anywhere. In [8] an attack was proposed by utilizing a malicious application connected to an OBD2 scan tool over bluetooth. The premise for this attack was: Attackers could upload a malicious self diagnostic application online, and have victims download the application. This application has a victim’s computer or smartphone connect to an OBD2 scanning tool. This scanning tool would be plugged into the vehicle’s diagnostic port by the victim, who would believe they were simply diagnosing problems on their own vehicle. This OBD2 tool would report information back to the malicious application downloaded by the user. This application would transfer the data to the attacker’s server. The attack could then issue CAN commands to the smartphone app, which would in turn send the commands to the OBD2 tool connected to the victim’s vehicle. The victims vehicle would receive the CAN command and execute on it since the CAN protocol has limited built in protections.

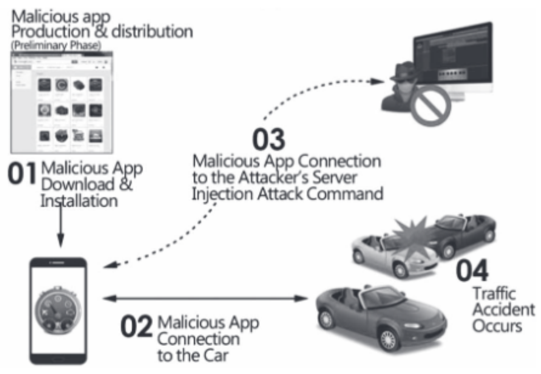

Malicious app Production & distribution (Preliminary Phase)
01 Malicious App Download & Installation
03 Malicious App Connection to the Attacker's Server Injection Attack Command
04 Traffic Accident Occurs
02 Malicious App Connection to the Car

Fig. 6. Network based attack in an autonomous vehicle [8]

When performed on an actual vehicle, the above attack was able to cause a physical malfunction of the test vehicle. While the attack does involve some social engineering. It displays a lack of software protection for the diagnostic port on many vehicles, as well as a lack of software security implementation in the CAN standard.

E. *Hardware-based exploits and attacks*

Electronic control units (ECU) are some of the most important control systems in modern vehicles. Many ECUs handle the core functionality of the vehicle such as the transmission, traction control, airbags, brakes and the engine to name a few. As of 2013, it was estimated, ”a typical car today includes 70 to 100 microprocessor-based ECUs” [2]. This means 70-100 different hardware systems which an attacker could attempt to exploit with physical access to a vehicle. The number of ECUs in vehicles has likely increased today and will continue to do so as vehicles receive additional features and sensors.

V. Review of the 2015 Jeep Hack

In 2015, Security Researchers Dr. Charlie Miller and Chris Valasek managed to remotely access a 2014 Jeep Cherokee. The result of this attack was the researchers were able to gain full control of the vehicle. They were able to send commands over the internet to disable the engine, turn on fans, change the radio station and volume as well as send various other commands to the vehicle. At the same time, they were able to completely override driver inputs.

A. What systems were targeted

Miller and Valasek managed this attack by finding a vulnerability in the Uconnect System. They were able to accomplish this task because both vehicle Controller Area Network (CAN) buses were connected to the head unit. This allowed them access to the vehicle's ECU modules.

B. How The attack unfolded

1) Identifying possible attack surfaces: The researchers began by analyzing the different types of attack surfaces the vehicle offered which were likely to be vulnerable. This included both physical attack surfaces as well as remote attack surfaces such as Bluetooth, remote key-less entry/start, passive anti-theft system, tire pressure monitoring system, cellular system, and the applications on the infotainment system. The researchers ultimately settled on attacking the cellular system (Uconnect). This system was determined to have the greatest possible attack surface.

2) Identifying Vulnerabilities Inside the Uconnect Infotainment System: The researchers initially looked at the operating system the Uconnect system ran on as well as underlying file system. By doing this, they discovered several files of note including boot scripts which could be modified if they were able to jailbreak one of the head units.

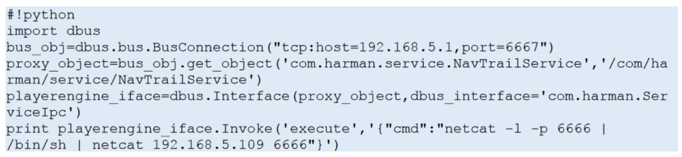
Miller and Valasek then moved on to looking at the on board WiFi equipped in their 2014 Jeep Cherokee. They discovered the Jeep wifi system had several vulnerabilities such as allowing users to require no password for the network or using WE for their WiFi security. The researchers also discovered a design flaw in the way the WPA2 password for the Jeep's WiFi was implemented. By disassembling the 'WifiSvc' binary the researchers determined the WPA2 default password was generated based on a function of the epoch time, likely on the head units first boot. This is a big issue in cryptography as an attacker could estimate the date the head unit was produced and drastically reduce the number of possibilities the password could be. As expected, the researchers were able to determine their Jeep's WiFi password very quickly through a brute force approach.

The next attack area the researchers investigated was what ports were open on the Jeep's Wifi network by running a port scan on the default gateway. This lead them to another software vulnerability. The researchers were able to connect anonymously to port 6667 using the telnet protocol. This protocol has been depreciated due to being insecure, but this doesn't really affect anything if there was no password to gain access to the port to begin with. The researchers were able to utilize this open port to run a debugger on the D-BUS system. This lead them to find several D-Bus services allowing them to directly interact with the Uconnect Infotainment system.

3) Jailbreaking the Uconnect Infotainment System: Jailbreaking the Uconnect Infotainment system was the next step. Jailbreaking the system was not necessary to perform the attack but, "the jailbreak was integral to figuring out how to explore the head unit and move laterally" [9].

The jailbreak was able to be accomplished by multiple methods. Miller and Valasek were able to tell the head unit to update with a USB stick holding an ISO. The firmware on the head unit would attempt to update. The head unit system reboots after verifying the contents of the "USB update". During this time, the researchers were able to swap the USB in the system with one containing the jailbroken firmware. For version 14.05.03 of the head unit, a bug was found allowing removal of the integrity check of the ISO verification process. Both of these methods allowed for unsigned code to run on the system. Using these methods allowed persistent changes to be made to the head unit.

4) Exploiting the D-BUS Service: As mentioned in one of the previous sections, it is possible to telnet into the D-BUS services anonymously. These services are used for inter-process communication. Miller and Valasek stated, "We don't believe that the D-Bus service should be exposed, so is not surprising that it is possible to exploit it to run attacker-supplied code" [9]. The researchers did not need to utilize any command injection as they found a service designed to execute arbitrary shell commands which remained in the production system. Figure 7 shows code the researchers produced which allowed them to gain root access to an unmodified head unit.



```
#!python
import dbus
bus_obj=dbus.bus.BusConnection("tcp:host=192.168.5.1,port=6667")
proxy_object=bus_obj.get_object('com.harman.service.NavTrailService','/com/ha
rman/service/NavTrailService')
playerengine_iface=dbus.Interface(proxy_object,dbus_interface='com.harman.Ser
viceIpc')
print playerengine_iface.Invoke('execute','{"cmd":"netcat -l -p 6666 |
/bin/sh | netcat 192.168.5.109 6666"}')
```

Fig. 7. Script to root access to D-BUS [9]

5) Utilizing access to the head unit: At this point in the attack Miller and Valasek already gained control of many vehicle features. Essentially they could control anything controlled by the head unit which includes the HVAC system, Radio, Display and Knobs. As part of this attack, they were able to disable the physical controls for these systems as well leaving drivers and passengers of the vehicle unable to physically counter an attacker.

6) Exploiting the vehicle from more distant ranges: Up to this point, Miller and Valasek had been using the Jeep Cherokee's WiFi network to conduct the attack. This had a time limitation of needing to be inside the range of the vehicle's WiFi network as well as the victim having paid for the in vehicle WiFi network. The next step taken was to determine a way to take control over the vehicle from longer ranges. It was found that the vehicle utilized Sprint's network to connect to the internet. Each time the vehicle was restarted it would get a new IP address on Sprint's network within a specific IP range assigned by the carrier. They discovered "the D-Bus service was bound to the same port (6667) on the cellular interface" [9] meaning their existing attack would still work. The caveat to this was they needed to be on the same Sprint network the Jeep was connected to. The research pair found that Sprint did not block traffic between devices on their network. This allowed the researchers to utilize a smartphone to connect to Sprint's network to remotely access their Jeep from anywhere in the United States.

Once on Sprint's network 15 other types of vehicles were discovered to have port 6667 open over Sprint's network. This would allow anyone to compromise any vulnerable vehicles as long as both target and attacker were connected to Sprint's network.

7) Controlling more aspects of the vehicle: At this point in the attack, Miller and Valasek had control over the infotainment system, but not critical vehicle features such as steering, brakes, and the engine. In order to get more control of the vehicle the researchers used their newly achieved access to move laterally through the vehicle systems.

CAN communications in the 2014 Jeep Cherokee are handled by the V850ES/FJ3 chip. This chip can send CAN messages which is connected to the OMAP chip in the head unit. The researchers found that while the OMAP chip cannot send CAN messages itself, it is connected to the V850 chip which can send the messages. Not only was the V850 chip connected to the OMAP chip, but the OMAP chip had privilege to update the V850 chip with new firmware.

Miller and Valasek found the file used for updating the IOC application firmware. They began reverse engineering the file to understand the makeup of the firmware and how to update it. The next step was "to reverse engineer and modify the IOC application firmware to add code to accept commands and forward them to the CAN bus" [9]. After much time was spent reverse engineering the IOC firmware, the researchers were eventually able to re-flash the IOC with modified firmware due to the fact it had no cryptographic signatures to verify firmware authenticity. The researchers were also able to locate the command needed to send out CAN commands to the system.

8) One final hurdle: Once Miller and Valasek had their modified firmware they needed a way to remotely flash the firmware to the V850 chip. The update system was "only designed to perform the upgrade from a USB stick" [9].

In order to accomplish this task they had to put the V850 chip in bootloader mode and the OMAP chip in update mode. Once they put it and the OMAP chip in the proper update modes, the next step was to overwrite a program called 'hd'. They replaced 'hd' with their modified firmware which allowed the V850 chip to be updated. The researchers then made the 'fs/mmc0' partition writable and then restarted the system in bootloader mode.

9) Results of the Research: After all this, Miller and Valasek had full control of the vehicle. This control includes turning the windshield wipers on and off, killing the engine, controlling the vehicle steering, and even disabling the brakes of the vehicle. Ultimately, "The recall that resulted from this research affected 1.4 million vehicles" [9].

10) Key Software security takeaways from this research: It is evident many security mistakes were made by Fiat Chrysler in the development of their vehicles between 2014 and 2015. Many of these errors were careless, like leaving the D-BUS port exposed to the WiFi network as well as the cellular network, allowing telnet connections to the port, no authentication when remotely communicating with the port, and leaving D-BUS services in production systems which allow arbitrary execution of code. Other factors like Sprint allowing communication with any device on its network may have been an external decision for functionality over security by Sprint. On top of these security errors, there was also the design flaw implemented by Fiat Chrysler of having the head unit connected to both CAN buses in the vehicle without proper security testing. From a security standpoint, the CAN buses should have been completely separated from any hardware that interacted with the internet. The 2015 Jeep Hack demonstrated how crucial software security is for vehicles.

VI. Developing secure vehicle software

In the above sections, this paper delineates the challenges of creating secure software in autonomous vehicles, possible attacks, and exploits, as well as several exploits discovered in the past. The following sections will focus on how to create secure software for autonomous vehicles.

The current, most popular, frameworks utilized to develop autonomous vehicle software are the "Microsoft Security Development Lifecycle (MSDL) and Secure Software Development Framework (SSDF)" [3]. However, these frameworks do not directly account for the challenges in autonomous vehicle software development. Some of the issues not addressed by these development frameworks are the additional security testing requirements, heterogeneous subsystems with different threat-level exposure, and the consideration of the extended time it takes to produce autonomous vehicle software.

A. Secure Vehicle Software Engineering

To address these issues, Moukahal, et al. in [3] created the Secure Vehicle Software Engineering (SVSE) development framework. This framework consists of "nine phases divided into four stages: planning, development, testing, and operation" [3]. The 9 phases are cybersecurity: planning, requirements, design, assurance planning, implementation, component testing, integration testing, resilience verification, and maintenance [3]. (This section will only cover cybersecurity planning, requirements, design, assurance planning, and implementation.)

1) Cybersecurity Planning: Cybersecurity planning is the first step in SVSE. It involves Identifying cybersecurity objectives, threat analysis and risk assessment, and identifying cybersecurity assurance levels. This, in part, covers the preliminary steps to actually developing the vehicle software. In order to successfully secure vehicle software a comprehensive plan must be established for things like ECU protection. This is where one of the failures in the 2015 Jeep attack occurred as the ECUs were not properly protected. Cybersecurity planning also involves mapping out possible attack surfaces a vehicle may have. Also involved is defining cybersecurity assurance levels for each component of a vehicle.

2) Cybersecurity Requirements: Establishing the cybersecurity requirements of vehicle software is the second step in SVSE. It involves developing and prioritizing the requirements for the primary development team as well as suppliers producing software for the vehicle. The requirements are created based on the results from the first step of Cybersecurity planning. All requirements must be defined at length to ensure coverage is met during the review phase. Once the requirements are established it is important to prioritize the requirements. Due to the complexity of vehicle systems, it is vital to focus on high-priority security objectives and work down to the lower security objectives.

3) Cybersecurity Design: Once the cybersecurity requirements have been established, design can begin. The design stage includes a review of cybersecurity requirements, secure design measures, design measures to detect cyber attacks, and a design review. Design measures taken by the development team must ensure cybersecurity principles are followed. Various cybersecurity measures must be utilized, ”this includes various security methods like authentication, authorization, data encryption, and logging” [3]. Vehicle systems must have some form of intrusion detection system. This should be included in the software design of the software for each system. Once completed, the design should be reviewed to ensure all security goals are met.

4) Cybersecurity Assurance Planning: Cybersecurity assurance planning involves assurance planning for suppliers, a plan for cybersecurity monitoring and incident response planning. Vehicle manufacturers must ensure each of their suppliers and subcontractors can and will conform to the cybersecurity plan, requirements, and design established by the manufacturer. Without total compliance, security holes can open up in the software systems developed by these suppliers or subcontractors. Evidence of this is seen again in the 2015 Jeep Hack where the Uconnect infotainment system was the attacker’s point of entry to the entire vehicle.

5) Cybersecurity Implementation: Cybersecurity Implementation is where the software really begins to be written in vehicle systems. It involves following security code practices, vulnerability and threat analysis, and prioritizing security testing. It is critical developers follow guidelines and best practices established by the industry during the software development such as AUTOSAR. As outlined in [3] the complexity of vehicle system software opens up a wide margin for vulnerabilities to appear. Vulnerability and threat analysis can also be conducted to locate what vulnerabilities are in the software at the current state. This can include finding common patterns of vulnerabilities occurring in the software. These common vulnerabilities can be made known to the development team in order to mitigate the risk of recurrence throughout the remainder of the software.

B. Automotive software inherits the risk of the C programming language

Most vehicle software today is written in C and C++. C and C++ aim to be as efficient or nearly as efficient as assembly language. In order to accomplish this efficiency, accessing memory outside of an array bounds is not checked as this would add additional overhead to programs. This opens up code to potential vulnerabilities such as buffer overflows and other memory space violation vulnerabilities. Memory safety can be accomplished in C and C++ however this task is placed solely on the developer. Standards such as Motor Industry Software Reliability Association (MIRSA) C have been introduced in an attempt to mitigate developer error when using C and C++ in the automotive industry. Unfortunately, adding guidelines does not ensure developers stickily adhere to them. Even when the guidelines are mandatory, humans are fallible and mistakes can easily be made when working on complex software.

In [10] it was found during the time frame of 2008-2014, ”nearly 23 percent of all severe vulnerabilities were buffer errors and 72 percent of buffer errors were severe”. This trend has continued. Currently it is estimated, ”that 60-70 percent of critical vulnerabilities in Chrome, Microsoft products and in other large critical systems owe to memory safety vulnerabilities” [11]. Another type of attack that the C programming language can be vulnerable to is the format string attack. As seen in figure 8, format string attacks were much less common than many other attack types, but had a high rating on severity. This illustrates the security trade off made when utilizing the C or C++ programming languages.

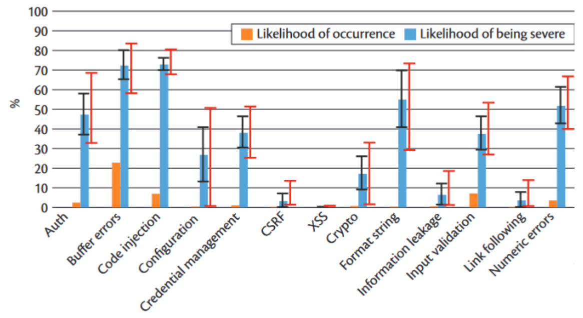


Fig. 8. Vulnerability likelihood and severity [10]

Reviewing processes, like static code analysis, do help in mitigating risk, but a better solution to the issue is utilizing a programming language built with memory safety principles in mind.

C. How Rust fixes the memory security issues of C

The overall goals of Rust is “to be fast, low-level, and type- and memory-safe [11]. Rust also emphasizes performance, type safety and concurrency. Instead of allowing users to manage memory (unsafe) or using a garbage collector (inefficient), Rust manages memory by using the ”Ownership” concept. This ”Ownership” concept gives Rust a more efficient memory management system than utilizing garbage collection. At the same time it remains memory safe, but allows its developers to switch between safe and unsafe modes. By default, Rust is in safe mode. This bars null pointers, dangling pointers, or data races from occurring. When using the unsafe keywords developers can avoid these restrictions thereby adding flexibility. This is why Rust is an optimal language choice for applications where high performance is needed.

1) Memory management in Rust: Rust utilizes a memory management technique called Ownership. As described in [12] Ownership operates based on three rules:

- Each value in Rust has an owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Once a value is owned by its variable it can be borrowed to be utilized as immutable references or borrowed only once as a mutable reference. The Rust compiler also keeps track of these references to determine the lifetime of the value. When a value in (safe) Rust is referenced it is guaranteed to point to a valid location in memory due to the Ownership system. This Ownership and borrower system is much more efficient than traditional garbage collection found in other programming languages like Java.

Rust also gives developers the ability to enter ”unsafe blocks”. This enables ”developers to deactivate some, but not all, of the borrow checker and other Rust safety checks” [11]. When these features are disabled memory management can be turned back to the user, similar to memory management in C and C++. The side effect of this is it requires developers to be more thoughtful when manually managing memory as ”unsafe blocks” have to be explicitly declared.

D. Utilizing the Rust programming language for secure software vehicle development

Developers of the Linux Kernel have begun to implement Rust in place of C. In [13] researchers ”introduced the use of the Rust programming language in kernel development, where the safety features of the Rust language were leveraged to prevent the introduction of memory bugs or synchronization bugs when writing kernel code” [13]. Rust being included in the Linux kernel is strong evidence of the language’s utility in complex systems. In their experimentation they found ”the performance overhead of the Rust kernel modules is on par with the C kernel modules” [13]. The performance of Rust also enables it to be used on vehicle subsystems which may have low power processes or memory/storage constraints that necessitated the use of C or C++. Rust can be a viable alternative programming language for these subsystems as well, further increasing Rust’s utility in automotive systems.

The Rust programming language has many features which make it ideal for automotive software development. The memory safety aspects of Rust can assist developers in creating safer software for autonomous vehicles. The safe and unsafe block concept can assist developers in debugging the massive code bases of autonomous vehicles by allowing them to focus on the areas in the code declared as unsafe. This has the potential to save massive amounts of time in the testing phase of vehicle software development. The functionality and efficiency of Rust also lend it to be an optimal candidate for use in vehicle subsystems. Vehicle manufacturers should strongly consider implementing Rust into their vehicle code stack.

VII. Analysing, Scanning, and testing software in autonomous vehicles

A. Software testing challenges in Autonomous Vehicles

There are several key challenges to testing vehicle software. These challenges are highlighted in [14] as System Complexity, Outsourcing, Input and Output fluctuation, and Test-Bed complexity.

1) System Complexity and Outsourcing: System complexity and outsourcing were discussed earlier in this paper. Just as these two principles are difficult in the development of autonomous vehicle software systems, they also increase the difficulty in testing this software. Due to this complexity it is essential to utilize an automated testing method. Attempting to debug an entire automotive code base by hand would be an insurmountable feat for even the largest of development teams.

2) Input and Output fluctuation: Input and output fluctuation account for the many sensor inputs an autonomous vehicle receives as well as the output decision the software makes based on the input. When it comes to input received by an autonomous vehicle, ”Assessing the set of all possible external environmental data is an intractable problem” [14]. Essentially every second the vehicle operates, it is likely to receive unique sensor data.

3) Test Bed Complexity: Test-Bed complexity is difficult as ”security engineers must guarantee that the vehicle system is developed and designed following cybersecurity requirements of vehicle standards like AUTOSAR, ISO 26262, and ISO/SAE 21434 standard” [14]. The challenge of deploying real-world testing systems, such as actively having a prototype vehicle on the road also exists. It is infeasible to build enough test vehicles to ensure software security alone. Many companies rely on simulation software to supplement their security testing. It is also difficult to develop the simulation software as there are so many environments in which a vehicle should be able to operate. Ultimately, the test bed or test suite must guarantee random events will not cause a security vulnerability or catastrophic error in the vehicle.

B. Locating vulnerabilities in software using automation

All software is difficult to debug. System complexity only increases this difficulty. As previously mentioned in this paper, vehicle software systems contain some of the most complex code in the world. This complexity is exacerbated as more autonomous features get added to vehicles. Due to the complexity and depth found in autonomous vehicle systems, it is vital to use automated testing software to locate vulnerabilities in the code repository. A few automated methods which can assist security engineers in scanning vehicle code for vulnerabilities are static code analysis, dynamic program analysis, and vulnerability scanning.

1) Static code analysis: The goal of static code analysis is to identify vulnerabilities in the source code of software. It does so by reviewing the source code provided to it in a static manner to see if there are design patterns matching know vulnerabilities. There is no execution of the target software during static code analysis.

While static analysis cannot find all software vulnerabilities, it is able to assist in the discovery of several types of coding errors as described by [15]:

- Unreachable Code
- Unused Code
- Variable boundary overflow
- Code-level defects not typically found in dynamic program analysis

2) Dynamic program analysis: Unlike static code analysis, Dynamic program analysis involves executing the code to be tested. Dynamic program analysis aims ”to look for dangerous conditions such as memory errors, concurrency errors, and crashes” [14]. It does this by looking at what is happening in the target code, while the target code is running. Dynamic program analysis tools can be created or tailored to scan specific vehicle software and autonomous software systems to increase the odds of finding vulnerabilities in those systems.

3) Vulnerability Scanning: Vulnerability scanning allows security engineers to scan software for already known vulnerabilities. The goal of vulnerability scanning is to find explicitly defined vulnerabilities that may be common to the type of software being scanned. In the automotive industry, many vehicle manufacturers contribute data to the Automotive Information Sharing and Analysis Center [14]. This allows vehicle automotive manufacturers to share data on common vulnerabilities facing the company while minimizing the risk of disclosure of the intellectual property or proprietary code. The shared data provided can be utilized to create vulnerability scanning programs that many manufacturers can run on their software.

4) Fuzz Testing: Fuzz testing is a valuable tool to find vulnerabilities in many pieces of software. Fuzz testing is the action of running invalid or unexpected inputs to a piece of software to see if the software breaks. Generally, Fuzz testing is automated and checks the output of the target program to see if any errors were generated. Fuzz testing can be valuable when testing automotive systems due to the many subsystems connected to the main system. This allows developers to learn how the primary system will react when fed malformed data from one of its subsystems.

C. Common Types of Fuzz Testing and its use in autonomous vehicle software

As mentioned previously, fuzz testing plays a critical role in ensuring the security of programs from malicious inputs. There are three different types of testing which include: white box testing, grey box testing, and black box testing. The white, grey, and black box testing methods are not limited to fuzzing alone and can be utilized with other testing methods like penetration testing. However, this section will primarily focus on their use in fuzzing.

1) White-Box Fuzz Testing: White-Box testing is not widely used in the automotive industry. This is because it is the most time-intensive of the three testing methods. It also assumes an attacker has access to the source code of the vehicle, as well as most internal documentation on the systems being tested. In terms of security alone, white box testing would be beneficial to run, however, the cost of performing this type of testing is prohibitive given the size of automotive code bases.

2) Black-Box Fuzz Testing: In Black-Box fuzz testing it is assumed the testing program knows nothing about the software being tested besides known ”good input”. The ”good input” can be utilized as a seed value in which to generate other variations of possible inputs. Black box fuzzing requires the testing program to send its malformed strings from outside the target program as input. This is a key differentiating factor from white-box fuzzing.

However, Black-Box fuzz testing is not an end-all solution. It likely leaves some security gaps untested as it ”cannot guarantee good coverage and a thorough evaluation of the system” [14].

3) Grey-Box Fuzz Testing: The Grey-Box fuzz testing approach is a compromise between black-box fuzz testing and white-box fuzz testing. Grey-Box fuzz testing allows for more relevant fuzzing tests to be conducted without the additional overhead found in white-box fuzzing. This makes grey-box fuzz testing suitable for automotive systems while providing additional security coverage over the scope of the system. This testing type addresses three major software security testing challenges: system complexity, outsourcing, and input/output fluctuation [14].

D. A fuzz testing approach tailored to autonomous vehicle systems - Vulnerability-oriented Fuzz Testing

Vulnerability-oriented Fuzz testing is a grey-box fuzz testing method proposed by Moukhaal, et al. The goal of vulnerability-oriented fuzz testing is to utilize, ”security vulnerability metrics designed particularly for connected and autonomous vehicles to direct and prioritize the fuzz testing toward the most vulnerable components” [14]. This essentially works by utilizing a vulnerability score that is generated by the modeled vulnerability engine. A weight is added to each function based on the vulnerability level perceived by the engine. The proposed method also takes into account the number of times a function is utilized in its weight metric. If a given function receives a high-priority rating, it is placed in a high-priority bucket to be focused on for the fuzz testing. This allows the system to pay more attention to testing functions deemed most vulnerable by weight. If a function is deemed low priority it is either disregarded or placed into a low-priority bucket to be tested towards the end of the fuzz testing.

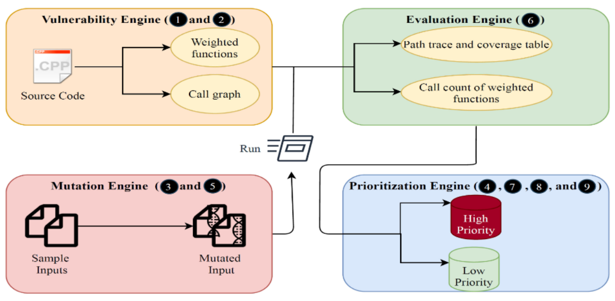

Vulnerability Engine (1 and 2)
.CPP
Source Code
Weighted functions
Call graph
Evaluation Engine (6)
Path trace and coverage table
Call count of weighted functions
Run
Mutation Engine (3 and 5)
Sample Inputs
Mutated Input
Prioritization Engine (4, 7, 8, and 9)
High Priority
Low Priority

Fig. 9. Vulnerability-oriented fuzz testing framework [14]

There are four main engines utilized in Vulnerability-oriented Fuzz Testing as proposed in [14]: the Vulnerability Engine, the Evaluation Engine, the Mutation Engine, and the Prioritization Engine. The vulnerability engine establishes a function’s weight. The mutation engine creates valid and invalid input to run against these functions. The evaluation engine looks at the output generated by running the program with the mutated engine’s input. It then ”assesses the usefulness of test cases” [14]. The prioritization engine utilizes the output of the evaluation engine to create a priority queue based on the evaluation of a function.

1) Deeper Dive - Vulnerability Engine: This particular approach to vulnerability-oriented fuzz testing was designed specifically to be utilized with automotive system source code. Some of the additional factors the vulnerability engine takes into account are: ECU coupling risk, communication risk, complexity risk, input and output risk, and history of security issues. Each of these risk types are accounted for in the weight value generated by the vulnerability engine. This allows for greater resource concentration during the testing of vehicle software.

The core of vulnerability-oriented fuzz testing is the vulnerability engine. The distribution of weights based on the above factors allows for autonomous vehicle systems to be robustly tested. While critical to the implementation, the other engines rely on the vulnerability engine to create the weights. These weights contribute significantly towards prioritizing the most vulnerable functions.

VIII. A modern software security approach: Utilizing deep learning for intrusion detection

Software security in autonomous vehicles can be taken a step further from ensuring the base security of the software itself. Deep Learning approaches can be used to mitigate the damage from an attacker who is inside the software or has some way to manipulate the data in the autonomous vehicle. Research conducted in [4], [16] proves deep learning can effectively mitigate data manipulation from an attack attempting to send false data in place of sensor data. The former research focuses on an attack manipulating sensor data, while the latter research focuses on monitoring communication and messages on the CAN bus. Both of these methods were able to identify malicious data and filter it out to a relevant degree. In examinations of both deep learning approaches the test simulation vehicles were able to operate normally while the intrusion detection system found and mitigated the attack.

This research [4] simulated an attack against four intra-vehicle resources: the camera system, the radar sensor, smart roadside units, and inter-vehicle beaconing. In this attack, it was assumed the attacker was able to inject malicious data in an effort to give a false reading on objects in the surrounding environment. This data would be sent to the proposed deep reinforcement learning algorithm. The vehicle utilized this deep reinforcement learning algorithm to determine the optimal distance from objects surrounding the vehicle. The attacker's goal was to manipulate the deep reinforcement learning algorithm to mislead the vehicle by "introducing deviation between the autonomous vehicles i.e. impacting optimal safe distance spacing while autonomous vehicle tries to minimize this deviation [4]." The researchers conducted several simulation cases using the model they created. In each of these simulations, they were able to successfully nullify the attack.

It is impossible to ensure the total security of a software system. Especially when the software is as complex as autonomous vehicles. Therefore, it is vitally important to utilize intrusion detection systems as a final line of defense in autonomous vehicle software. Utilizing these methods can thwart an attacker's attempts to manipulate an autonomous vehicle's decision-making ability. The above example is only one case out of many where deep learning models can be applied to help reduce the damage from attackers who infiltrate or attempt to manipulate autonomous vehicle software.

IX. Discussion and Conclusion

This paper conducted an overview of the importance, challenges, development, and testing of software in autonomous vehicles. It has also covered consequences of providing software with vulnerabilities as well as adversarial countermeasures that can be implemented for attack mitigation.

Outlined below are the most critical points for development of software security in autonomous vehicles moving forward:

1) Risk Mitigation for sensor based attacks: Autonomous vehicles are already being produced even while many sit at level 1 autonomy, many already have features which necessitate electronic control of critical systems. As vehicles move to become more sensor reliant, the issue of software security in autonomous vehicles will only grow. This is especially true for sensor based attacks. Manufacturers must take care to implement intrusion detection systems, and be able to compensate for manipulated sensor data.

2) Risk Mitigation for the increase in connected vehicles on the road: Many vehicles today are moving to a model where they are constantly connected with the outside world. Many vehicles receive over-the-air updates through on board cellular service. However, opening up vehicles to the internet, creates significant vulnerability risk as seen in the 2015 Jeep attack. With vehicles becoming more connected, manufactures must take care to ensure best practices are followed in the code utilized in their vehicles.

3) Utilize the Rust programming language in vehicle software development: The Rust programming language offers many additional security features compared to C and C++. At the same time Rust is able to maintain similar performance. Rust also adds the flexibility of unsafe blocks where developers are able to have near the same level of memory control as in C and C++. These unsafe blocks ensure developers are more conscious when writing potential memory vulnerabilities into software. This has the side effect of easier debugging, as well as being able to target unsafe blocks during testing. These features make Rust an idea programming language candidate for autonomous vehicle software. Vehicle manufactures should strongly consider adding Rust to their code stack. Even if this addition happens slowly over time, it is likely to pay dividends in the future.

4) Utilize the Secure Vehicle Software Engineering development framework: The Secure Vehicle Software engineering development framework proposed in [3] provides an excellent software development framework tailored to the development of autonomous vehicles. This framework could be utilized on its own, or enhanced by vehicle manufactures to create an industry wide development standard specific to autonomous vehicles.

5) Ensure adequate testing of vehicle software: Testing is critical to mitigating vulnerability risk in vehicle software. The testing covered in this paper includes static code analysis, dynamic code analysis, vulnerability scanning, and fuzz testing. Each of these testing methods cover different aspects of possible vulnerabilities in code. When used together they can be a powerful tool for vulnerability mitigation throughout any range of software. While there are many more testing methods which can be covered, these represent testing methods which should absolutely be run on any vehicle software systems currently in development.

The recommendations delineated above will help vehicle manufactures and software developers ensure greater software security in autonomous vehicles.

REFERENCES

[1] "Automated Driving Systems: A Vision for Safety," p. 36.

[2] A. M. Wyglinski, X. Huang, T. Padir, L. Lai, T. R. Eisenbarth, and K. Venkatasubramanian, "Security of Autonomous Systems Employing Embedded Computing and Sensors," *IEEE Micro*, vol. 33, no. 1, pp. 80–86, Jan. 2013, conference Name: IEEE Micro.

[3] L. J. Moukahal, M. Zulkernine, and M. Soukup, "Towards a Secure Software Lifecycle for Autonomous Vehicles," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2021, pp. 371–377.

[4] I. Rasheed, F. Hu, and L. Zhang, "Deep reinforcement learning approach for autonomous vehicle systems for maintaining security and safety using LSTM-GAN," *Vehicular Communications*, vol. 26, p. 100266, Dec. 2020.

[5] J. Kocić, N. Jovičić, and V. Drndarević, "Sensors and Sensor Fusion in Autonomous Vehicles," in *2018 26th Telecommunications Forum (FOR)*, Nov. 2018, pp. 420–425.

[6] K. Kim, J. S. Kim, S. Jeong, J.-H. Park, and H. K. Kim, "Cybersecurity for autonomous vehicles: Review of attacks and defense," *Computers & Security*, vol. 103, p. 102150, Apr. 2021.

[7] C. Yan, W. Xu, and J. Liu, "Can You Trust Autonomous Vehicles: Contactless Attacks against Sensors of Self-driving Vehicle," p. 13.

[8] S. Woo, H. J. Jo, and D. H. Lee, "A Practical Wireless Attack on the Connected Car and Security Protocol for In-Vehicle CAN," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 993–1006, Apr. 2015, conference Name: IEEE Transactions on Intelligent Transportation Systems.

[9] D. C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," p. 91.

[10] H. Homaei and H. R. Shahriari, "Seven Years of Software Vulnerabilities: The Ebb and Flow," *IEEE Security & Privacy*, vol. 15, no. 1, pp. 58–65, Jan. 2017, conference Name: IEEE Security & Privacy.

[11] K. R. Fulton and A. Chan, "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study," p. 21.

[12] "What is Ownership? - The Rust Programming Language."

[13] S.-F. Chen and Y.-S. Wu, "Linux Kernel Module Development with Rust," in *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, Jun. 2022, pp. 1–2.

[14] L. J. Moukahal, M. Zulkernine, and M. Soukup, "Vulnerability-Oriented Fuzz Testing for Connected Autonomous Vehicle Systems," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1422–1437, Dec. 2021, conference Name: IEEE Transactions on Reliability.

[15] A. Imparato, R. R. Maietta, S. Scala, and V. Vacca, "A Comparative Study of Static Analysis Tools for AUTOSAR Automotive Software Components Development," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2017, pp. 65–68.

[16] I. A. Khan, N. Moustafa, D. Pi, W. Haider, B. Li, and A. Jolfaei, "An Enhanced Multi-Stage Deep Learning Framework for Detecting Malicious Activities From Autonomous Vehicles," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–10, 2021, conference Name: IEEE Transactions on Intelligent Transportation Systems.