One solution of providing ADAS data to the IVI domain via the Android Java service

Dušan Stanišić
Department of Computing and Control Engineering
Faculty of Technical Sciences,
University of Novi Sad
Novi Sad, Serbia
Dusan.Stanisic@rt-rk.com

Marija Antić
Department of Computing and Control Engineering
Faculty of Technical Sciences,
University of Novi Sad
Novi Sad, Serbia
marijantic@gmail.com

Dušan Kenjić
Department of Computing and Control Engineering
Faculty of Technical Sciences,
University of Novi Sad
Novi Sad, Serbia
Dusan.Kenjic@rt-rk.com

Luka Bilač
Department of Computing and Control Engineering
Faculty of Technical Sciences,
University of Novi Sad
Novi Sad, Serbia
Luka.Bilac@rt-rk.com

2023 Zooming Innovation in Consumer Technologies Conference (ZINC) | 979-8-3503-4772-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/ZINC58345.2023.10174158

Abstract* — In recent years, manufacturers are working parallel in developing and integrating two in-vehicle systems, an Advanced Driver Assistance System (ADAS) and In-Vehicle Infotainment (IVI). ADAS's purpose is to increase safety and aid the driver while the vehicle is in motion, by using various sensors to perceive the vehicle surrounding or the driver's errors and respond accordingly. Consumer applications in the IVI domain would significantly benefit from accessing such vehicle-specific data. Unfortunately, due to active vehicle system architecture requirements, the IVI domain cannot achieve communication with the ADAS by following traditional signal-oriented mechanisms. In addition, Android paved its way into the IVI domain as the most common Operating System because of its high flexibility and customization, but without being adapted for communication with ADAS. This paper offers a solution for enabling Android applications to access ADAS data via Java services. Data is procured over GENIVI company's SOME/IP-based communication middleware, integrated within the services. Communication performance is measured, evaluated, and limitations are identified.

***Keywords* — ADAS, IVI, Android, SOME/IP, Java service, IPC, CommonAPI**

I. INTRODUCTION

The backbone of modern vehicles is composed of two separate disjunct systems, the Advanced Driver Assistance System (ADAS) and the In-Vehicle Infotainment (IVI) system. As stated in [1], ADAS represents the integration of programming algorithms and hardware aimed at providing a heightened level of safety and security for drivers. Usually, it performs complex algorithms for vehicular environmental perception and autonomous driving to assist drivers, since 94% of all car accidents are caused by human error [2]. Consequently, ADAS must possess sufficient awareness to make informed decisions in dangerous situations, avoid potential fatalities, and minimize environmental harm.

On the other hand, the In-Vehicle Infotainment system, as the name suggests, should primarily focus on providing audio-visual entertainment for the user [3]. Additionally, it should furnish relevant information about the car systems, such as tire pressure, fluid levels, vehicle surroundings, navigation systems, and features such as Bluetooth connectivity, Wi-Fi, and other general information.

Establishing communication between the ADAS and the IVI system has numerous benefits. Firstly, the user can take advantage of the multi-purpose hardware within the ADAS system, using its data as if this hardware was part of the IVI domain, thereby reducing overall vehicle hardware complexity and cost while preserving functionality. Secondly, the data produced by the algorithms within the ADAS system will be readily accessible to the user through various Android applications. As mentioned in [4], since the creation of Android Automotive in 2017, a full-fledged extension of the regular Android operating system, Android has become the most widely adopted operating system for IVI systems, providing a codebase for vehicle manufacturers for developing their versions of the operating system and integrate third-party applications.

The automotive industry utilizes communication protocols, including CAN, Ethernet, FlexRay, MOST, Bluetooth, and Wi-Fi, to communicate between the ADAS and IVI systems. Emerging protocols like SOME/IP and DDS offer higher data transfer rates with lower latency. Despite these advancements, integrating ADAS and IVI systems poses challenges such as ensuring secure and reliable data transfer, meeting low latency requirements for real-time ADAS data, and ensuring compatibility between different communication protocols.

This paper proposes a solution for establishing communication between the ADAS and IVI domains within a vehicle, utilizing Java Android services that use the GENIVI communication middleware, CommonAPI. This solution employs the SOME/IP protocol for communication with the ADAS domain.

The organization of this paper is as follows: Section II provides an overview of the relevant concepts related to Android services and CommonAPI. Section III details the implementation of the proposed solution. Section IV evaluates the solution through measurements and results, and Section V presents the conclusion of the study.

This research (paper) has been supported by the Ministry of Science, Technological Development and Innovation through project no. 451-03-47/2023-01/200156 "Innovative scientific and artistic research from the FTS (activity) domain".

II. Related Works

A. CommonAPI

CommonAPI is a standardized Application Programming Interface (API) developed by the company GENIVI. It is designed for the creation of distributed applications based on a client-server paradigm, independent of the underlying Inter-Process Communication (IPC) stack. This allows for freedom in selecting the communication middleware mechanism (such as D-Bus or SOME/IP) while still utilizing the same API for application development.

CommonAPI stack consists of two parts as it is depicted in Figure 1 [5]:

- CommonAPI Core - middleware-independent part (CommonAPI Core source code, generated FIDL files)
- CommonAPI Bindings - middleware-specific part (CommonAPI Bindings source code, generated FDEPL files)

B. Android Services

Services are one of the Android application components besides Content Providers, Broadcast Receivers, and Activities that have no Graphical User Interface (GUI), and represent an entry point through which the system or a user can enter the application.

The Service's primary function is to execute long-running tasks or to perform inter-process communication (IPC) with clients [6]. Services can be classified into three categories and behave quite differently (See Figure 2):

- Background service – Perform operations that are not noticed directly by the user and will be interrupted by the application's termination.
- Foreground service – Perform operations that are noticeable by the user and will continue to execute even when the application is terminated. Also, they must display a notification so that the user is aware of executing the task.
- Bound service – Provide a type of client-server paradigm where a client can be bound to the service and interact with the service by sending requests and receiving results by using Remote Procedure Calls (RPC).

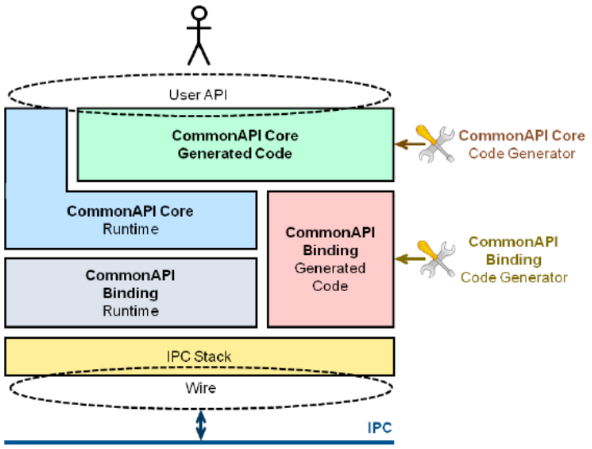

Figure 1. CommonAPI Stack Overview

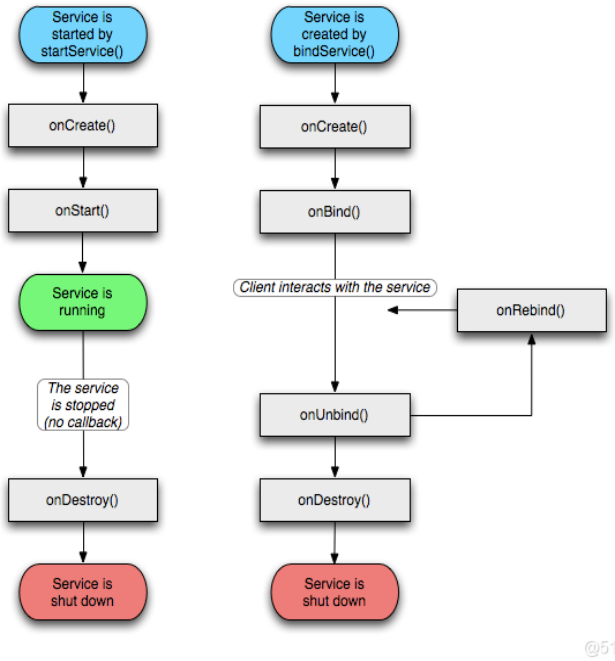

Figure 2. Service Lifecycle

C. Binder

The binder serves as a mechanism for inter-process communication between the client and the service. The architecture of the binder driver, which is responsible for establishing communication, is abstracted at the core (kernel) layer to simplify implementation for the user [7].

Considering the indefinite nature of the term binder, we will enumerate some of its most understood interpretations:

- Binder object – Represents an instance of a class that has implemented the binder interface.
- Binder protocol - The Binder middleware uses a very low-level protocol to communicate with the driver.
- Binder interface - A Binder interface is a well-defined set of methods, properties, and events that a Binder can implement.
- Binder token - A numeric value that uniquely identifies a binder.

In the context of Android, inter-process communication is restricted, and a process cannot access any system resources or proprietary data of other processes. For two processes to communicate, the complex objects must be broken down into basic data types through a process known as serialization. This process of breaking down objects is referred to as marshaling, and each object must implement the Parcelable interface.

On the client side, the serialized objects received through marshaling are reconstituted to their original form through the process of unmarshalling or deserialization. The Binder takes part in all of this.

Writing the code for marshaling and unmarshalling can be time-consuming and tedious, hence it is recommended to automate it using Android Interface Definition Language (AIDL) files. AIDL is the preferred method for achieving Remote Procedure Calls (RPC) in client-service communication in Android. As depicted in Figure 3, the use of Android Interface Definition Language (AIDL) generates a two-part

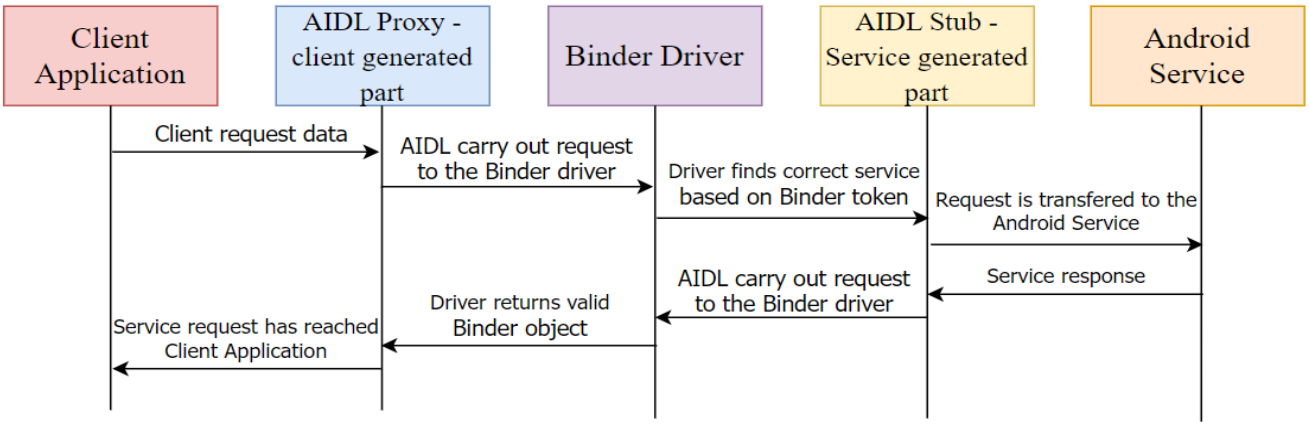

Figure 3. Communication between client and service

code consisting of a Stub for the service and a Proxy for the client. These components facilitate the entire communication process between two remote processes.

D. JNI

The Java Native Interface (JNI) provides a bridge between the Java programming language and C/C++ code, enabling interaction and communication between the two platforms [8].

Java's portability is one of its defining characteristics, enabled by its ability to convert source code into bytecode, executed by the Java Virtual Machine (JVM), and then into machine code. This allows for Java code to be executed on any machine running an instance of the JVM, regardless of the architecture. However, in some cases, it may be necessary to integrate native code into Java code, such as when utilizing drivers for specific hardware, improving performance, reducing resource consumption in certain operations, or leveraging existing shared libraries written in native code. The JNI allows for the integration of C/C++ code into Java code, as shown in Figure 4.

E. SELinux Policy

The Android operating system employs a modified version of the security-enhanced Linux protection system, which imposes mandatory access control (MAC) on all processes, regardless of their root privileges [9]. This implementation enhances the security of applications and services while mitigating the risk of malicious software. The Security Enhanced Linux (SELinux) operates based on the automatic denial of access principle, effectively prohibiting any action that has not been explicitly authorized.

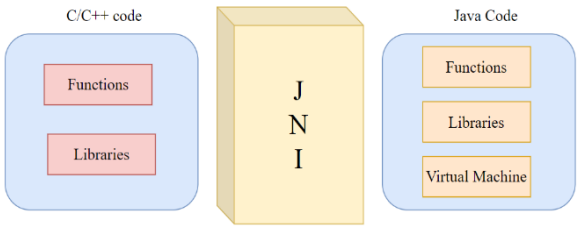

Figure 4. Relationship between Native and Java code

SELinux operates in two modes:

- Permissive mode - SELinux will log unauthorized actions attempted by the user, however, they will still be able to proceed with their work.
- SELinux will log unauthorized actions attempted by the user and immediately halt the current process.

As a general practice, operating the device or emulator in the Enforcing mode is recommended as much as possible. This enables the detection of errors and bugs over time, allowing them to be addressed before the product is made available to the market. Additionally, a device that is not in the Enforcing mode cannot pass the Android Compatibility Test Suite (CTS). CTS is a series of tests that evaluate the software's compatibility with hardware and established software handling conventions, ensuring that security is not compromised, and standards are upheld.

III. SOLUTION OVERVIEW

In Figure 5, the software architecture of the solution for ADAS and IVI inter-domain communication is presented.

A. Mapping types of SOME/IP communication onto the Android paradigm

There are 3 types of communication described in SOME/IP [10] :

- Methods – The client initiates the communication, sending a request message to the service and returning a response message.
- Events – The service initiates communication, and the client subscribes to the service, listening for incoming events.
- Attributes - They allow the access or modification of a single type of variable either primitive or complex. By default, the user is allowed to get, set, or be notified of the attribute's value. We would not use a setter since changing the value is not viable. The reasons are safety-oriented since Android must not have an impact on the safety-critical ADAS domain.

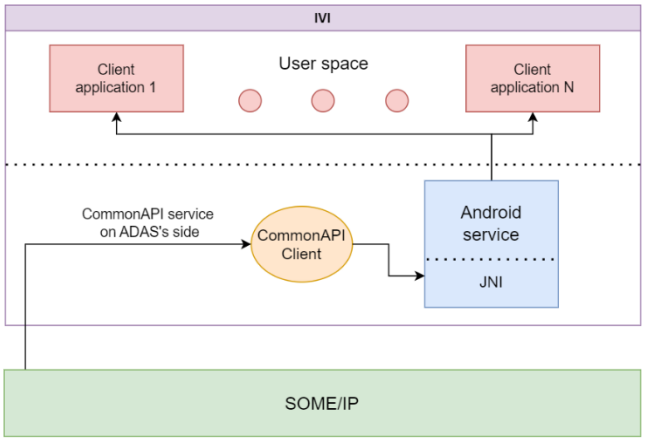

Figure 5. Software architecture of cross-domain communication

SOME/IP communication is specified in FIDL files, which are utilized by CommonAPI to generate Stubs for the service side and Proxies for the client side. As the AIDL file operates similarly, mapping the FIDL paradigm to AIDL is necessary.

1) Mapping methods

Android is designed to initiate communication from the client side, making mapping SOME/IP methods straightforward. When a client calls an Android method, it invokes the CommonAPI method, which transmits information to the ADAS side. As a result, the function description in the AIDL file can match the corresponding function described in the FIDL file, using the same arguments and return type as shown in Figure 6.

2) Mapping Events

In the case of events, communication is initiated by the SOME/IP service, and the client is informed of any changes that occur within the system or a change in the value of a software component. As this type of communication does not exist or is unsupported in Android, resolving this issue necessitates the use of two AIDL files. As described in [11], the first AIDL file contains a function that is called by the service, but its implementation is located on the client side. The second AIDL file has a function that takes as an argument an instance of the first AIDL file, used for registering clients and monitoring upcoming events. This is visualized in Figure 7.

3) Mapping Fields

In SOME/IP communication, attributes can be represented in AIDL as either method (if viewed as getters and setters) or an event (if viewed as notifiers). In FIDL files, users can specify three flags for attributes: "readonly", "noRead", and "noSubscriptions". The "readonly" flag disables the ability to set the value, the "noRead" flag prohibits reading the value but still allows for setting a new one, and the "noSubscriptions" flag prohibits the notifier option.

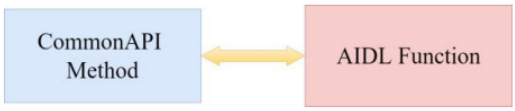

Figure 6. Mapping SOME/IP method

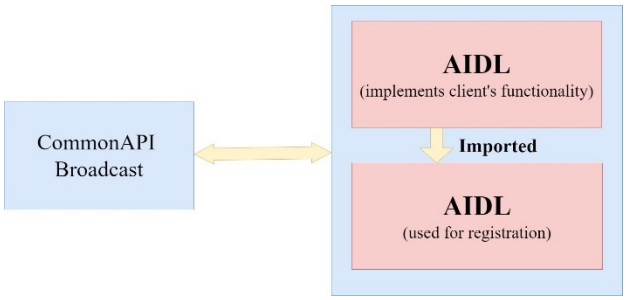

Figure 7. Mapping SOME/IP event

B. Communication with CommonAPI service

The CommonAPI service of GENIVI is implemented and written in C++. To access the CommonAPI service, the user must integrate the CommonAPI client into their Android service in the form of dynamic libraries. The CommonAPI client instantiates a shared pointer to the CommonAPI Proxy class, which enables direct calls to the CommonAPI Service. To accomplish this, the use of Java Native Interface (JNI) is mandatory, as it enables cross-language operations such as calling native code from Java or vice versa.

JNI functions are declared within the Java class of the Android service, and corresponding declarations must be present in a header file for proper usage. Due to its complex syntax, the header file is typically generated using a Java compiler with a special flag, rather than being written manually. Table 1 shows Java types and their native counterparts [12] :

TABLE I JAVA TYPES AND THEIR NATIVE COUNTERPARTS

Java type	Native type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	decimal 32 bits
double	jdouble	decimal 64 bits
Class<E>	jclass	Class
String	jstring	String
Complex type	jobject	Complex type
Primitive array	j<primitive>Array	Primitive array
Complex array	jobjectArray	Complex array

C. Implementation of Android service

The Android service operates as a client to the software components of ADAS and as a service to other Android applications. To serve Android applications, it is necessary to expose binder objects to clients by extending the abstract Stub class and implementing its functions.

Binder objects can be propagated in two ways: the first is when the service behaves as a bound type of service, and the binder is passed when the client explicitly binds to it. This approach is limited to Java-only clients. The second way is by adding a binder to a list of system services using the ServiceManager class. This class is part of the system framework and unsuitable for public use, but it is the preferred approach in this case, as it is available to native and Java clients.

D. Safety Overview

In addition to ensuring proper functionality and performance, it is crucial to prioritize safety and security. Exposing the binder object through the ServiceManager class poses a potential risk, as malicious clients with knowledge of the binder's name can access its object and exploit AIDL functions. To mitigate this, each AIDL function is executed in a separate thread, reducing the strain on the main thread, and includes two layers of protection, which verify the client's signature and grant necessary permissions. If these conditions are not met, an exception will be thrown.

Furthermore, the service implements the LinkToDeath interface for every registered client listening to events, linking the binder objects directly to the client's lifecycle. If the client terminates unexpectedly, the service will remove it from the collection, effectively unregistering it from future events.

Finally, several modifications have been made to the SELinux policy, including adjustments to the Android domain files and the explicit addition of previously ungranted actions, enabling the service to function optimally in enforcing mode.

IV. Evaluation

The solution was tested on a Qualcomm SA8155P board [13] equipped with the Android 12 operating system. The testing focused on three distinct use cases: one method and two events.

The first use case involved the calculation of the current tire pressure by an algorithm on the ADAS side and the subsequent delivery of this information to the Android service. The Android service then facilitated forwarding this information to the client's Android application upon request. To measure the performance of this method, the necessary measurements were conducted entirely on the client side, both before and after the function call.

The evaluation of the second and third use cases involves monitoring the driver's state of vigilance and detecting speed limit traffic signs, respectively. The algorithms responsible for these tasks are implemented on the ADAS side, and the results are delivered to the Android service for further processing and propagation to the clients. The measurement of these events is performed in three distinct phases:

1. On the ADAS side before and after the call of the function which sends the data to the subscribed clients.
2. Measuring Round Trip Time (RTT), divided by two.
3. On the Android service side, from the function that is subscribed in JNI when the event occurs, measured time is transferred via the parameter to the clients.

Table 2 presents the minimum and maximum recorded time values, alongside the mean value and standard deviation, for approximately 1000 measurements.

TABLE II Measured time for method and events

	Average [ms]	Standard deviation [ms]	Worst case time [ms]
Tire Pressure	39.48 ms	6.32	57.49
Driver Monitoring	8.54 ms	0.88	10.08
Sign Detection	8.32 ms	0.96	9.89

Table 3 displays the resource utilization (CPU and RAM) of the Android service both in its idle state and while servicing clients.

TABLE III Resource consumption

	Android service
CPU Usage (IDLE)	1.2%
CPU Usage (Working)	44%
RAM Usage	1.5%

As the results presented in Table 2 demonstrate that the method's execution time is acceptable and relevant for the non-critical use case when compared to the execution time measurements of the native service method mentioned in the paper[11]. Nevertheless, for critical use cases, a native service may be more suitable. Additionally, the execution time for events is consistent and acceptable, even for more critical cases.

In contrast, the data analysis of Table 3 reveals that the service exerts a noticeable workload on the system while serving multiple clients.

V. Conclusion

The proposed solution entails the deployment of an Android Java service that employs the extensions of CommonAPI to facilitate communication with the ADAS domain. Through this approach, an Android-based IVI system can access vehicle information from another domain, thereby supporting the development of novel functionalities.

The objective of Android services is to maintain consistent performance over prolonged periods of time, without succumbing to collisions or other malfunctions, and with good response times. Although we have incurred somewhat high resource consumption, we have accomplished this goal in a satisfactory manner.

In future work, we strive to implement a Priority Scheduling Algorithm to sort registered clients based on preferred criteria, thereby enhancing Service performance. Furthermore, we plan to introduce Kotlin-based Android services, with a focus on leveraging the latest language feature, coroutines.

References

[1] K. Omerovic, J. Janjatovic, M. Milosevic and T. Maruna, "Supporting sensor fusion in next-generation android In-Vehicle Infotainment units," *2016 IEEE 6th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 2016, pp. 187-189, doi: 10.1109/ICCE-Berlin.2016.7684751.

[2] A. Dang and M. Gupta, "An Evaluation of IT Next Gen–Unmanned Vehicle," *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, 2020, pp. 1-4, doi: 10.1109/ic-ETITE47903.2020.350.

[3] P. Sivakumar, R. S. Sandhya Devi, A. Neeraja Lakshmi, B. VinothKumar and B. Vinod, "Automotive Grade Linux Software Architecture for Automotive Infotainment System," *2020 International Conference on Inventive Computation Technologies (ICICT)*, 2020, pp. 391-395, doi: 10.1109/ICICT48043.2020.9112556.

[4] G. Macario, M. Torchiano and M. Violante, "An in-vehicle infotainment software architecture based on google android," *2009 IEEE International Symposium on Industrial Embedded Systems*, 2009, pp. 257-260, doi: 10.1109/SIES.2009.5196223.

[5] COVESA, “CommonAPI C++ User Guide”, Available: https://usermanual.wiki/Document/CommonAPICppUserGuide.1126244679/html

[6] “Android for developers”, Available: https://developer.android.com/

[7] Schreiber, Thorsten. "Android binder." A shorter, more general work, but good for an overview of Binder. http://www. nds. rub. de/media/attachments/files/2012/03/binder. pdf (2011).

[8] C. Qian, X. Luo, Y. Shao and A. T. S. Chan, "On Tracking Information Flows through JNI in Android Applications," *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 180-191, doi: 10.1109/DSN.2014.30.

[9] R. Gove, "V3SPA: A visual analysis, exploration, and diffing tool for SELinux and SEAndroid security policies," 2016 IEEE Symposium on Visualization for Cyber Security (VizSec), 2016, pp. 1-8, doi: 10.1109/VIZSEC.2016.7739580.

[10] M. Vujanić, N. Trifunović, I. Kaštelan and B. Kovačević, "Bitroute SOME/IP: Implementation of a Scalable and Service Oriented Communication Middleware," *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2022, pp. 1426-1429, doi: 10.23919/MIPRO55190.2022.9803691.

[11] L. Bilac, D. Stanisic, D. Kenjic and M. Antic, "One Solution of an Android In-Vehicle Infotainment Service for Communication with Advanced Driver Assistance System," *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2022, pp. 1420-1425, doi: 10.23919/MIPRO55190.2022.9803790.

[12] *“JNI Types and Data Structures” Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html.*

[13] Qualcomm Technologies Inc.,”SM8150 Linux Android Software User Manual”, Dec 2016