A Testbed for Security Analysis of Modern Vehicle Systems

Xi Zheng*, Lei Pan*, Hongxu Chen†, Rick Di Pietro*, and Lynn Batten*
*Deakin University, Geelong, Australia
Email:xi.zheng@deakin.edu.au, l.pan@deakin.edu.au, rick.my.mail@gmail.com, lynn.batten@deakin.edu.au
†State Key Laboratory of Automotive Safety and Energy, Tsinghua University, China
Email: herschel.chen@gmail.com

***Abstract*—As wireless and telecommunicaton infrastructure communications have been integrated into modern vehicle systems (i.e., infotainment systems and vehicle to vehicle systems), the security implications on the relatively unchanged underlying network protocols inside the vehicles are investigated by researchers and industrial experts in the corresponding domain. Some researchers have achieved the investigation by using actual vehicles or vehicle components, and although this is an effective way to produce an accurate testing environment it provides little or no flexibility in its configuration. In this work, we produce a testbed architecture for vehicle security investigation that not only can reproduce the complexity of numerous vehicle control units running at real time inside the vehicle CAN bus, but is also reconfigurable giving the testbed the ability to replicate various test configurations. We create a prototype of the testbed and conduct an experiment to demonstrate a new type of security attack through an infotainment gateway in the testbed, the source code of the prototype and the test data of the experiment are all uploaded for sharing.**

I. Introduction

Modern car systems are managed by a collection of networked Electronic Control Units (ECUs) over a network bus, the most common being the Controller Area Network (CAN) Bus. These systems manage driver displays, emission controls, engine timing and more, while also managing safety-critical functions such as braking and steering, etc. Modern cars can include up to 100 independent ECUs, or nodes, that work independently with no requirement for a central CPU, which enables a reliable communication network for the real time requirements of vehicles [16].

However, with the advent of infotainment systems and more such systems have been integrated into vehicles (i.e., iDrive of BMW [3] and MMI of Audi [1]), which requires wireless connection to personal devices and vehicle communication with cellular networks, it has exposed the original vehicle system, especially the CAN bus, to significant external vulnerabilities that the underlying CAN bus protocol was never designed for. The potential damage that could occur from these exposures is immense, not only in regards to the cost of damages and recalls for manufacturers but also potential injury or even loss of life. Therefore, implementing security into the CAN bus should not be seen as an expense but rather an investment in order to avoid any situation where this could be possible.

In 2015 Chrysler had to recall over 1.5 million vehicles due to security vulnerabilities discovered in their vehicle systems, enabling remote code execution to security researchers who gained access through the cellular network [15]. In order to avoid this, several manufacturers are looking into Over-The-Air (OTA) update systems so they can patch potential vulnerabilities as they occur. OTA systems are potentially very valuable in avoiding incidents such as the one Chrysler faced and are predicted to become commonplace in infotainment-systems, navigation, apps, telematics firmware and even individual node patching by 2022 [18]. However, the implementation of these systems, if not done correctly, could end up costing manufacturers more and expose cars to more vulnerabilities. Thus it is important to create a testbed to verify these systems in regard with security vulnerabilities. Some researchers have achieved this goal by using actual vehicles or vehicle components and although this is an effective way to produce an accurate testing environment it provides little or no flexibility in its configuration, and sometimes very costy and dangerous.

In this paper, we investigate different testbed available for the security analysis of modern vehicle systems which integrate original close-looped CAN bus with infotainment-systems. We propose a novel testbed architecture for flexible, effective, and efficient vehicle security analysis. We also create a testbed prototype which is built on top of National Instruments dedicated hardware and LabVIEW software. The proposed testbed architecture is a real time simulation of a real electric vehicle system which integrates CAN bus with a simulated Infotainment system. The testbed is able to capture internal CAN messages for future security analysis and is also able to inject secuirty attack CAN messages from simulated infotainment terminals, which is one of the first kind in close loop security testbed for modern vehicle systems.

The rest of the paper is organized as follows. Section II gives a brief introduction of related work on vehicle testbed. The architecture of the testbed along with its details are explained in Section III. Section IV walks through the prototype of the testbed and the experiment we conducted on the prototype to verify the effectiveness of the testbed, and explains a new attack launched in the testbed through the infotainment gateway. Finally, Section V give the conclusions about the paper.

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.357

II. Related Work

Vehicles today have a plethora of networked independent electronic control units (ECUs) interconnected mostly by a communication message bus operating the CAN protocol. However, the security vulnerability of the underlying CAN bus has been demonstrated in many recent research [6], [9], [4], [13], [5], [14], [11], [15]. Among these work, Miller and Valasek [13], [14], [15] have presented technical solutions of compromising the security of vehicles via manipulating and hijacking on-board ECU. As a result of which, many vehicle vendors began to pay close attention to vehicular security subject to cyber attacks.

In fact, Hoppe et al. [6] predicted the feasibility and practicality of such attacks against ECU in 2008; a simple example in [6] showed how an attacker may easily control the CAN bus and alert the driver through a message displayed on the CD player to switch off the engine. Koscher et al. [9] demonstrated in the lab that a malicious attacker can control most key vehicular functions like braking, accelerating and halting the engine despite of driver's intervention; other auxiliary but important functions like speedometer display can also be tampered with by the attacker via a compromised ECU. In consequence, the safety of vehicle users will be severely endangered if attackers gains access to ECU.

Furthermore, Miller et al. [15] showed that an attacker could successfully access ECU remotely. Enabling attacks from remote machine is the last straw to break the camel's back, that is, attackers may exploit wifi to remotely crash a vehicle on the motor way at a high speed. More specifically, a communication system enabling wifi hot spot and bluetooth interconnected to an on-board ECU can be very vulnerable [15]. Because this communication system is used as a self-contained package, there are many vehicles subject to this type of attack. According to [14], [15], there were more than 290,000 vulnerable vehicles in the United States alone.

Nevertheless, testing involving vehicles is a daunting task. Most of research use real vehicles in the lab for testing, such as in [9], [4], [13], [14], [11], [15]. However, the use of production vehicle is not ideal for most security specialists due to various constraints like budget, space, ethics approval and so on. Other research adopt a hybrid method — many software simulation cases together with a few real vehicle components [6], [5]. Though the hybrid approach significantly reduces the cost and lowers the risk of injury, the testable capacities are limited.

The fatal shortcomings of the above two methods are inefficiency and lack of flexibility: Each test case is only able to be tested against a single configuration; and a test case can only be applied to the ECUs already in production. The first drawback is somewhat inconvenient to the tester, but the latter forces the vendor to waste resources in producing actual ECUs for testing. Hence, there exists a research gap in the literature on how to use software simulation approach together with configurable and programmable ECUs to conduct security testing on vehicles.

To integrate software-based simulator with ECUs, we will need to wielder digital signals generated from software and analog signals used in ECUs [8]. The use of hardware in the loop (HIL) for testing the functions of ECUs is a mature technology [10], [19], [20]. But security testing on ECU is not commonly practiced, which is the second research gap we identified.

Lastly, to our best knowledge, the state of the art literature does not provide a portable and flexible testbed solution which enables security experts to conduct security tests on ECUs.

III. The Testbed

A. Basic Information

A Controller Area Network (CAN) bus is a message-based protocol designed to allow devices such as microcontrollers to communicate with each other without a host computer. There are other popular in-vehicle network protocols such as FlexRay, LIN, or MOST deployed in today's modern vehicles but CAN is one of the most prolific and so it is CAN that is the initial focus of this vehicle security testbed [12].

Real-time simulation is an "online" version of discrete-time simulation, where time moves forward in steps of pre-defined duration [17]. In our testbed, every CAN packet is an array of 6 unsigned 32 bit integers, giving a packet that's 192 bits long. Each packet is captured inside the testbed and written to a log file on a new line as a series of 6 tab delineated 32 bit values that are displayed as a hexadecimal string. Hexadecimal was chosen as the display format as it is the default display in many dominating package sniffing tools (i.e., Wireshark). As shown in Fig. 1, the vehicle ECUs are strongly recommended to run on Field-Programmable Gate Array (FPGA) devices and a real time operating system environment for portability and avoid time synchroniation errors of the real-time simulation [2]. We also assume that in the testbed, functionality of each Electronic Control Unit (ECU) in the provided vehicle is modelled with LabVIEW software, and the specificiation of the vechile and ECUs is provided and verified by a real vehicle company to make sure the accuracy of the input specfciation to the testbed is guaranteed against the real vehicle and ECUs modelled. We also presume that the underlying simulation device vendor can also provide industrial grade add-on modules that are able to add other desired interfaces in the future, such as the FlexRay, LIN, or MOST as mentioned earlier, which can be achieved by National Instruments devices.

B. Architecture Design

The aim of this paper is to establish a real time simulation environment not only for the existing CAN bus based on a real vehicle, but also to integrate the simulated CAN bus system with an emulated infotainment system to explore any possible security vulnerabilities exposed.

1) ECU: In Fig. 1, the vehicle Electronic Control Units (ECUs) are the main controller units for the simulated vehicle. They listen on the CAN bus for frames and filters out packets with its unique identifier. In the current version of our test design, the data frame carries information regarding the

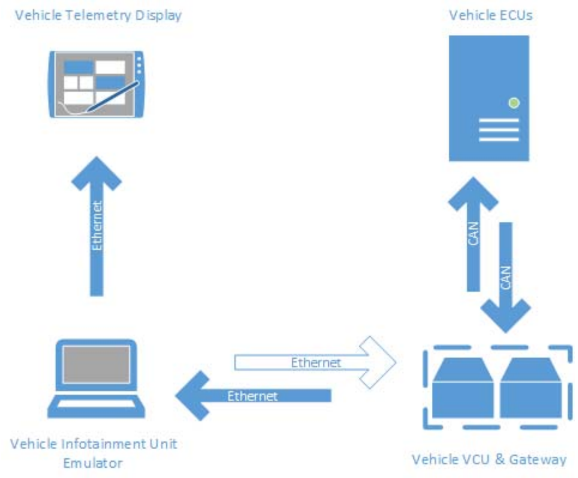


Fig. 1: The Testbed Architecture - Overview

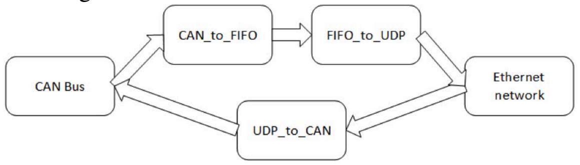


Fig. 2: The Gateway

vehicle angle and force vector, which can be easily extended to include more realistic ECU information.

The vehicle ECUs broadcast 192bit CAN packets with a unique Arbitration ID (Identifier) (i.e., 769). This ID is used to share information among vehicle ECUs. In the architecture, the ECU or a group of ECUs can be deployed to dedicated machine with real-time operating system (i.e., NI cRIO-9062) and a dedicated CAN bus interface module (i.e., NI 9853). In Vehicle ECUs, we can incorporate ECUs for brake control, battery management, Door Control, power steering etc.

2) VCU: In Fig. 1, the Vehicle Control Unit (VCU) is a dynamic model for the simulated electric vehicle. In the current design, It navigates the simulated electric vehicle according to the current position and velocity messages published on the CAN bus from the ECU. A supervisory UI is provided to check the virtual vehicle in the testbed. At a later stage, a real-world map can be incorporated inside the supervisor UI to give a realistic driving context with simulated vehicle.

3) CAN Gateway: In Fig. 1, the role of the CAN Gateway is to provide two main functions that, although similar, do differ according to the desired use; simulated In-vehicle CAN to Ethernet gateway and a CAN to Ethernet data capture gateway.

As in Fig. 2, the in-vehicle Ethernet to CAN gateway resides on both the CAN bus and the in-vehicle Ethernet network. CAN frames are read off the bus, the CAN packet is bundled into a UDP frame and transmitted on the Ethernet network and vice versa.

The data capture Ethernet to CAN gateway functions in a similar way to the in-vehicle gateway but differs in that the CAN packets are bundled in TCP frames and for capture and recording from a dedicated supervisory module.

The CAN to Ethernet gateway receives every packet on the CAN bus and re-transmits it on the Ethernet network. CAN packets are packaged into an array of seven 32bit unsigned integers with the first element holding the array size (in this case six). These arrays are then transmitted over UDP to the infotainment system on port number 4284. These arrive at the receiving port as twenty-eight byte packets with the same format as ECUs that generated it (detailed above).

4) Infotainment Unit: In Fig. 1, the primary purpose of the simulated Infotainment unit is to act as a conduit for studying remote CAN bus attacks and as such its main function is to simply sit on the Ethernet and CAN networks. The unit will however be running some simple functions to demonstrate its main function and to simulate some essential infotainment features. To this end the infotainment unit will listen on a UDP port 4284 for any traffic which is designed to be UDP bundled CAN packets. The infotainment unit will perform some basic processing of these packets and will broadcast speed, direction and current location information destined for the remote In-vehicle telemetry display. The infotainment unit can be further extended to display more information from the CAN networks (e.g., Fuel Consumption or Battery Power remaining).

The infotainment unit receives every packet sent from the gateway and re-transmits telemetry data. Telemetry data is packaged into an array of six 32 bit signed integers. These arrays are then transmitted over UDP for the UI system on port number 8353. These arrive at the receiving port as twenty-four byte packets, that are processed and used to display the vehicles current state.

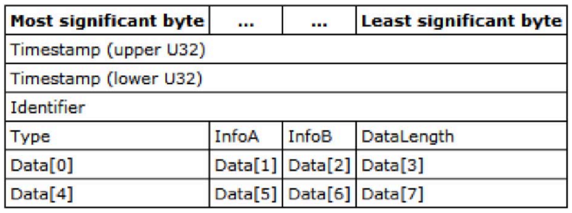
The infotainment unit contains three main components CAN_UDP_Daemon, CAN_UDP_Server, and CAN_Telemetry. The Infotainment unit is running as a service and include no user interface. This unit simulates the infotainment system inside a vehicle. Start/Stop of the unit is executed from a console window within the containing operating system.

CAN_UDP_Daemon runs automatically on start-up and performs the following actions:

1) Execute Daemon, the daemon is started. If an error occurs, it is logged and the daemon exits;
2) CAN_UDP_Server process running, the process checks if the CAN_UDP_Server process is running, if true the process sleeps for 30 seconds;
3) Start CAN_UDP_Server process, if the CAN_UDP_Server is found to be not running an attempt to start it is made. If an error occurs, it is logged and the daemon exits;
4) Exit, the exit method can intercept the kill signal (TERM) and exit cleanly.

CAN_UDP_Server is started by the CAN_UDP_Daemon and performs the following actions:

1) Read config, reads in the port number for this service;
2) Bind port, connects to a UDP socket with port number defined in the config File;
3) Listen to the port, waits for an incoming UDP connection;



Most significant byte	...	...	Least significant byte
Timestamp (upper U32)			
Timestamp (lower U32)			
Identifier			
Type	InfoA	InfoB	DataLength
Data[0]	Data[1]	Data[2]	Data[3]
Data[4]	Data[5]	Data[6]	Data[7]

Fig. 3: UDP Packet Format

4) Accept incoming connection, attempts to connect to the incoming client. If an error occurs, it is logged and the server exits;
5) Receive, reads the incoming UDP packet;
6) Parse CAN packet, attempts to parse the incoming UDP packet. If an error occurs, it is logged and the server exits;
7) Parsing incoming UDP packet, Parse and Assert that the packet data is an array of six unsigned 32 bit integers. The packet data is defined as in Fig. 3. The packet is an array of six unsigned 32 bit integer as conformed to the industry standard [7].
8) UDP_Telemetry process running, checks if the UDP_Telemetry process is running, if true the parsed packed is sent via a named pipe to the UDP_Telemetry process;
9) Start UDP_Telemetry process, if the UDP_Telemetry process is found to not be running an attempt to start it is made. If an error occurs, it is logged and the server exits;
10) Kill Server, the Kill Server method can intercept the kill signal (TERM) and exit cleanly.

UDP_Telemetry is started by CAN_UDP_Server and performs the following actions:

1) Read config, reads in the IP address and port number for this service;
2) Read FIFO, attempts to read the named pipe for the next packet. If an error occurs, it is logged and the process exits;
3) Process packet, the packet is processed. If an error occurs, it is logged and the process exits;
4) Kill Telemetry, the Kill Telemetry method can intercept the kill signal (TERM) and exit cleanly;
5) Processing incoming packet, parse and assert that the packet data is an array of six unsigned 32 bit integers;
6) Transmit, the processed data is broadcast the IP address and port number defined in the config file.

5) Telemetry Display: In Fig. 1, the in-vehicle telemetry display is a remote android display designed to receive broadcast telemetry data from the infotainment system on UDP port 8353. The in-vehicle telemetry display application is designed for handheld Android devices.

We designed the Telemetry Display unit as a separate unit outside the infotainment system to allow multiple display units to be integrated with the simulated infotainment unit, which allows more flexibility for security analysis.

A Nexus6p running Android Marshmallow 6.0.1 is acting as the telemetry display device to emulate the infotainment screen in a real vehicle. A custom telemetry application is installed using the TelemetrySim.apk file and contains the following functionality: Upon opening TelemetrySim the user is presented with a splash screen, on pressing the start button the Telemetry Server process is started and displays the live telemetry data on screen. One thing not covered from the Fig. 1 is an in-vehicle router, which provides the simulated vehicle with its 3G cellular connectivity. It paves the way for the future work where we can extend the testbed for vehicle to vehicle, vehicle to infrastructure application scenarios. In our testbed proproty used for the paper, we use a normal wifi router, which can be replaced with an actual router used inside a specific vehicle to be modelled.

In summary, today's modern vehicles have plethora of computers in the form of electronic control units (ECUs) and a mix of network types used for ECUs to communicate. This paper builds on a simulated test bed of such a network. Having both CAN and Ethernet nodes, with the two networks bridged via a CAN to Ethernet gateway.

The VCU and ECU units act as a modelled vehicle and are in continuous communication via the CAN bus. The data exchange inside the CAN bus might contain useful telemetry information that a (hypothetical) vehicle occupant may find useful so these CAN data can be broadcasted via the Gateway to an Infotainment unit designed in our testbed. The infotainment unit processes this data and transmits telemetry data over an included in-vehicle wi-fi that occupants can view with a specialized mobile platform application. This testbed thus provides a complete real-time simulation for a given real vehicle by integratin real-time CAN bus simulation with emulated infotainment system including an infotainment unit and a Telemetry display unit (both of which running on real computation devices).

IV. Experiment

A. Experiment Setup

In this section, we walk through the prototype we built based on the testbed architecture and the environment we use to evaluate the effectiveness of the testbed. We evaluate whether we can successfully capture all the CAN bus messages for security analysis and whether we can launch new security attack to the vehicle system through the infotainment gateway. Fig. 4 shows the overall experiment setup and we will walk through each key component in the section.

In this prototype, we deployed the Vehicle ECU unit to NI cRIO-9063 running NI Linux Real-Time (ARM-based) OS. We implemented a combined navigation and electric motor ECU which is able to control the speed and direction of the simulated vehicle. The specification of the ECU is provided by a real electric vehicle company. In our prototype, the ECU model takes the angle and force vector, and then calculates the current position and velocity, which is in turn bundled into a CAN packet and transmitted on the bus, which is to be processed by the Vehicle Control Unit. The architecture proposed in this paper is able to support multiple ECU units,

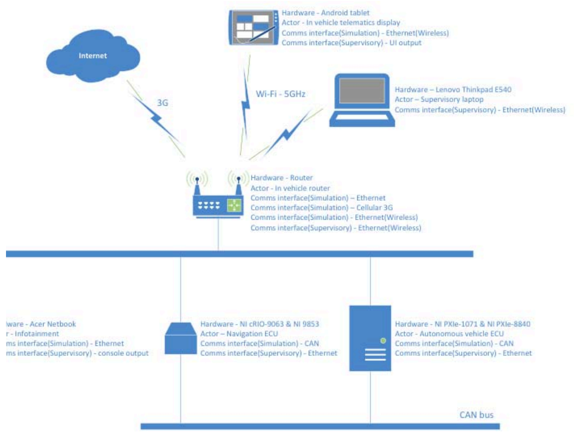

Internet
3G
Wi-Fi - 5GHz
Hardware - Android tablet
Actor - In vehicle telematics display
Comms interface(Simulation) - Ethernet(Wireless)
Comms interface(Supervisory) - UI output
Hardware – Lenovo Thinkpad E540
Actor – Supervisory laptop
Comms interface(Supervisory) - Ethernet(Wireless)
Hardware - Router
Actor - In vehicle router
Comms interface(Simulation) – Ethernet
Comms interface(Simulation) – Cellular 3G
Comms interface(Simulation) - Ethernet(Wireless)
Comms interface(Supervisory) - Ethernet(Wireless)
ware - Acer Netbook
r - Infotainment
ms interface(Simulation) - Ethernet
ms interface(Supervisory) - console output
Hardware - NI cRIO-9063 & NI 9853
Actor – Navigation ECU
Comms interface(Simulation) - CAN
Comms interface(Supervisory) - Ethernet
Hardware - NI PXIe-1071 & NI PXIe-8840
Actor - Autonomous vehicle ECU
Comms interface(Simulation) - CAN
Comms interface(Supervisory) - Ethernet
CAN bus

Fig. 4: Our Test Environment

with each ECU unit deployed separately to a dedicated device with real-time operating system (e.g., the NI cRIO-9063 device running NI Real-Time OS) and interconnected with a CAN bus interface (in the prototype NI 9853)

The specification of the VCU (the dynamic model of the vehicle) is also provided by the same electric vehicle company. In this prototype we only implement a simplistic version of VCU which shows in a user interface of the location of the simulated vehicle. Our next target is to integrate the simulated vehicle with real-world road map. The architecture provided in the paper also allows multiple vehicles to be integrated inside the same simulation environment. This is designed for verification against vehicle to vehicle and vehicle to infrastructure applications, details of which are discussed in Section V. The device to host the VCU is NI PXIe-8840 (CPU) installed in NI PXIe-1071 (PC Chasis). The device is running Phar Lap ETS OS. Each VCU device requires a CAN interface card, and in this prototype we use NI PXI-8512. We deploy the infotainment unit into this Acer Aspire One Pro laptop running the Ubuntu 16.10/4.8.11rt7. We deploy the Telemetry Display unit into a Samsung Nexus 6 mobile phone running Android Marshmallow 6.0.1. We use Router (ASUS RT-N66U) running 3.0.0.4.380_3831 as the in-vehicle router. The experiment aims to validate whether the developed solution conforms to the real system, satisfies its intended use for security analysis, and verifies the correct operation of the various elements of the simulation system. We covered two categories of testing covering Hardware and Software. The hardware testing is intended to verify the intercommunication of the devices of the system. The software testing will verify the modules developed can achieve what they are meant to.

The following software tools are used for the testing — Wireshark, Netcat, Tcpdump, Nmap/Zenmap, GNU Coreutils, Visual Studio 15 with Linux extension, LabVIEW 2015, and Android Studio.

We used the following evaluation metrics — at the hardware level, we checked whether each device can correctly boot, connect and communicate with other components; at the software level, we ensured that each software module correctly works according to the specifications described in Section. III.

B. Experiment Results

To validate the effectiveness of a CAN bus attack, we conducted a number of test cases. In each of our cases, the malicious attacker could hijack the maneuver of a running vehicle. That is, the packets transmitted on the CAN bus were captured by the attacker when the attacker accessed the CAN bus via a compromised ECU component. The attacker launched an Android App which communicated to the infotainment system of our testbed. Subsequently, the vehicle navigation data will be intercepted by the attacker. In this case, we merely used the two important types — heading and acceleration data. Due to the openness of the CAN bus, the attacker can then inject forged CAN bus packets back to the testbed CAN bus. Such forged messages may perform any of the following sabotaging actions:

1) swerve to an arbitrary direction to drive the vehicle out of the road;
2) swerve back and forth to roll over the vehicle;
3) accelerate suddenly to crash the vehicle to the object in front;
4) deaccelerate suddenly to crash the vehicle to the object behind;
5) Denial of Service (DoS) attack or
6) a combination of above actions.

In this paper, we successfully conducted DoS attack to the CAN bus on our testbed. Upon connection to the testbed, a malicious Android App began to broadcast a large number of fake CAN bus messages to the CAN bus. In fact, the App sent CAN bus packets of 6 bytes long with the arbitration id (the id for the ECU modelled) set to 1. The flood of such messages successfully disabled the routine operation of the CAN bus and caused the vehicle to stop responding to the driver's inputs. Subsequently, the vehicle would probably to be involved with a severe traffic accident. A demo of this successful attack has been summarized in a 15 minute Youtube clip available at https://youtu.be/qm80H3sVYnE.

More specifically, we captured the DoS attack packets at the gateway of our testbed. The captured data were in the string `0000 0006 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000`. That is, the DoS packets are of 6 bytes with arbitration id as 1 targeting the navigation unit on CAN0. During the period of the DoS attack, we recorded over $59,000$ CAN bus messages. The initial part of the dumped CAN bus messages looks as below:

```
2B D2DDE2B7 02 08 64007E99 00
2B D2DF0F57 02 08 64007E99 00
2B D2E05E31 02 08 64007E99 00
2B D2E18585 02 08 64007E99 00
2B D2E2C82E 02 08 64007E99 00
2B D2E410F0 02 08 64007E99 00
2B D2E536D9 02 08 66004870 00
2B D2E68CBB 02 08 66004870 00
2B D2E7A4E5 02 08 66004870 00
```

```
2B D2E8F460 02 08 66004870 00
2B D2EA2A12 02 08 66004870 00
2B D2EB4762 02 08 66004870 00
2B D2EC918F 02 08 66004870 00
2B D2EDB2EE 02 08 66004870 00
2B D2EEED02 02 08 66004870 00
2B D2F0449A 02 08 66004870 00
2B D2F172A5 02 08 66004870 00
2B D2F29C27 02 08 66004870 00
2B D2F3DB11 02 08 66004870 00
2B D2F5217A 02 08 66004870 00
2B D2F6338E 02 08 66004870 00
```

A complete copy of this dump is available at https://goo.gl/ucPmiQ. These messages clearly showed that the CAN bus was under a bombardment of regular CAN packets. It is worth noting that the first part of each CAN packet is slightly different to its neighbor message due to clock change. A complete version of the testbed prototype is available at https://github.com/bushman-rick/CAN_Gate.

In summary, the above case clearly showed the effectiveness of our testbed. We also were able to spend as little as ten minutes when we switched between different attack scenarios, which is a significant improvement than the existing solutions. Furthermore, our testbed is flexible enough to be used to test any other attacks launched from the cyber space via remote connections.

V. Conclusion

This paper presents a novel testbed to explore security vulnerability for modern vehicle systems which connects a real-time CAN bus simulation with an emulated infotainment system. We also present the architecture design and a prototype for the testbed so that others can reproduce the testbed for security analysis of the extended CAN bus with infotainment systems. In our experiment, we demonstrate the effectiveness of the testbed and initialize a new attack to the CAN bus of an electronic vehicle through the gateway. All the source codes, experimental data and video are all uploaded and shared for any team to use the testbed for their needs. Our next target is to include more ECUs in our prototype provided by a real vehicle company, integrate the simulated vehicle with real-world road map, and evaluate more attacks scenarios based on the testbed.

Acknowledgment

This work is supported by the China Postdoctoral Science Foundation (Grant No. 2016M591162) and the National Natural Science Foundation of China (Grant No. 51605242).

The authors thank Simon Lee Martin and Cameron Graham from Deakin Australia for help conducting tests and ZiWang Lu from Tsinghua University China for the simulation models used in the evaluation.

References

[1] Audi. Audi glossary MMI-Multi Media Interface. https://www.audiusa.com/technology/intelligence/mmi, 2016. [Online; accessed 4-April-2016].

[2] J. Bélanger, P. Venne, and J. N. Paquin. The what, where, and why of real-time simulation. In *Power and Energy Society*, 2010.

[3] BMW. BMW Technology Guide iDrive. http://www.bmw.com/com/en/insights/technology/technology_guide/articles/idrive.html, 2016. [Online; accessed 4-April-2016].

[4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[5] C. E. Everett and D. McCoy. Octane (open car testbed and network experiments): Bringing cyber-physical security research to researchers and students. In *CSET*, 2013.

[6] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive can networks–practical examples and selected short-term countermeasures. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2008.

[7] N. Instrument. CompactRIO Reference and Procedures (FPGA Interface). https://zone.ni.com/reference/en-XX/help/370984T-01/, 2016. [Online; accessed 5-April-2016].

[8] S. Köhl and D. Jegminat. How to do hardware-in-the-loop simulation right. Technical report, SAE Technical Paper, 2005.

[9] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, D. Shacham, et al. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[10] J. Kwon and K. Park. Development of the hardware-in-the loop simulator for evaluating performance of eps system. Technical report, SAE Technical Paper, 2011.

[11] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim. Fuzzing can packets into automobiles. In *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, pages 817–821. IEEE, 2015.

[12] C. Lin and A. Sangiovanni-Vincentelli. Cyber-security for the controller area network (can) communication protocol. In *Proc. of CyberSecurity*, pages 1–7. IEEE, 2012.

[13] C. Miller and C. Valasek. Adventures in automotive networks and control units. *DEF CON*, 21:260–264, 2013.

[14] C. Miller and C. Valasek. A survey of remote automotive attack surfaces. *black hat USA*, 2014.

[15] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015, 2015.

[16] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty. Security challenges in automotive hardware/software architecture design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 458–463. EDA Consortium, 2013.

[17] J. Sanchez-G., R. D'Aquila, W. Price, and J. Paserba. Variable time step, implicit integration for extended-term power system dynamic simulation. In *Proc. of PICA*, 1995.

[18] M. Shavit, A. Gryc, and R. Miucic. Firmware update over the air (fota) for automotive industry. Technical report, SAE Technical Paper, 2007.

[19] L. Yu and H. Zheng. The development and verification of hardware-in-the-loop test-bench of electrically controlled steering system. Technical report, SAE Technical Paper, 2015.

[20] H. Zheng and M. Zhao. Development a hil test bench for electrically controlled steering system. Technical report, SAE Technical Paper, 2016.