Wireless Pers Commun
DOI 10.1007/s11277-016-3618-9

Vulnerabilities of Android OS-Based Telematics System

Hyo Jin Jo[1] · Wonsuk Choi[1] · Seoung Yeop Na[1] · Samuel Woo[1] · Dong Hoon Lee[1]

Abstract Intelligent vehicle technologies have been developed rapidly. Modern vehicles include many Electronic Control Units (ECUs) and in-vehicle networks. While these technologies offer accurate vehicle control and increase the convenience and safety of drivers, their vulnerabilities also have been analyzed and exploited. Nevertheless, open platforms, such as the Android OS, have been introduced into vehicle systems without careful consideration about security issues. In this paper, we indicate the security problems of an Android OS-based telematics system. Our target device's firmware is offered on a public Web site and is easily analyzed using public analysis tools. This means that our analysis methods are more scalable and practical than previous ones for remote attacks that require difficult analysis skills, such as signal processing and reverse engineering. We also found that the device allows malicious firmware to be updated because of a problem related to misuse of certificates. Furthermore, we conducted attack experiments using a real vehicle.

Keywords Telematics communication · Controller Area Network · Android · Smart vehicle · Open platform

✉ Dong Hoon Lee
donghlee@korea.ac.kr

Hyo Jin Jo
hyojinjo86@gmail.com

Wonsuk Choi
beb0396@naver.com

Seoung Yeop Na
sy_na@korea.ac.kr

Samuel Woo
samuelwoo@korea.ac.kr

[1] Graduate School of Information Security, Korea University, Seoul, South Korea

Published online: 26 August 2016

1 Introduction

Modern vehicles have adopted multiple electronic subsystems, from powertrain control, body control, and comport control systems to infotainment systems. In particular, Electronic Control Units (ECUs) play a key role in accurate vehicle controls. A modern vehicle includes tens to hundreds of ECUs, which are connected through in-vehicle networks, such as the Controller Area Network (CAN), Local Interconnect Network (LIN), and Media Oriented System Transport (MOST) [30]. Among these in-vehicle networks, CAN is used for time-critical engine control, powertrain control, and safety control subsystems. However, the vulnerabilities of CAN were widely reported in many studies [17, 18, 27, 28]. With physical access to a vehicle, an attacker can send bogus data and modify the firmware of ECUs to control the vehicle [17, 20]. Recently, vehicle owners have begun to face growing security threats because various wireless channels are now connected to in-vehicle networks [29].

In [24], the Tire Pressure Monitoring System (TPMS) can be exploited to send a false "low tire pressure" warning. Furthermore, Checkoway et al. [4] analyzed the possible attack surfaces of modern automotives, which compromised many I/O channels of a target vehicle connected to external networks: the diagnostics equipment, media player, hands-free Bluetooth and telematics system. C. Miller et al. [19] also performed remote attacks on a telematics system. However, these works focused mainly on a specific environment, e.g., aqLink[1] protocol, and required the attackers to have good analysis skills, e.g., reversing engineer, signal processing, for finding and exploiting the vulnerabilities of external channels. Thus, the possibilities and scalability of these attacks may be low.

This narrow range of potential risks related to vehicle hacking has not prevented various open platforms for providing a strong eco-system from being adopted in the infotainment and telematics systems of vehicles. MirrorLink [21], as an interoperability standard, provides integration of smart phones and the infotainment systems. Several vehicle manufactures offer public APIs to allow third party developers to implement various types of applications, for example, GM's API [12] and Ford's OpenXC [25]. Moreover, Google, Audi, Honda, Hyundai, Kia, and NVIDIA formed the Open Automotive Alliance (OAA) to integrate the Android OS in vehicles' infotainment systems [22]. Google also released Android Auto to extend Android applications such that they can be implemented in-vehicle console systems [1]. According to REUTERS [10], Google will develop its next version of the Android OS, which will be built directly into vehicles. Honda, Hyundai, and Kia already produced an Android OS-based telematics system [14, 16]. GM's next-generation entertainment and navigation systems will be based on the Android OS [13].

However, these open systems can cause dangerous situation. In general, it is easy for an attacker to compromise open systems, because rich information about the systems' vulnerabilities is publicly announced through the Internet. In particular, the widely known vulnerabilities of the Android OS can affect the security of modern automotive systems.

In this study, we analyzed the vulnerabilities of Android OS-based telematics systems and exploited them. Our objective is to show that careful consideration of security is the most important factor for enjoying the many advantages of built-in open systems in vehicles.

[1] The protocol is used by GM's OnStar (infotainment system) to provide connectivity to external networks.

Our contributions are as follows.

- Remote attacks on vehicles investigated in previous studies focused mainly on a specific environment, e.g., a specific protocol or devices, and required that attackers have good analysis skills, e.g., reversing engineer, signal processing, and fuzzing tests on CAN packets. Thus, the scalability of remote attacks on vehicles may be low. However, our proposed remote attack procedures are more scalable and practical than those used in previous studies because of the built-in open platforms in vehicles.
- We analyzed the vulnerabilities of an Android OS-based telematics system using free public analysis tools. There is no need to require difficult analysis skills, such as signal processing, finding a debugging port, and using debugging tools. Our results show that anyone familiar with the Android OS and Android development environment can compromise an Android OS-based telematics system relatively easily.
- We demonstrate our results using a real vehicle with an Android OS-based telematics system. We were able to perform unauthorized remote control of functions, e.g., door unlock and GPS trace, of the target vehicle by using the vulnerabilities we analyzed.

2 Related Works

With the development of IT technologies, vehicles now also include many ECUs to provide accurate control, safety systems, infotainment systems, and so on. At the same time, modern vehicles are exposed to various security threats. In [18, 27] and [28], the possible security problems in CAN were indicated and cryptographic methods for securing vehicles were proposed. In the study reported in [17] and [20], it was found in experiments using real vehicles that their systems can be compromised; the critical control CAN packets of the body control module (BCM), Engine Control Module (ECM), and Electronic Brake Control Module (EBCM) were obtained. In [24], the TPMS signals were manipulated by spoofing erroneous tire pressure readings. However, these works have the limited attack distance because [17] and [20] require a wired line to transmit malicious commands and TPMS sensors of [24] has the limited communication range.

To show remote attacks on vehicles practically possible, [4, 11, 19] and [29] were proposed. In [11] and [29], the authors used dongles supporting communications between CAN and outside networks. Even though these dongles allow an adversary to send malicious commands to target vehicles, the use of dongles restricts attack possibilities. In [4], various automotive attack surfaces were analyzed. In particular, the authors performed a long-range wireless attack using the vulnerabilities of GM's OnStar telematics service. For identification of the vulnerabilities of the target telematics device, a communication protocol, i.e., aqLink, and the telematics device were reverse engineered using signal processing, finding debugging flags, and implementing a software modem. They also found vulnerabilities in the authentication process used in remote controls, that is, reinitialization of a random number and a bug that accepts incorrect responses. In Miller et al. [19], found vulnerabilities of Jeep's Uconnect system and exploited them. However, the attacks proposed in both studies can be performed in specific environments; [4] and [19] focused on GM's OnStar service using the aqLink protocol and the Jeep's Uconnect using Sprint,[2] respectively. Particularly, [19] needs the Sprint's femtocell device to send malicious commands to the target vehicles.

[2] The telecommunications company in US.

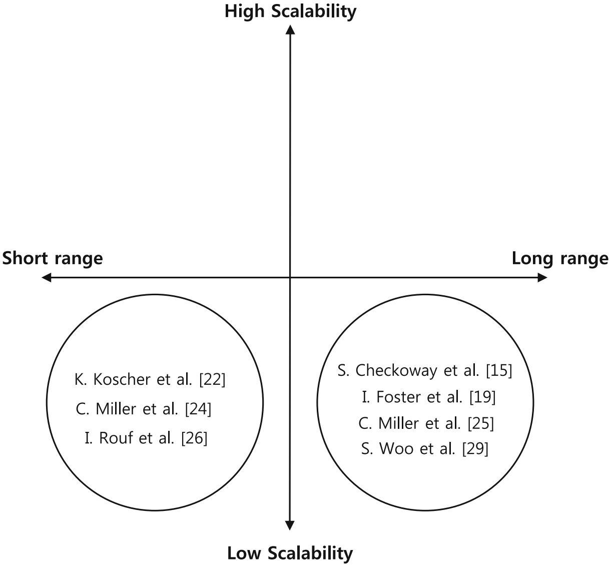


Fig. 1 Classification of vehicle attacks

Fig. 1 classifies the previous works in terms of "Attack Scalability[3]" and "Attack Distance".[4] Since there is no attack method satisfying long range and high scalability as shown in Fig. 1, various open platforms have been introduced into modern vehicles without careful consideration of the security problems. Thus, we analyzed the Android OS-based telematics system to show more scalable attack on vehicles than previous works and to propose desirable directions of the open platform using the Android OS built-in in vehicles

3 Background

3.1 IT-Vehicle Convergence

3.1.1 Controller Area Networks

Controller Area Network (CAN) is a serial data communication protocol that supports high-reliability control. CAN adopts a multi-master broadcast bus system. It was developed by Robert Bosch and has been used as a de facto standard of in-vehicle networks (ISO 11898-1). There are two versions of the CAN protocol: CAN 2.0A (11 bits identifier) and CAN 2.0B (11 bits or 29 bits identifier). Table. 1 shows the data frame format of CAN 2.0B, which is selected for in-vehicle networks. In the CAN protocol, if a sender ECU broadcasts its own data (data field) on the bus using its own ID (identifier field), receiver ECUs receive data selectively after filtering the ID field of the CAN frames. According to [7], CAN can be used for both high (500 Kbits/s) and low speed networks (100 or 150 Kbits/s). While the former is used to connect the chassis and transmission control, e.g.,

[3] High scalability indicates the attack method could be widely used in a general environment (e.g., easy to analyze, no additional device, analysis of public algorithms), and low scalability is opposite to the former.

[4] It indicates attack distance between victims and adversaries.

Table 1 Data frame format of CAN 2.0B (11 bits identifier)

SOF field	Identifier (ID) field	Control field	Date field	CRC field	ACK field	EOF field
1bit	11bit	6bit	0–64bit	16bit	1bit	1bit

engine control and anti-lock braking system controls, the latter is used to connect the body and comport modules, e.g., door and seat control.

3.1.2 Telematics System

Recently, the development of telematics services that support wide area communication, e.g., 3G/4G, has been rapid. The connectivity with external networks provides vehicles with various functions, e.g., remote vehicle control and diagnostics.

Various services exist: BMW's ConnectedDrive, Ford's SYNK, GM's OnStar, Hyundai's Blue Link, and Kia's UVO. For example, a vehicle equipped with GM's OnStar provides an authorized driver with diagnostics of vehicle problems, emergency calls by monitoring crash sensors, remote door unlock, and tracking of the vehicle's location. In addition, the OnStar telematics device offers the "Slowdown" service, purportedly effected by stopping the fuel injection to the engine, to facilitate recovery in the case of theft.

3.2 Android

3.2.1 Android Application

Android applications are implemented in Java and are executed on the Dalvik VM.[5] They are distributed in the form of Android application package (APK) files. An APK contains all the files required for application execution. In general, the file types included in the APK are META-INF directory, lib directory, res directory, assets directory, AndroidManifest.xml, classes. dex, and resources.arsc. In particular, AndroidManifest.xml contains components, permissions, the API version, and so on for the application and classes.dex is a Dalvik EXecutable (DEX) file that runs on Dalvik VM.

In comparison, system libraries and pre-installed applications, called system applications, contain an ODEX (Optimized DEX) file not a DEX file. According to [9], DEX files are converted to ODEX files for performance optimization.

3.2.2 Over the Air Firmware Updates

Over the air (OTA) firmware updates are used to remove bugs or improve functionalities using a wireless channel. According to [9], OTA updates of Android are performed by downloading an OTA firmware file, i.e., update.zip, and rebooting a recovery mode.[6] It is

[5] The Java virtual machine of Android is called Dalvik. Android runs Dalvik bytecode produced from Java bytecode. Thus, Android applications have a .dex file or .odex file, which can be executed on the Dalvik virtual machine. (They do not have .class files).

[6] According to [8], Android devices typically have two different modes: a normal boot mode and a recovery mode. The recovery mode is a minimal Linux-based OS that includes a kernel and RAM disk and is used for system update processes.

also possible to manually update OTA firmware files. In general, Android devices have hardware buttons, e.g., volume up/down, power, and so on, for entering the recovery mode, which supports firmware updates through the Android debug bridge tool (ADB) or SD cards. The recovery mode allows only firmware update files with valid digital signatures to be updated; in general, an `update.zip` file is signed with a private key of an Android device manufacture.

3.3 Firmware Modification Attack

The objective of a firmware modification attack is to inject malware into an embedded device. The study in [3] presented four steps for firmware modification: (1) firmware sample acquisition; (2) binary analysis; (3) firmware disassembly; and (4) derivation of the firmware update validation method. In general, the firmware is obtained from the vendor Web site, and then, the binary files are extracted from it. The binary files are used for selecting a target file and identifying the file format, e.g., the ext4 or YAFFS file system. Disassembly processes are performed to identify function names, string analysis, and so on. Finally, a disassembly analysis, including black box testing or hardware debugging, is performed to reduce the number of analysis functions, identify control flows, and circumvent update validation methods.

4 Attack Model and Scenario

In this section, we propose an attack model and an attack scenario of an Android OS-based telematics device.

4.1 Telematics Services

In south Korea, the Android OS-based telematics device provides several services for the driver's convenience: remote door lock/unlock, remote engine start/stop, self-diagnostics, and so on. In the US, vehicles with same telematics model provide similar services. These services can be classified into two types: one related to low speed CAN and the second to high speed CAN.

- Services related to low speed CAN: remote door unlock/lock and remote engine start/stop;
- Services related to high speed CAN: self-diagnostics.

Both service types are triggered by either the commands of an authorized smart phone application produced by the vehicle manufacturer or the commands of applications installed on the telematics device. Then, the device generates CAN packets for the vehicle controls. Fig. 2 shows the telematics service architecture of our target device. The remote controls are operated by SMSs from the telematics center.

4.2 CAN Packets from the Telematics Device

According to [29], network packet monitoring of CAN is important for obtaining the meaningful CAN packets of target devices. However, we performed CAN packet monitoring to find exploitable commands. In other words, monitoring was used for identifying the exploitable remote command that generates CAN packets. The monitoring results were

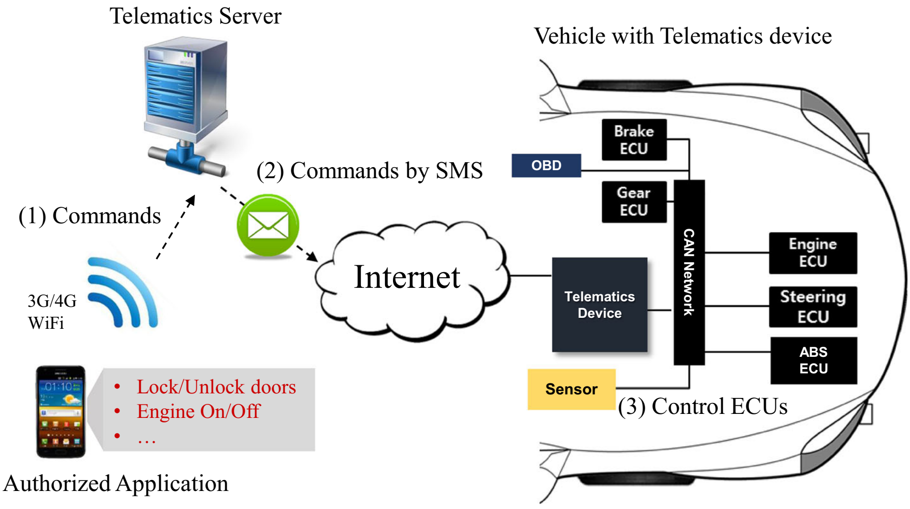


Fig. 2 Telematics services architecture of the target device

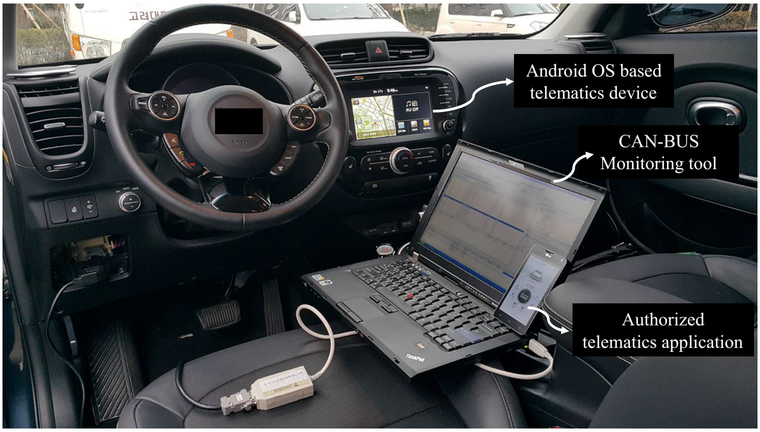

[redacted]
Android OS based telematics device
CAN-BUS Monitoring tool
Authorized telematics application

Fig. 3 Environment of CAN packet monitoring

Table 2 Tools used for the monitoring experiment

Product	Model name	Description
Vehicle	Mid-size car	Equipped with an Android 2.3.4-based telematics device
CAN-BUS monitoring tool	PCAN Explorer	Commonly used S/W
Smart phone	Galaxy Note 4	Equipped with an authorized telematics application

utilized for creating an attack scenario and analyzing the telematics device's firmware. The environment for CAN packet monitoring was set up as shown in Fig. 3 and Table 2. We designed the following three scenarios and obtained the packet monitoring results shown in Table 3. We omit detailed packet information to prevent unauthorized use.

Table 3 CAN packet related to telematics services

Types		CAN ID	Data frame
Low speed CAN service	Unlock door command	0x078	40 00 · · · ·00 00
	Lock door command	0x078	80 00 · · · ·00 00
High speed CAN service	Self diagnostics command	0x7D0	04 18 · · · ·00 00
		0x7D1	04 18 · · · ·00 00
		0x7D6	04 18 · · · ·00 00
		0x7D9	02 58 · · · ·00 00
		0x7DE	02 58 · · · ·00 00
		0x7E0	04 18 · · · ·00 00
		0x7E1	04 18 · · · ·00 00
		0x7E8	02 58 · · · ·55 55
		0x7E9	02 58 · · · ·AA AA

- Scenario 1 (using telematics services): we performed the CAN packet monitoring without using any telematics services. There exist 45 CAN IDs in high speed CAN and 16 CAN IDs in low speed CAN.
- Scenario 2 (using services related to low speed CAN): we performed the CAN packet monitoring when telematics services related to low speed CAN were used. The results confirmed that the telematics device generates several B-CAN packets, as shown in Table 3.
- Scenario 3 (using services related to high speed CAN): we performed the CAN monitoring when the services related to high speed CAN were used. We were able to find additional CAN IDs, as shown in Table 3, whenever the self-diagnostics service was in progress. However, we could not find the request packets with CAN ID "7DF,[7] which is considered to be the trigger packet generated by the telematics device for the self-diagnostics system. We thought that the packets obtained during self-diagnostics processes were generated by general ECUs not the telematics device. We could not find meaningful packets generated by the telematics device in this scenario.

Through the CAN monitoring experiments, we were able to select "Unlock Door Command" as the exploitable command, because there appears to be no CAN packet from the telematics device in Scenario 3.

4.3 Attack Model and Scenario

4.3.1 Proposed Attack Model

We constructed an attack model based on the Computer Emergency Response Team (CERT) Taxonomy [15], as shown in Table 4. The assumptions of our attack model are as follows.

[7] OBD2 onboard diagnostics parameter ID (PID) codes are used in diagnostic tools. The diagnostic tools send PID queries defined by the Society of Automotive Engineers (SAE) J1979. In general, the CAN ID of request packets is "7DF." ECUs that listen to the request packets send response packets to notify their status using one CAN ID in the range "7E0"–"7EF."

Table 4 Proposed attack model based on CERT

	Explanation
Attacker	Can modify Android OTA firmware (i.e., update.zip)
Tools	Malicious OTA firmware
Vulnerability	Android OTA firmware update using test certificates
	Repackaging of Android applications
	Design of CAN (no authentication)
Action	A command for remote door open
	A command for GPS trace
Target	Vehicles with an Android OS-based telematics device
Unauthorized result	Unauthorized door open
	Violation of location privacy
Object	Vehicle theft
	Trace a target vehicle

- Abilities of an attacker: an attacker is assumed to know the abstract architecture of the telematics system and features of CAN. He/she can also analyze and modify Android OTA firmware, i.e., update.zip. Furthermore, the attacker can find the device specific key combination for setting the recovery mode. In general, information about the abstract telematics system architecture, features of CAN, and modification tools of Android OTA firmware is publicly known. Hardware buttons for the recovery mode are also identified easily by a brute force test; if a telematics device has ten hardware buttons, the attacker has to perform $({}_{10}C_1 + {}_{10}C_2 + {}_{10}C_3 = 175)$ trials at most, because the number of buttons for the recovery mode is usually up to three for convenience of setting the recovery mode [9].
- Vulnerabilities of target: the target vehicle with an Android OS-based telematics device has two types of vulnerability, that of CAN and that related to the Android OS. According to [17], CAN does not provide any security mechanisms, e.g., no access control or data authentication. Therefore, this leads to replay attacks on the target vehicle. In addition, many Android OS-based devices allow malicious custom ROMs to be installed on the devices and it is possible to decompile and repack Android applications.
- Behavior of victims: the victim, i.e., the driver of the target vehicle, downloads a malicious ROM that provides attractive features, e.g., watching either television or DVDs while driving, and installs it using the recovery mode.

4.3.2 *Proposed Attack Scenario*

We propose a realistic attack scenario, as follows.

1. An attacker downloads valid OTA firmware from a public Web site of a telematics center;
2. The attacker modifies the valid firmware to produce malicious OTA firmware, including applications with an unauthorized remote door open command, a GPS trace command, mobile number notification, and attractive functions.

3. The attacker finds a specific key combination for the recovery mode of the target telematics device. Then, he/she distributes the malicious firmware with information about the positions of hardware buttons for the recovery mode.
4. The victim having the vehicle with the target telematics device downloads the modified OTA firmware and installs it. Then, the compromised device sends its own mobile number to the attacker.
5. The attacker traces the GPS of the target vehicle whenever he/she wants. Then, he/she commits vehicle theft using the malicious commands (commands are transmitted by SMSs).

The result of this attack scenario is similar to a scenario in [4]. However, the attack methodologies used for the scenario are different in terms of platform analysis.

5 Analysis and Modification of OTA Firmware

In this section, we describe our analysis and modification of an Android OS-based telematics system. The overall processes of our analysis are shown in Fig. 4. We omit detailed information to prevent attackers from using our results directly.

5.1 Analysis of OTA Firmware

5.1.1 Structure of OTA Firmware

OTA firmware for the target telematics device was obtained from a public Web site of a manufacture. It includes an update directory. Fig. 5 shows important subdirectories and files in the update directory. A detailed explanation of the subdirectories and the files is given in Table 5.

5.1.2 Target Selection for Analysis

As shown in Table 5, system.img includes system applications (app directory), system libraries (framework directory), and other directories. We thought that applications related to the telematics services exist in the app directory and they may use libraries in the framework directory. Thus, we selected the system.img file as a target for the analysis of telematics services. Table 6 shows the tools used in our analysis.

5.2 Analysis of system.img

The system.img file of target firmware is the ext4 file system, so that it can be mounted by using the command “sudo mount -o loop -t ext4 system.img [destination directory].” After mounting system.img, we determined that there are 67 system applications.

5.2.1 Deodex of system.img

Every system application in the app directory is composed of a .apk file and a .odex file. Since a .odex file has a device-dependent format, it can be converted back to the corresponding .dex file for detailed analysis. This is called a deodex process. In the deodex

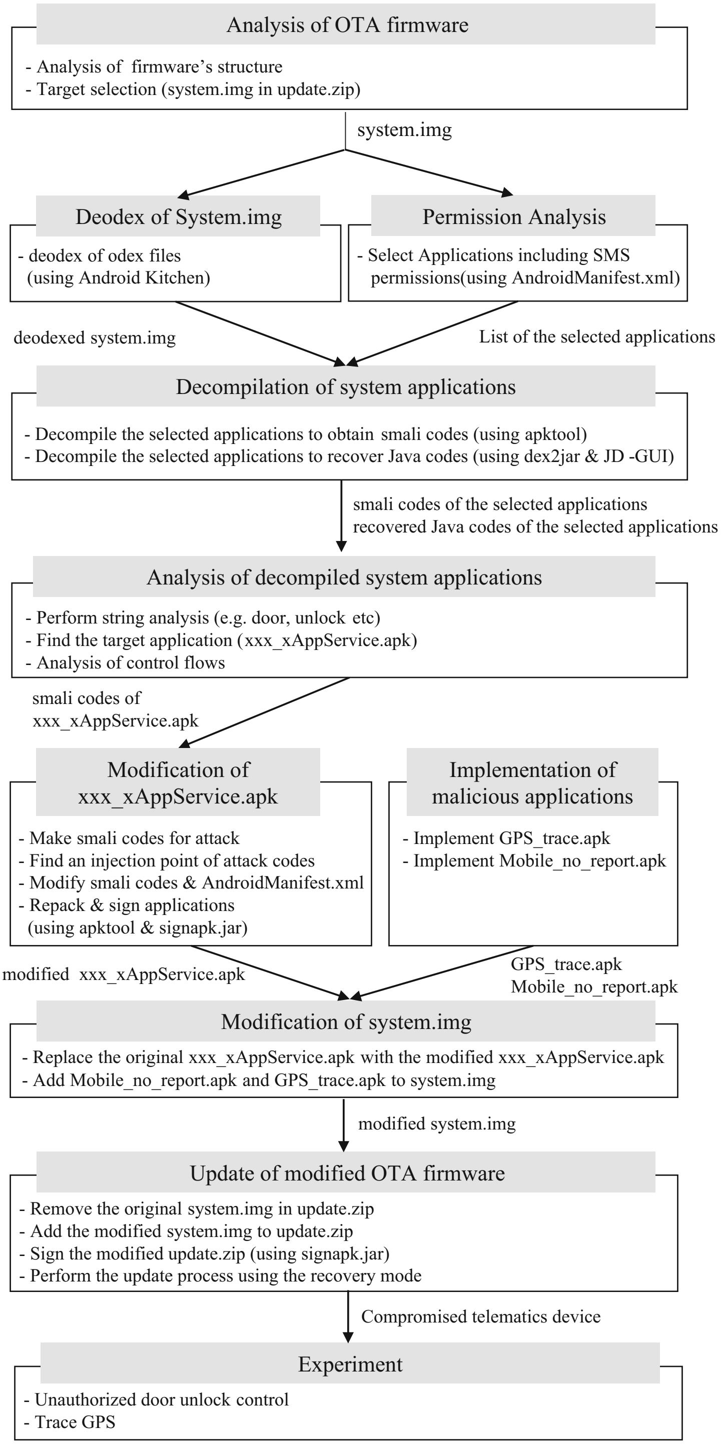

Analysis of OTA firmware
- Analysis of firmware's structure
- Target selection (system.img in update.zip)
system.img
Deodex of System.img
- deodex of odex files
(using Android Kitchen)
Permission Analysis
- Select Applications including SMS
permissions(using AndroidManifest.xml)
deodexed system.img
List of the selected applications
Decompilation of system applications
- Decompile the selected applications to obtain smali codes (using apktool)
- Decompile the selected applications to recover Java codes (using dex2jar & JD -GUI)
smali codes of the selected applications
recovered Java codes of the selected applications
Analysis of decompiled system applications
- Perform string analysis (e.g. door, unlock etc)
- Find the target application (xxx_xAppService.apk)
- Analysis of control flows
smali codes of
xxx_xAppService.apk
Modification of
xxx_xAppService.apk
- Make smali codes for attack
- Find an injection point of attack codes
- Modify smali codes & AndroidManifest.xml
- Repack & sign applications
(using apktool & signapk.jar)
Implementation of
malicious applications
- Implement GPS_trace.apk
- Implement Mobile_no_report.apk
modified xxx_xAppService.apk
GPS_trace.apk
Mobile_no_report.apk
Modification of system.img
- Replace the original xxx_xAppService.apk with the modified xxx_xAppService.apk
- Add Mobile_no_report.apk and GPS_trace.apk to system.img
modified system.img
Update of modified OTA firmware
- Remove the original system.img in update.zip
- Add the modified system.img to update.zip
- Sign the modified update.zip (using signapk.jar)
- Perform the update process using the recovery mode
Compromised telematics device
Experiment
- Unauthorized door unlock control
- Trace GPS

Fig. 4 Overall processes of our analysis

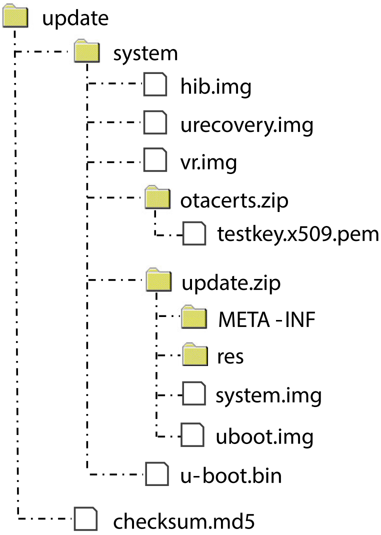


Fig. 5 Structure of OTA firmware

Table 5 Description of the OTA firmware

Name	Description
hib.img	The hibernation process stores all the states of the telematics device in non-volatile memory before power off. Then, the stored states are used for fast-boot
urecovery.img	A separate partition for Android maintenance routines such as system update or factory reset
vr.img	A separate partition for voice recognition
otacerts.zip	A Zip file containing an X.509 certificate
META-INF	A directory containing CERT.RSA, CERT.SF, updater-script, and update-binary, which are used for verification of signature on the OTA firmware and setting update parameters
res	A directory containing a set of public keys built into the OTA firmware
system.img	Contains system applications (app directory) and system libraries (framework directory)
uboot.img, uboot.bin	Universal boot loader for embedded devices, used for initialization of hardware, setting kernel parameters, and starting kernel
checksum.md5	Contains MD5 hash values of each file in the OTA firmware

process, we used the command “advanced option → Deodex files in your Rom” of the Android Kitchen tool and finally obtained the .dex files of the system applications.

5.2.2 Permission Analysis

In order to reduce the number of analysis objects, we analyzed each AndroidManifest.xml included in every system application to check whether it contained the “RECEIVE_SMS” permission. Since the target telematics services, such as remote door unlock, are triggered by SMSs from the telematics server, this checking process helps determine the system

Table 6 Tools used for the analysis of system.img

Name	Description
Android Kitchen	Used to convert ODEX files into DEX files.
apktool	Used to decode an APK file into smali codes and to build a modified APK file.
dex2jar	Used to convert DEX files into the Java ARchive (JAR) format.
JD-GUI	Used to reconstruct Java source codes from CLASS files.
signapk.jar	Used to sign APK files or OTA update files

application used for the remote vehicle control. As a result, we found six system applications that include the “RECEIVE_SMS” permission and these applications were selected for detailed analysis.

5.2.3 Decompilation of System Applications

In general, the analysis of a .dex file comprises two steps. In the first step, the .dex file as Dalvik VM bytecode is converted into Java Virtual Machine (JVM) bytecode using dex2jar. Then, a Java decompiler, such as JD-GUI, is used to recover the Java source codes. However, the codes recovered by the decompiler are not the same as the original codes, so that they are usually used to find meaningful names of the function/variable and control flows of function calls. In the second step, the .dex file is converted into the smali source codes as Dalvik VM assembly using a tool called apktool. Although the smali codes are more difficult to understand than the recovered Java source codes of the first step, they are used for modifying the original code, because accurate smali codes can directly convert into Dalvik VM bytecodes during repackaging. Thus, the above two steps were used in our analysis. After converting six .dex files of six system applications into six .jar files using dex2jar, we recovered six Java source files from six .jar files using JD-GUI. Fortunately, the original names of Java classes, methods,and variables were recognizable, because the .dex files were not obfuscated.

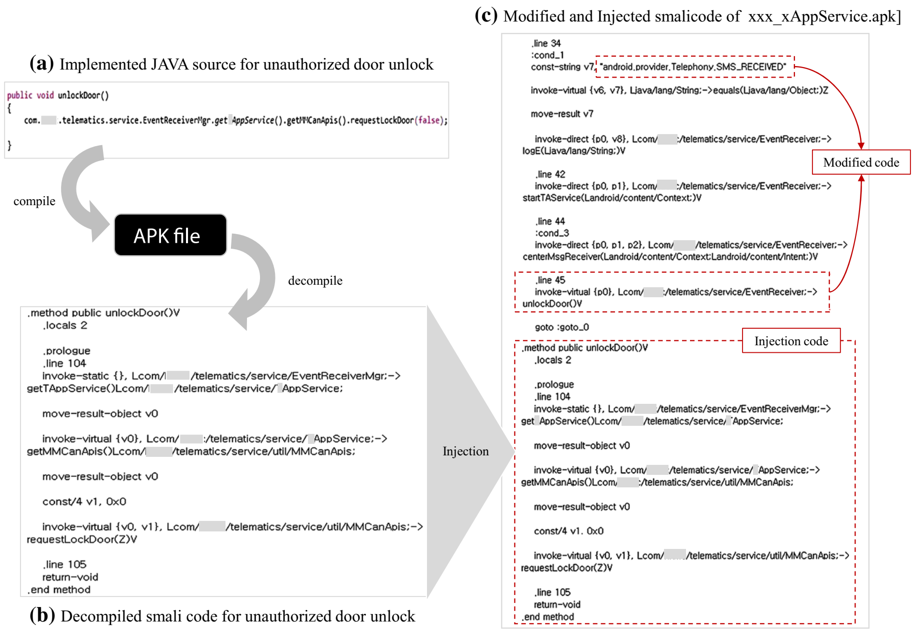
5.2.4 Analysis of Decompiled System Applications

We performed a search process that checks whether meaningful words, such as “door” or “unlock” are included in the obtained Java sources. Through the search process, we could find the “requestlockDoor(boolean input)” function in the xxx_xApp Service class in xxx_xAppService.apk. If the input value is set as false, this function performs the door unlock control. To determine the control flow of this function, we performed trace orders of function calls. Finally, we determined the order of the function call “xxx_xAppService() → xMMCanApis() → requestLockDoor(false).” In addition, it was determined that xxx driver.jar in the framework directory is used for door control.

5.3 Modification of system.img

5.3.1 Modification of xxx_xAppServices.apk

Making Smali Code for Attack: It may be difficult for an attacker to produce correct smali codes that do not have any errors, because smali codes are used in the assembly for Dalvik



(a) Implemented JAVA source for unauthorized door unlock

```
public void unlockDoor()
{
    com.[redacted].telematics.service.EventReceiverMgr.get[redacted]AppService().getMMCanApis().requestLockDoor(false);

}
```

(b) Decompiled smali code for unauthorized door unlock

```
.method public unlockDoor()V
    .locals 2

    .prologue
    .line 104
    invoke-static {}, Lcom/[redacted]/telematics/service/EventReceiverMgr;->
getTAppService()Lcom/[redacted]/telematics/service/[redacted]AppService;

    move-result-object v0

    invoke-virtual {v0}, Lcom/[redacted]/telematics/service/[redacted]AppService;->
getMMCanApis()Lcom/[redacted]/telematics/service/util/MMCanApis;

    move-result-object v0

    const/4 v1, 0x0

    invoke-virtual {v0, v1}, Lcom/[redacted]/telematics/service/util/MMCanApis;->
requestLockDoor(Z)V

    .line 105
    return-void
.end method
```

(c) Modified and Injected smalicode of xxx_xAppService.apk]

```
    .line 34
    :cond_1
    const-string v7, "android.provider.Telephony.SMS_RECEIVED"

    invoke-virtual {v6, v7}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

    move-result v7

     invoke-direct {p0, v8}, Lcom/[redacted]/telematics/service/EventReceiver;->
 logE(Ljava/lang/String;)V

     .line 42
     invoke-direct {p0, p1}, Lcom/[redacted]/telematics/service/EventReceiver;->
 startTAService(Landroid/content/Context;)V

     .line 44
     :cond_3
     invoke-direct {p0, p1, p2}, Lcom/[redacted]/telematics/service/EventReceiver;->
 centerMsgReceiver(Landroid/content/Context;Landroid/content/Intent;)V

     .line 45
     invoke-virtual {p0}, Lcom/[redacted]/telematics/service/EventReceiver;->
 unlockDoor()V

     goto :goto_0

 .method public unlockDoor()V
     .locals 2

     .prologue
     .line 104
     invoke-static {}, Lcom/[redacted]/telematics/service/EventReceiverMgr;->
 get[redacted]AppService()Lcom/[redacted]/telematics/service/[redacted]AppService;

     move-result-object v0

     invoke-virtual {v0}, Lcom/[redacted]/telematics/service/[redacted]AppService;->
 getMMCanApis()Lcom/[redacted]/telematics/service/util/MMCanApis;

     move-result-object v0

     const/4 v1. 0x0

     invoke-virtual {v0, v1}, Lcom/[redacted]/telematics/service/util/MMCanApis;->
 requestLockDoor(Z)V

     .line 105
     return-void
 .end method
```

Fig. 6 Modification and injection of smali codes

VM. To obtain the accurate smali code for attacks, we implemented the Java source, as shown in Fig. 6a, and compiled the sources in Android 2.3.4 with xxxdriver.jar. Then, we extracted the .apk file from an Android emulator using ADB. Finally, the classes.dex file of the .apk file was converted into the smali codes by using the apktool.

Finding an Injection Point of Attack Codes: xxx _xAppService.apk contains EventReceiver.class, which extends BroadcastReceiver.[8] In particular, the OnReceive() function of the EventReceiver.class handles intents related to SMSs. Thus, we chose this function as a point of attack code injection for the modified codes to be triggered by SMSs.

Modification of Smali Codes: In the original EventReceiver.smali corresponding to the EventReceiver.class, the onReceive() function includes only custom SMS intent ("SMS_CENTERMSG_RECEIVED" and "CDMA_SMS_CENTERMSG_RECEIVED"). For responding to general SMSs, not SMSs from a telematics center, we changed one of the custom intents to "android.provider.Telephony.SMS_RECEIVED", which is the original Android intent-related SMS reception. Then, we added the produced smali codes for door unlock control generated in Section 5.3.1 to ".line 45" of the EventReceiver.smali, because an object of the xxx_xAppService class is activated after ".line 42" of the EventReceiver.smali. Fig. 6c shows the modification and injection of smali codes.

Modification of AndroidManifest.xml: We added the <action> element "android.provider.Telephony .SMS

_RECEIVED" in the <intent-filter> of the <receiver> section in order to handle the intents-related events of SMS reception.

[8] This is an Android component that responds to the system events called broadcast.

Repackaging and Signing: We repacked the modified smali codes using the apktool to obtain the modified xxx_ xAppServices.apk. Fortunately, we could find a signing key used in the original xxx_xApp Services.apk on the Internet; the platform.pk8 file and the platform.x509.pem file of the Freescale i.MX53 board are used. Then, this modified .apk file was signed using the signapk.jar with the obtained signing key.

5.3.2 Implementation of Malicious Applications

We produced two malicious applications: a mobile number report application (Mobile_no_report.apk) and a GPS trace application (GPS_trace.apk). Mobile_no_report.apk has "READ_PHONE_STATE and "RECEIVE_BOOT_COMPLETE" permissions to send a mobile number of a device to an attacker when the device completes a boot process.

GPS_trace.apk has "READ_SMS", "RECEIVE_SMS", and "ACCESS_FINE_LOCATION" permissions. The application reads the GPS information of the telematics device and sends this information to a server or the mobile device of the attacker whenever it hears an intent of SMS reception.

5.3.3 Modification of system.img

We mounted the deodexed system.img. Then, the app directory was modified; the original xxx_xAppServices.apk was replaced with the modified xxx_xAppServices.apk and Mobile_no_report.apk and GPS_trace.apk were added. After modification, the modified system.img was unmounted.

5.4 Update and Experiments

5.4.1 Update of Modified OTA Firmware

To update modified OTA firmware, the update.zip file including the modified system.img file should be signed with a signing key. In general, signing keys are protected from leakage by using cryptographic methods. However, we could find a valid signing key of the target telematics device, because the device uses a testkey that is provided on a public Web site. The test certificate in the otacerts of the OTA firmware was used for searching the corresponding signing key. With the signing key of the telematics device, we could sign the modified update.zip using the signapk.jar; the testkey.pk8 file and the testkey.x509.pem file of the Freescale i.MX53 board are used. Finally, we replaced the original update.zip with the modified update.zip for creating modified OTA firmware and updated the modified firmware on the target device using the recovery mode.

5.4.2 Experiments

Unauthorized Door Unlock Command After update of the modified firmware, anyone who knows the mobile number of the telematics device can perform remote control door unlock using an SMS. The results can be shown in the (https://www.youtube.com/watch?v=TKFP6hLoyco&feature=youtu.be).

GPS Trace Figrue 7 shows the location of our test vehicle. The compromised telematics device sent the GPS information of the vehicle whenever we sent a trace command SMS.

5.5 Comparison and Discussion

5.5.1 Comparison

There have been four results supporting the long-range wireless attacks on vehicles as shown in Table 7. While the existing attacks are based on specific threats using vulnerabilities of an authentication protocol or a femtocell device, our attack method is only based on Android OS's vulnerabilities which are originated from the open source policy. In other words, an adversary can get rich information about vulnerabilities of Android from public sites and can perform actual attacks on Android OS. Our attack procedures (permission analysis of applications → unpack and repack the target applications → system update using the recovery mode) could also be applied to any other telematics systems using Android OS. Thus, the proposed attack method is more scalable than other long-range attacks.

According to the work [5], architecture-neutral JAVA class files make the applications easy to decompile and it has much of information in original source codes. This means that our attack procedures based on analysis of JAVA class files are easier than other long-range attacks requiring binary code analysis or signal processing. In addition, we checked permissions (i.e., SMS related permissions) of applications in advance to minimize targets of detailed analysis.

5.5.2 Discussion

In our experiments, we implemented the GPS application to trace the locations of vehicles. However, it is easy to implement many malicious applications that threaten privacy because of the Android development environment, e.g., applications for accessing the SD card or recording a voice. Thus, open platforms not only have many vulnerabilities, but also provide attackers with public application development environments.

6 Countermeasures

We present several countermeasures, as follows, to prevent the analysis of firmwares and update of malicious firmwares.

6.1 Code Obfuscation

Code obfuscation is used to render source or machine codes more difficult to analyze. There are two types of code obfuscation for Android: source code obfuscation and bytecode obfuscation. ProGuard [23] as a Java source code obfuscator changes the names of package, class, and variable into random strings. Dalvik-obfuscator [6] and APKfuscator [2] are open-source bytecode obfuscation tools that support junk/dead bytecode injection.

6.2 Code Signing and Key Management

Code signing can prevent installation of malicious or unauthorized code. Code signing is used for booting a process, application distribution, and OTA firmware updates. According to [8], Nexus, an Android device, uses code signing so that it does not allow malicious

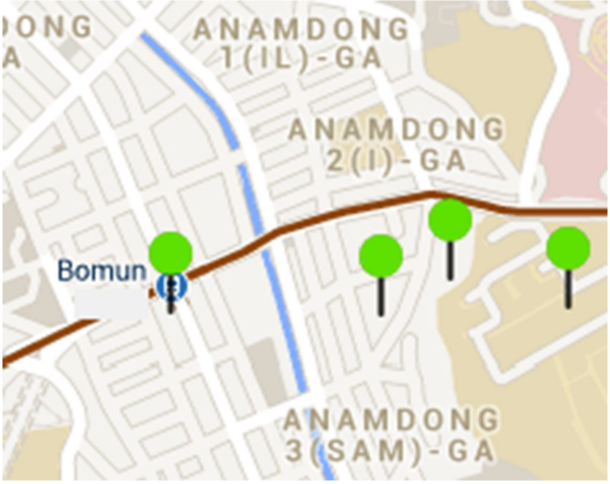


Fig. 7 GPS trace of the experimental vehicle

Table 7 Comparison of long-range wireless attacks on vehicles

	Target	Exploitation	Scalability	Additional devices
S. Checkoway et al. [4]	GM's OnStar	Vulnerabilities of the aqLink protocol	Specific environment	No
I. Foster et al. [11]	An aftermarket TCU (telematics control unit)	Vulnerabilities of an aftermarket TCU	Specific environment	Yes (a CAN-Bluetooth Dongle)
C. Miller et al. [19]	Jeep's Uconnect	Vulnerabilities of Sprint's femtocell device	Specific environment	Yes (a Sprint's device)
S. Woo et al. [29]	Self-diagnostic applications	Distribution of malicious applications	Specific environment	Yes (a CAN-Bluetooth Dongle)
Ours	Telematics system using Android OS	Vulnerabilities of Android OS	General Android OS	No

firmware with an invalid signature to be installed on itself. For example, KNOX of Samsung provides secure boot to prevent unauthorized boot loaders and Android OS from loading during the startup process using a hardware root of trust and code signing. Although the target telematics device also uses a code signing method, it allows malicious firmware to be installed, because the manufacture of the device uses the test certificate and the corresponding test private key. Thus, a private key for code signing should be issued to an authorized entity and it should be managed securely.

6.3 Remote Attestation

Remote attestation is used for verifying that the system has not been compromised. An attesting device transmits the current information about its own configuration to a remote verifier server to detect malicious firmware. In general, remote attestation is based on trusted hardware. KNOX also provides a hardware-based remote attestation service [26].

7 Conclusion

In this paper, we indicated the security problems of an Android OS-based telematics system. Because of the Android OS's open platform, the analysis of vulnerabilities and implementation of attack codes presented in this study are more practical than those in previous works. Our experiment using a real vehicle showed that anyone who knows the mobile number of the telematics device can perform unauthorized remote control of the door unlock and GPS trace functions of the experimental vehicle. We hope that our results facilitate the construction of secure telematics environments.

References

1. Android auto, http://www.android.com/auto/.
2. Apkfuscator, https://github.com/strazzere/APKfuscator.
3. Basnight, Z., Butts, J. L, Jr., & Dube, T. (2013). Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, *6*(2), 76–84.
4. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., & Kohno, T. (2011). Comprehensive experimental analyses of automotive attack surfaces. *Proceedings of the 20th USENIX Conference on Security*, SEC'11, Berkeley, CA, USA (pp. 6–6). USENIX Association.
5. Collberg, C. S., & Thomborson, C. (2002). Watermarking, tamper-proffing, and obfuscation: Tools for software protection. *IEEE Transactions on Software Engineering*, *28*(8), 735–746.
6. Dalvik-obfuscator, https://github.com/thuxnder/dalvik-obfuscator.
7. Davis, R., Burns, A., Bril, R., & Lukkien, J. (2007). Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, *35*(3), 239–272.
8. Drake, J. J., Lanier, Z., Mulliner, C., Fora, P. O., Ridley, S. A., & Wicherski, G. (2014). *Android Hacker's Handbook*. Hoboken: Wiley.
9. Elenkov, N. (2014). *Android security internals an in-depth guide to Android's Security Architecture*. San Francisco: No Starch Press.
10. Exclusive: Google aiming to go straight into car with next android - sources, http://www.reuters.com/article/2014/12/18/us-google-cars-idUSKBN0JW2PS20141218.
11. Foster, I., Prudhomme, A., Koscher, K., & Savage, S. (2015). Fast and vulnerable: A story of telematic failures. *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C. USENIX Association.
12. Gm developer network, https://developer.gm.com/.
13. Gm to adopt android os in. (2016). http://gas2.org/2014/11/06/gm-adopt-android-os-2016/.
14. Honda debuts android-based infotainment system for europe, http://www.cnet.com/news/.
15. Hoppe, T., Kiltz, S., & Dittmann, J. (2011). Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, *96*(1), 11–25. Special Issue on Safecomp 2008.
16. Hyundai kia will offer android-based infotainment systems, http://telematicsnews.info/.
17. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H. and Savage, S. (2010). Experimental security analysis of a modern automobile. *Security and Privacy (SP), 2010 IEEE Symposium on*, (pp. 447–462) May.
18. Larson, U. E., & Nilsson, D. K. (2008). Securing vehicles against cyber attacks. *Proceedings of the 4th annual workshop on Cyber security and information intelligence research (CSIIRW '08),* Article No.30, New York, NY: ACM.
19. Miller, C., & Valasek, C. (2015). Remote exploition of an unaltered passenger vehicle. *Black Hat USA, 2015.*

20. Miller, C., & Valasek, C. (2013). Adventures in automotive networks and control units. In *DEFCON 21*, Las Vegas, NV, August 2013.
21. Mirrorlink, http://www.mirrorlink.com/.
22. Open automotive alliance, http://www.openautoalliance.net.
23. Proguard, http://proguard.sourceforge.net/.
24. Rouf, I., Miller, R., Mustafa, H., Taylor, T., Oh, S., Xu, W., Gruteser, M., Trappe, W., & Seskar, I. (2010). Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, Berkeley, CA, USA (pp. 21–21). USENIX Association.
25. The openxc platform, http://openxcplatform.com/.
26. White paper : An overview of samsung knox platform, https://www.samsungknox.com/en/support/knox-workspace/white-papers.
27. Wolf, M., Weimerskirch, A., Paar, C., & Bluetooth, M. (2004). Security in automotive bus systems. *Proceedings of the Workshop on Embedded Security in Cars (escar)04*.
28. Wolf, M., Weimerskirch, A., & Wollinger, T. (2007). State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems*, *2007*(1), 074706.
29. Woo, S., Jo, H., & Lee, D. (2015). A practical wireless attack on the connected car and security protocol for in-vehicle can. *Intelligent Transportation Systems, IEEE Transactions on*, *16*(2), 993–1006.
30. Zhang, T., Antunes, H., & Aggarwal, S. (2014). Defending connected vehicles against malware: Challenges and a solution framework. *Internet of Things Journal, IEEE*, *1*(1), 10–21.

Hyo Jin Jo received the BS degree in industrial engineering and the Ph.D. degree in information security from the Korea University, Seoul, Korea, in 2009 and 2016, respectively. Currently, He is a Postdoctoral Researcher with the Department of Computer and Information System, University of Pennsylvania, Philadelphia, PA, USA., His research interests include cryptographic protocols in authentication, applied cryptography, security and privacy in ad hoc networks and smart car security.

Wonsuck Choi received the B.S. degree in Mathematics from University of Seoul, Seoul, Korea in 2008, and the M.S. degree in Information Security from Korea University, Seoul, Korea in 2013. Currently, he is working toward the Ph.D. degree in Information Security, Graduate school of Information Security at Korea University. His research interests include applied cryptography, healthcare security and authentication and key exchanging in sensor network.

Seoung Yeop Na received his B.S. degree from the Computer Science at Myung-Ji University in 2013, and the M.S. degree from the Information Security at Graduate school of Information Security, Korea University in 2015. Currently, he is a security engineer of Korea Federation of Community Credit. His research interests include personal information protection, financial security.

Samuel Woo received the M.S. degree in Computer Science from Dankook University, Seoul, Korea in 2010, and the Ph.D. degree from Information Security at Gratudate school of Information Security, Korea University in 2016. His research interests include cryptographic protocols in authentication, applied cryptography, security and privacy in vehicular networks and Controller Area Network security.

Dong Hoon Lee received the B.S. degree from the Department of Economics at Korea University, Seoul, in 1985, and the MS and Ph.D. degrees in Computer Science from the University of Oklahoma, Norman, in 1988 and 1992, respectively. Currently, he is a professor and the dean of the Graduate School of Information Security at Korea University. Since 1993, he has been with the Faculty of Computer Science and Information Security at Korea University. From 2004 to 2015, he served as the president of Ubiquitous Information Security Organization, which has been supported by BK21 Project in Korea. His research interests include the design and analysis of cryptographic protocols in key agreement, encryption, signature, embedded device security, and privacy-enhancing technology (PET).