A Survey on Media Access Solutions for CAN Penetration Testing

Enrico Pozzobon
University of Applied Sciences Regensburg
Regensburg, Germany
Enrico.Pozzobon@othr.de

Nils Weiss
University of Applied Sciences Regensburg
Regensburg, Germany
Nils2.Weiss@othr.de

Sebastian Renner
University of Applied Sciences Regensburg
Regensburg, Germany
Sebastian1.Renner@othr.de

Rudolf Hackenberg
University of Applied Sciences Regensburg
Regensburg, Germany
Rudolf.Hackenberg@othr.de

ABSTRACT

Controller Area Network (CAN) is still the most used network technology in today's connected cars. Now and in the near future, penetration tests in the area of automotive security will still require tools for CAN media access. More and more open source automotive penetration tools and frameworks are presented by researchers on various conferences, all with different properties in terms of usability, features and supported use-cases. Choosing a proper tool for security investigations in automotive network poses a challenge, since lots of different solutions are available. This paper compares currently available CAN media access solutions and gives advice on competitive hard- and software tools for automotive penetration testing.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

CAN, Penetration Testing, Benchmarks

ACM Reference Format:
Enrico Pozzobon, Nils Weiss, Sebastian Renner, and Rudolf Hackenberg. 2018. A Survey on Media Access Solutions for CAN Penetration Testing. In *2nd Computer Science in Cars Symposium - Future Challenges in Artificial Intelligence & Security for Autonomous Vehicles (CSCS 2018), September 13–14, 2018, Munich, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3273946.3273949

1 INTRODUCTION

Since the automotive industry is moving towards autonomous driving, more and more cars are or will soon be connected to a backend system in the Internet, the security of automotive systems becomes a crucial factor in the process of developing self-driving cars. With additional remotely accessible interfaces, which are needed to enable the vehicle-to-internet communication, the attack surface of a modern car significantly increases. Furthermore, this development can expose protocols like CAN, which usually only have been used in the car's internal network, to remote ports. Since protocols used for intercommunication between Electronic Control Units (ECUs) were developed decades ago, they do not introduce any kind of mechanisms to secure the communication. The combination between the introduction of new connected interfaces, which are necessary for the use of self-driving features, and the use of legacy and potentially unsecured protocols creates a new high-impact risk in the context of attacks on the car's IT security. Therefore the evaluation of this risk is an important factor in the process of security testing. To ensure reliable and sufficient testing, professional tools specifically developed to support automotive protocols are needed.

This paper introduces the most commonly used frameworks, software and hardware available in the field of automotive security testing. Additionally, a method on how to cluster them into subgroups will be shown. Different criteria valid to conduct an extensive comparison between the chosen tools will be described, followed by a survey on the the actual tools. After explaining the test process for each subject under test, the test results will be concluded. The last paragraph will cover possible aspects that may be researched in the future.

2 RELATED WORK

In the area of performance evaluation of CAN drivers the work of Sojka et. al is mentionable. They performed an extensive timing analysis of the commonly used driver SocketCAN, in comparison with their own solution LinCAN on different Linux Kernels [24][23][8]. This differs from the research proposed in this paper, since it focuses solely on the driver itself, while we observe CAN media access devices as a whole. Further research regarding CAN tools was published by Lebrun et. al in 2016. Lebrun and Demay introduced CANSPY, a tool for CAN frame inspection and manipulation, especially built to aid with security evaluations of CAN devices [13].

Besides the work done on CAN testing, surveys relevant to analyzing security tools have been conducted in the field of web applications. For example, Fonseca et. al benchmarked web vulnerability scanning tools using criteria such as vulnerability coverage and the false positive rate [3]. Doupe et. al accomplished similar research

CSCS 2018, September 13–14, 2018, Munich, Germany

ACM ISBN 978-1-4503-6616-8/18/09…$15.00
https://doi.org/10.1145/3273946.3273949

with the evaluation of vulnerability scanners for web applications in 2010 [1].

3 GENERAL REQUIREMENTS FOR AUTOMOTIVE PENETRATION TESTING

This section introduces some basic requirements for automotive penetration testing. A competitive tool should fulfill the following requirements. All mentioned requirements are derived from practical use cases in automotive penetration testing. The requirements describe features that are elementary to start with automotive penetration testing and therefore they can be seen as a needed foundation in the domain of automotive security analysis on CAN.

3.1 Protocol Viewer

A clear way to inspect the traffic on the network is absolutely necessary to improve the efficiency of automotive security investigations. A security researcher needs an interface which displays the live traffic and allows detailed investigations on captured data. A penetration tool has to detect and interpret the used protocol automatically. In addition to that, such a viewer needs to be extensible with additional brand specific protocol information. The following protocols should be supported.

- CAN (ISO 11898)
- Unified Diagnostic Services (UDS) (ISO 14229)
- General Motor Local Area Network (GMLAN)
- CAN Calibration Protocol (CCP)
- Universal Measurement and Calibration Protocol (XCP)
- ISO-TP (ISO 15765)
- Diagnostic over Internet Protocol (IP) (DoIP) (ISO 13400)
- On-board diagnostics (OBD) (ISO 15031)
- Ethernet (ISO 8802/3)
- TCP/IP (RFC 793, RFC 7323 / RFC 791, RFC 2460)

3.2 Packet Manipulation and Fuzzing

Fuzzing describes the process of automatically feeding semi-random input values to a system and observing its behavior in response to different inputs. This technique can help in discovering for example flaws of a protocol's implementation. An automotive penetration test tool should be able to fuzz all common automotive and Internet protocols. A fuzzer with an easy-to-use interface and the capability of fuzzing and listening on different network interfaces at the same time should be available in a penetration test tool. The desired response of a fuzzed message often shows up on a different interface, maybe with a different protocol, or on output pins of an ECU.

3.3 ECU Simulation

Usually, an ECU under test needs a special environment for its normal operation. This environment has to be simulated with periodic CAN and UDS messages on a connected bus. For more advanced investigations, a remaining bus simulation is required. Vector CANoe for example is specialized in this kind of simulation. For white-box security tests, this would be the tool of choice for complex remaining bus simulations [5]. On black- and gray-box security tests, periodic messages are sufficient to spoof a certain operation state. Capturing, modifying and periodically replaying of various messages has to be supported by a penetration test tool.

3.4 Man in the Middle (MITM) Capabilities

In a MITM investigation the penetration device is used to hijack the connection between two or more ECUs and route the occurring traffic. The main goal is to gather more information about the CAN communication and possibly filtering unwanted messages. In order to investigate the communication from a specific ECU, this ECU has to be isolated through a MITM attack. An advanced penetration test tool needs some functionality to setup a MITM proxy between two CAN or Ethernet interfaces. In addition, functions to filter or hijack the communication between ECUs are very useful during black-box security investigations.

4 MEDIA ACCESS LAYER REQUIREMENTS

The mentioned higher level requirements are necessary for efficient penetration testing in automotive networks. It is possible to derive the following media access requirements from this previously mentioned general requirements. For verification of media access layer requirements, multiple benchmarks will be created on available interfaces for the media access to CAN. A performance evaluation of media access interfaces for automotive penetration tests will be introduced in section 8.

4.1 Latency

The required time between the initiation of a write to an automotive network from an user space application until the actual presence of this data on the bus is crucial for all kind of fuzzing and spoofing tests. For example, if one wants to respond to a certain message faster than the legitimate ECU, the latency of the media access layer has to be smaller than the time it takes for the legitimate ECU to generate a response.

4.2 TX-Rate

During the investigation of denial of service or flooding attacks on the CAN bus, a penetration test tool should be capable of creating enough CAN frames to reach a bus load of 100 percent. To achieve this load from user space writes, the interface driver, the operating system and the media access device itself need to be able to handle writes to the bus faster than the time a message is present on the bus.

4.3 RX-Rate

A common use case in automotive penetration testing is the sniffing of firmware updates on the CAN bus. During firmware updates of ECUs on the CAN bus, usually only the target ECU and the ECU or the repair shop tester which is delivering an update are allowed to communicate on the bus. All other ECUs remain silent until the flashing procedure finishes. This dedicated communication between only two ECUs on the bus guarantees maximum bandwidth for the firmware transfer. This also leads to a maximum bus load. A media access device for penetration tests, the operating system and the used drivers need to be capable of handing over this bandwidth to a user space application. Otherwise, the penetration tester is missing

messages due to performance issues of the device, which leads to inconsistent results.

4.4 Reliability

For any security investigation in automotive networks, the penetration test tool needs to be reliable. Often, traces and captures can be done only once or require extensive preparations to the ECU under test. Unreliable tools will make difficult security investigations in automotive networks even harder or lead to wrong results.

5 CLASSIFICATION OF MEDIA ACCESS TYPES

To get an overview of existing open-source automotive penetration test frameworks, a survey on existing applications and tools for automotive penetration tests was conducted. This survey focused on hard- and software interfaces to access automotive networks. The following table gives an overview of common used solutions for the media access to CAN.

	SLCAN Device	ELM327	SocketCAN	python-can	Others
Busmaster [16]					X
c0f [21]			X		
can-utils [17]			X		
CANiBUS [20]	X				X
CANToolz [19]	X				X
Caring Caribou [18]				X	
Kayak [14]			X		
Metasploit [15]		X	X		
O2OO [29]		X			
pyfuzz_can [10]				X	
python-OBD [30]		X			
Scapy [28]			X	X	
UDSIM [22]			X		
Wireshark [31]			X		

Table 1: Overview of used hard- and software for CAN-bus access in open-source automotive penetration test frameworks [9][11][26].

With respect to the evaluation of used media access interfaces on public available penetration test frameworks, the most used media access interfaces will be taken into account on a performance evaluation. In order to be able to compare available CAN media access solutions for automotive penetration testing, representative groups for the different interface solutions have been selected. Devices inside a group have an identical hard- and software architecture, and will therefore show a similar behavior during benchmarks.

The following list shows the chosen groups for the media access layer comparison of CAN-bus interfaces:

(1) **Native-CAN**
The CAN-peripheral module is directly accessible from the main processor. A BeagleBone Black (BBB) with an AM335x 1 GHz ARM Cortex-A8 processor and a Banana Pi Pro with an Allwinner A20 dual core ARM Cortex-A7 processor are the used devices under test in this group. All automotive penetration test frameworks which use SocketCAN or python-can are able to use lower layer CAN-bus interfaces from this group.

(2) **Serial Peripherial Interface (SPI)-to-CAN**
The CAN-peripheral module is accessible over a SPI connection. A Raspberry Pi 2 with a MCP2515 SPI-CAN module is used as Device Under Test (DUT). SocketCAN is the common way to give an user space application access to the CAN-bus in this group.

(3) **Universal Serial Bus (USB)-to-CAN over Serial Line CAN (SLCAN)**
The CAN-peripheral module is accessible over a serial line communication. The SLCAN protocol is used to access the CAN-bus from the DUT. An USBtin is used as an interface in this class. User space applications can either access the CAN-bus directly over a serial connection, or can connect to a SocketCAN socket, which is offered by an application called slcand [2].

(4) **ELM327**
The ELM327 is an OBD-to-serial interface with a custom AT command set. As a DUT, an ELM327 with USB-interface is used. The CAN-bus is accessible over a serial interface.

(5) **USB-to-CAN**
The CAN-peripheral is accessed over a USB-interface in this class. As devices under test, a PEAK PCAN-USB FD and a Vector VN1611 USB-to-CAN adapter are used. Devices inside this group can require proprietary drivers. For user space applications, the CAN-bus is accessible through dynamic linked libraries supported by python-can or SocketCAN sockets [4][7].

6 BENCHMARKING CRITERIA AND PROCESS

6.1 Test-Setups

Every media access device has to be tested individually and with a different setup. All tests on media access devices were performed by user space applications, running with maximum priority by the root/administrator user and compiled with maximum optimization settings (where applicable). All the tests involving a USB-to-CAN device were executed on the same laptop computer. All operating systems were tested with the minimum amount of modifications from the moment of installation, limited to the installation of the required software to run the tests, drivers for the CAN interfaces, and device trees for the Native-CAN implementations.

- **TI AM3358 (Native-CAN)**
  - Platform: BBB
  - OS: Debian GNU/Linux 8
  - Kernel: 4.4.54-ti-r93 #1 SMP
  - User space application: Compiled program, written in C, using SocketCAN
- **Allwinner A20 (Native-CAN)**
  - Platform: Lemaker Banana Pro
  - OS: Armbian GNU/Linux 9
  - Kernel: 4.14.18-sunxi #24 SMP

  - User space application: Compiled program, written in C, using SocketCAN
- **MCP2515 (SPI-to-CAN)**
  - Platform: Raspberry Pi 2 Model B
  - OS: Raspbian GNU/Linux 9
  - Platform-specific settings: SPI clock frequency set to 8 MHz
  - Kernel: 4.14.30-v7+ #1102 SMP
  - User space application: Compiled program, written in C, using SocketCAN
- **USBtin (USB-to-CAN over SLCAN)**
  - Platform: Dell Latitude E5470
  - OS: Antergos Linux
  - Kernel: 4.15.15-1-ARCH #1 SMP PREEMPT
  - User space application: Compiled program, written in C, using SocketCAN
- **ELM327 (USB-to-OBD)**
  - Platform: Dell Latitude E5470
  - OS: Antergos Linux
  - Platform-specific settings: UART baud rate set to 460.8 kHz
  - Kernel: 4.15.15-1-ARCH #1 SMP PREEMPT
  - User space application: Compiled program, written in C, using glibc termios
- **PEAK PCAN-USB FD (USB-to-CAN)**
  - Platform: Dell Latitude E5470
  - OS: Antergos Linux
  - Kernel: 4.15.15-1-ARCH #1 SMP PREEMPT
  - User space application: Compiled program, written in C, using SocketCAN on Linux
- **PEAK PCAN-USB FD (USB-to-CAN)**
  - Platform: Dell Latitude E5470
  - OS: Windows 10
  - User space application: Python script
- **Vector VN1611 (USB-to-CAN)**
  - Platform: Dell Latitude E5470
  - OS: Windows 10
  - User space application: Python script

6.2 Test Procedure

All tests were executed with the maximum baud rate supported by the tested device. This maximum baud rate was 1 MHz for every device except for the ELM327 which only supports a maximum rate of 500 kHz.

All tests involved connecting the DUT to a microcontroller and a logic analyzer for taking measurements. The logic analyzer was connected to the microcontroller CAN TX and RX pins, in parallel to the transceiver. The length of the bus connecting the tested device to the microcontroller was 1.2 meters.

Testing some aspects of the ELM327 was impossible, since it is supposed to be used as a OBD interface and has limited ability to work with raw CAN messages. In particular, it can only send raw CAN frames with 8 bytes of payload, and can not receive CAN frames with a data length of 0 bytes. Because of these reasons, only the latency was tested for this interface.

6.2.1 Latency Test. The objective of this test is to measure the amount of time taken by each tool and framework to forward a CAN frame from the physical layer to the user application and vice versa.

In order to test for latency, a microcontroller with CAN capabilities (Espressif ESP32) was connected to the bus and set to send a CAN frame every 20 ms, while the DUT was configured to receive these CAN frames and reply as soon as possible with another frame.[25] A logic analyzer was connected to the receive and transmit lines between the transceiver and the microcontroller, and set to sample data at 10 million samples per second. Since the maximum CAN baud rate reachable by the tested devices is 1 MHz, a sampling rate of 10 MHz is sufficient to capture and parse the individual CAN frames. A C application was developed using the Sigrok library to interface with the logic analyzer and capture precise timings of the time between a CAN frame coming from the microcontroller and the response coming from the DUT [27].

Given a pair of "request" and "response" CAN frames (where the request is sent by the microcontroller and the response is sent by the DUT), we define the latency introduced by the tested device and framework to be the time passed from the "ACK" field of the request CAN frame and the "START OF FRAME" field of the response frame. During this time, the frame is received by the hardware of the DUT and it is forwarded to the user space application, which immediately generates the response CAN frame without any processing or delay. The measured latency therefore represents the time which a CAN frame takes to travel from the physical layer to the application layer and back to the physical layer.

It is notable, that the latency measured in this way can never be lower than ten baud lengths according to the CAN specification, since between the "ACK" field and the "START OF FRAME" field of the next frame there always has to be an "END OF FRAME" field consisting of seven recessive bits and an "INTERMISSION" field consisting of three recessive bits. However, while achievable on a microcontroller, such a low latency was never obtained on any of the tested devices, since they all involve an user space application running on a non real-time operating system.

6.2.2 TX-Rate Test. The objective of this test is determining how fast a given device can write frames to the CAN bus, and what is the maximum data rate it can transmit at. Such a test is important to verify that a device is capable of flooding the bus with frames and simulating a high load on the bus.

To execute this test, a small program was written for each tested software stack that would simply send the same CAN frame one million times in a loop, while connected to a microcontroller that would only acknowledge every frame. The CAN lines were probed with a logic analyzer to measure the time passed between each frame and statistical data was extracted from these measures.

The transmission rate results are expressed in kb/s. This refers to the amount of useful information bits that are sent in one second, which includes the 11 bits of the standard identifier plus any bit stored in the data fields.

The test was repeated with two different kinds of CAN frames: Frames without a payload and frames with a maximum payload. The smaller frames, which still have an effective amount of data 11 bits in the identifier (contained in the "ARBITRATION FIELD"), were used to test the maximum frame rate achievable by each device. The longer frames which contain 75 bits of data were used to test

the maximum achievable bit rate of each device. The maximum achievable rates on a CAN bus with baud rate of 1 MHz are 675.6757 kb/s for the test involving longer frames, and 229.1667 kb/s for the test with the shorter frames.

In this test, there was no noticeable difference in the results when testing a high-level framework (e.g. Scapy) or the low layer implementation (e.g. SocketCAN), due to the amount of buffering done by the operating systems and the relatively low data rate of the CAN bus when compared to the speed which frames are processed at. Therefore, only one test setup result is shown for each tested device.

6.2.3 RX-Rate Test. The objective of this test was determining how fast a given device can read frames from the CAN bus, and what maximum data rate it can receive at. This test is important for determining if a device can receive all CAN frames in a situation of high bus load, such as when an ECU firmware is transmitted as part of a software update.

To execute this test, the microcontroller was programmed to output a CAN frame one million times with the smallest delay allowed by the CAN specification between one frame and the next. In order to achieve maximum consistency in the transmission interval, as well as sending the same number of frames whether they were being acknowledged or not, the CAN frames were transmitted using the Inter-Integrated Circuit (IC) Sound (I2S) peripheral of the ESP32 microcontroller.

The test was repeated multiple times for each device, with both short (no payload) and long (8 bytes payload) frames. The time between the start of two consecutive frames was 48 bauds for the short frames, and 111 bauds for the long frames, giving an effective bit rate of 675.6757 kb/s for long frames and 229.1667 kb/s with short frames. During each test sequence, the number of frames received by the DUT was recorded with a simple program, and the test was repeated 1000 times, for a total of 10^9 frames sent to each device for both long and short frames.

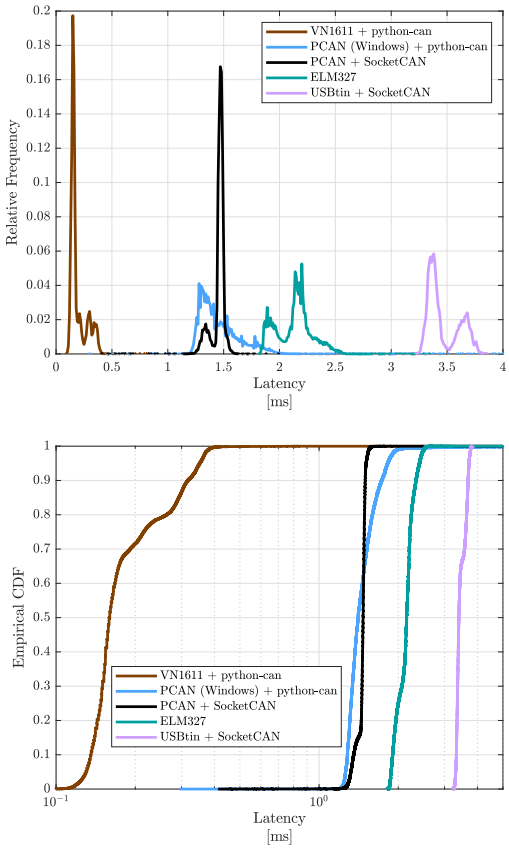

Figure 1: Latency histogram and Empirical distribution function for USB-to-CAN devices, all tested on the same laptop computer.

7 EVALUATION OF MEDIA ACCESS DEVICES

7.1 Latency Evaluation

The results for the latency tests are presented in the form of histograms, Empirical cumulative distribution functions (ECDFs) and box plots.

The vertical axis of the histogram represents the relative frequency of the time measured in one request-response pair. The most desirable result would be to have a single sharp peak centered as close to 0 ms as possible and a wide curve indicates a large variance in the measurements. The presence of multiple peaks in many histograms might indicate that a buffer is being filled asynchronously with the received CAN frames and it is only being read after a timeout expires, or it might be due to the operating system executing code for handling interrupts from other peripherals.

The ECDF plots present the integral of the data shown in the histograms. Given a point (x, y) in these plots, y is the probability that a frame was replied to in time less than x ms. The intersection between the curve and the 0.5 horizontal line in the ECDF plot shows the average latency for that device, while the upmost part represents the worst case scenario.

All SBCs were able to reply to CAN frames with less than one millisecond of latency, with the Banana Pro consistenly achieving times under the tenth of a millisecond.

7.2 TX-Rate Evaluation

When testing the transmission speed of the PEAK PCAN-USB FD and Vector VN1611 USB-to-CAN interfaces, on any framework or operating system, the performance was always good enough to obtain a 100% utilization of the bus. Any variance found in the results of these interfaces is only present in the first couple of frames, after which the buffer becomes full and any subsequent sent frame will enter the bus as soon as it becomes free.

SBCs with native interfaces could almost constantly send enough CAN frames to reach 100% bus load, but had a larger variance in the results, likely due to other processes sharing time on the limited CPUs.

The bottleneck for the transmission speed of the USBtin is the emulated USB Universal Asynchronous Receiver/Transmitter (UART) device, which appears to operate at a maximum speed of 411 kb/s. The speed is further limited by the overhead of the SLCAN protocol, which is introduced by the use of hexadecimal characters and

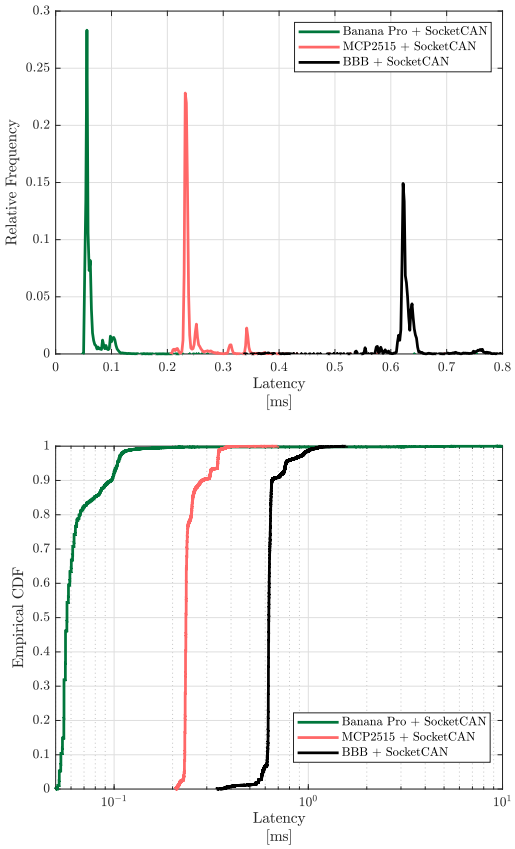

Figure 2: Latency histogram and Empirical distribution function for Single Board Computers (SBCs) using SocketCAN.

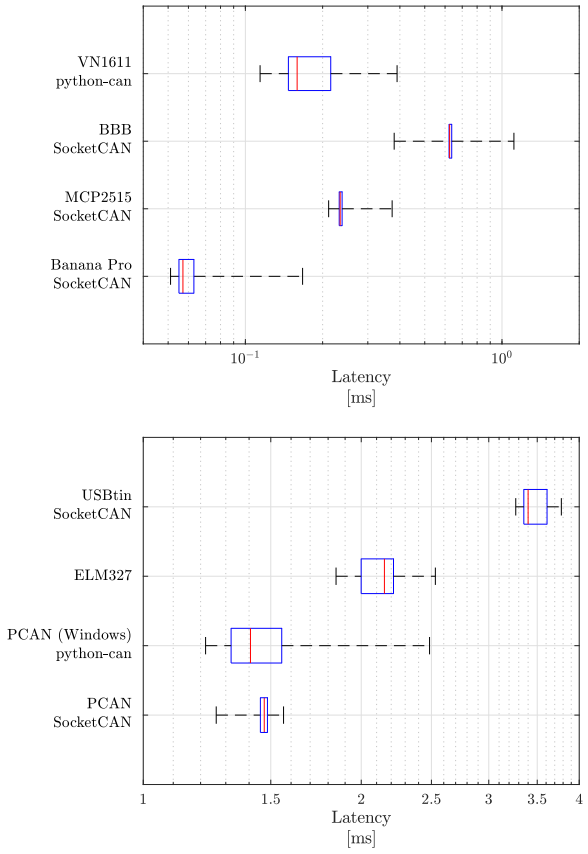

Figure 3: Box plots of the measured latency for the tested devices. Whiskers represent the 5‰ and 995‰ quantiles.

	$\mu\ [kb/s]$	$\sigma\ [kb/s]$
USBtin + SocketCAN	57.9261	0.94254
Raspberry Pi 2 MCP2515 + SocketCAN	66.0764	4.0136
BBB + SocketCAN	225.2256	14.7024
Banana Pro + SocketCAN	209.29	7.054
PCAN-USB FD + SocketCAN	229.1065	0.046076
PCAN-USB FD + python-can on windows	228.5465	1.6716
VN1611 + python-can	228.568	0.042923

Table 2: Results for TX-Rate evaluation with short CAN-frames on the tested media access devices. σ is the standard deviation computed on the taken measurements.

	$\mu\ [kb/s]$	$\sigma\ [kb/s]$
USBtin + SocketCAN	138.7733	7.3381
Raspberry Pi 2 MCP2515 + SocketCAN	348.8258	9.6974
BBB + SocketCAN	672.3208	12.6014
Banana Pro + SocketCAN	649.1451	5.1531
PCAN-USB FD + SocketCAN	675.5985	0.2309
PCAN-USB FD + python-can on windows	674.9153	0.052812
VN1611 + python-can	674.9029	0.13812

Table 3: Results for TX-Rate evaluation with long CAN-frames on the tested media access devices.

the framing of SLCAN commands, more than halving the effective bitrate.

The MCP2515 SPI-to-CAN interface has also a bottleneck. While the speed of the SPI interface would be sufficient to transfer all the commands necessary to send a CAN frame in a time smaller than the duration of the CAN frame itself, each successful transmission enables the TX0IF flag in the CANINTF register of the MCP2515 [12]. When this happens, the interrupt pin is signaled and the MCP2515 driver on the operating system will query the interface for the pending interrupt. The three SPI commands necessary for reading and resetting the TX0IF flag and the interruption caused by the driver amount to a total overhead of approximately 70 μs for each sent CAN frame when the SPI clock frequency is set to 8 MHz.

7.3 RX-Rate Evaluation

Both Vector VN1611 and PEAK PCAN-USB FD did not miss a single frame out of the 10^9 when running on windows, which is expected because of the large amount of free Random Access Memory (RAM) to be used as buffer and the low overhead introduced by USB. When running on Linux, approximately 75000 packets were not received in the test with short frames for unknown reasons, while the test with long frames showed no lost frames. Out of the 1000 tests executed with 10^6 frames each using the PEAK PCAN-USB FD

	$\mu\ [kb/s]$	$\sigma\ [kb/s]$
USBtin + SocketCAN	89.0387	0.0001
Raspberry Pi 2 MCP2515 + SocketCAN	217.5796	2.8753
BBB + SocketCAN	94.3395	1.4791
Banana Pro + SocketCAN	228.855	1.5459
PCAN-USB FD + SocketCAN	229.1495	0.028675
PCAN-USB FD + python-can on windows	229.1667	0
VN1611 + python-can	229.1667	0

Table 4: Results for RX-Rate evaluation with short CAN-frames on the tested media access devices.

	$\mu\ [kb/s]$	$\sigma\ [kb/s]$
USBtin + SocketCAN	179.9568	0.0020
Raspberry Pi 2 MCP2515 + SocketCAN	482.6823	16.6219
BBB + SocketCAN	674.4699	2.2591
Banana Pro + SocketCAN	675.5324	0.30141
PCAN-USB FD + SocketCAN	675.6757	0
PCAN-USB FD + python-can on windows	675.6757	0
VN1611 + python-can	675.6757	0

Table 5: Results for RX-Rate evaluation with long CAN-frames on the tested media access devices.

under Linux, 535 successfully received all frames, while the others lost an average of 161 frames each.

While SBCs with Native-CAN interfaces performed acceptably with long frames, with both missing less than 0.2% of the frames, the BBB struggled with short frames. More than half of the short frames were not received by the BBB, likely due to the single core CPU being overwhelmed by the large number of interrupts. This result highlights how important it is to have more than one core, since the Allwinner A20 on the Banana Pro only lost 0.14% of the frames.

Similarly to what happened in the transmission speed test, the speed of the MCP2515 is bottlenecked by the way the interrupt flags are handled. The number of missed frames on the MCP2515 is higher for long frames than it is for short frames. This is caused by the interrupt flag not being turned off in the short frames test, because there is no time to do so between a frame and the next. With long frames, the interrupt flag is turned off by the driver because no new frame is received immediately and this requires the transmission of three extra SPI commands and consequently a waste of time that causes the loss of an incoming frame. Interestingly, lowering the SPI clock frequency improves the speed at which the MCP2515 driver receives frames because it causes less switches of the interrupt flags.

SLCAN is once again the bottleneck for the USBtin, which missed a large amount of both long and small frames. It is notable that there is very little variance in the results gathered from the USBtin, since in every single test it received exactly either 388532 or 388533 short frames, and either 266333 or 266339 long frames.

8 EVALUATION OF PENETRATION TEST FRAMEWORKS

While on the laptop computer Scapy introduces a fraction of a millisecond of latency, it affects the less powerful SBCs in a harder way, adding several milliseconds.

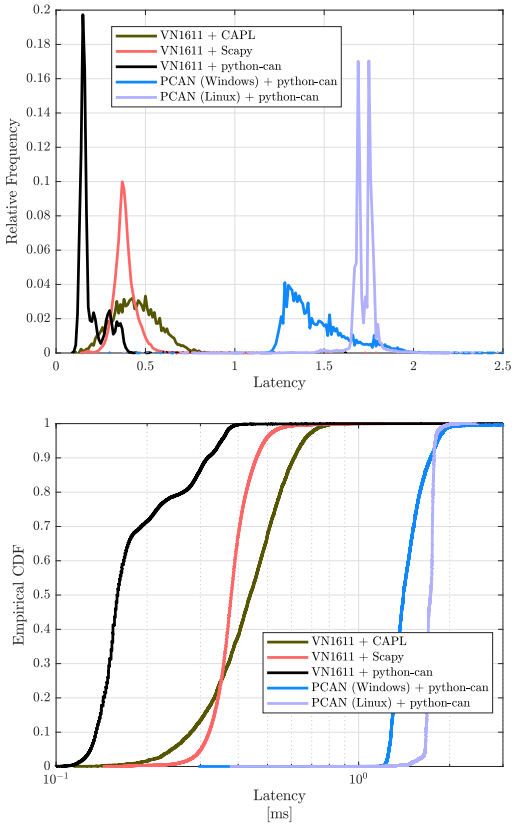

Figure 4: Latency histogram and Empirical distribution function for frameworks using USB-to-CAN devices, all tested on the same laptop computer.

Also can-utils as a Linux framework was tested, by using a bash script to read a single CAN frame with candump and sending a response with cansend. However, the overhead introduced by the creation of a new process for every sent frame renders can-utils always inferior to any other framework on any platform.

The CAPL scripting language by Vector is limited when compared to other general purpose languages like python, and doesn't show better results in terms of latency than Scapy or python-can [6].

9 CONCLUSION AND FUTURE WORK

None of the investigated tools and frameworks showed an error-free performance on Linux. Proprietary tools on Windows performed very good overall. The python-can framework delivered good results on both tested operating systems, Linux and Windows. Furthermore, this framework supports both professional and low-cost media access devices.

Very popular tools like the ELM327 or USB-to-CAN over SLCAN devices had a bad outcome in our performance evaluation. These tools aren't usable for advanced penetration tests with higher CAN baud rates.

SBCs are a very good alternative to professional CAN media access devices. The availability of a full-featured operating system

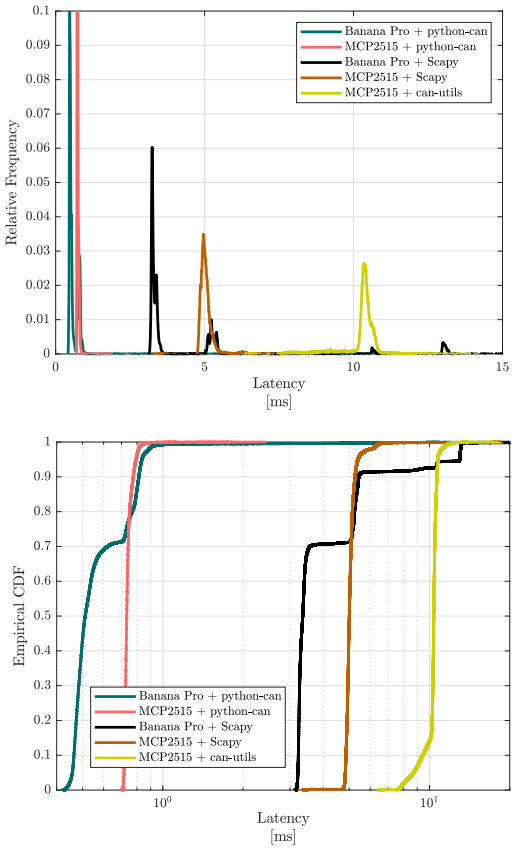

Figure 5: Latency histogram and Empirical distribution function for SBCs using different frameworks.

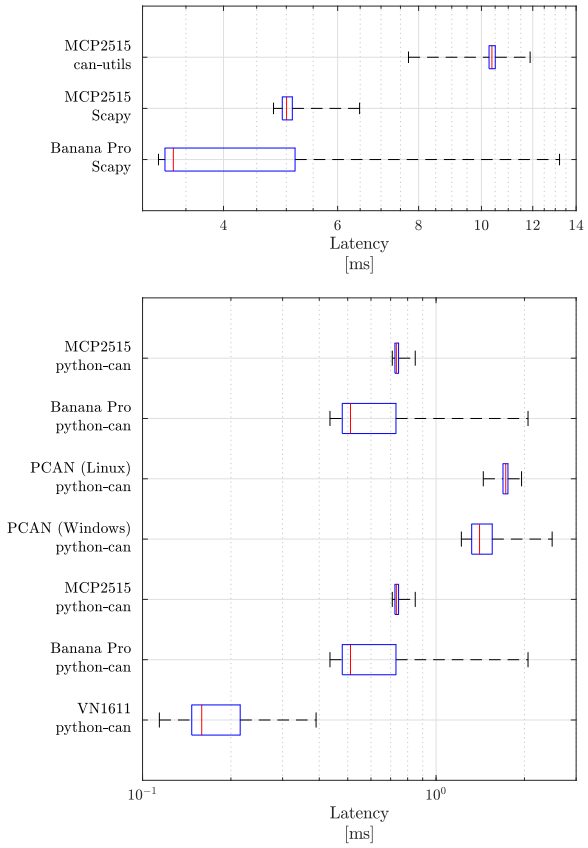

Figure 6: Box plots of the measured latency for the tested frameworks. Whiskers represent the 5‰ and 995‰ quantiles.

creates a high level of usability in a wide variety of applications. The tested devices are available for less than 50 euros. This fact brings SBCs into the same price range as popular CAN interfaces like the ELM327 or USB-to-CAN over SLCAN devices. However, running advanced frameworks like Scapy on these low-power computers introduces a large overhead.

Professional CAN tools for automotive engineering tasks showed the best results in terms of reliability and transmission speed. However, these tools are of course much more expensive than SBCs.

Inspected automotive penetration test frameworks showed only very limited penetration test capabilities. Most tools only support one specific use case. An open source tool with support of various car brands and proprietary automotive protocols is not available yet. On the other hand, commercial tools do not perfectly fit for penetration testing tasks and are very expensive in general. Also the fact that commercial tools are closed source software products decreases the suitability and flexibility for specific penetration tests. To improve automotive penetration testing in general, comprehensive open source tools which fulfill the mentioned requirements from section 3 are necessary. This paper showed that the soft- and hardware interfaces, python-can and SocketCAN, are appropriate for the implementation of advanced automotive penetration frameworks.

In the future, our research in the area of automotive penetration testing tools will be focused on developing a new framework based on existing open-source software. The experience gathered during the work for this paper will serve as a base of requirements when designing the tool. Later, similar tests will be conducted to evaluate and verify the framework's performance.

REFERENCES

[1] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 111–131.

[2] Thomas Fischl. 2018. USBtin - USB to CAN interface. (2018). Retrieved April 16, 2018 from http://www.fischl.de/usbtin/

[3] Jose Fonseca, Marco Vieira, and Henrique Madeira. 2007. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*. IEEE, 365–372.

[4] PEAK-System Technik GmbH. 2018. PCAN-USB FD - CAN-FD-Interface für High-Speed-USB 2.0. (2018). Retrieved April 16, 2018 from https://www.peak-system.com/PCAN-USB-FD.365.0.html#

[5] Vector Informatik GmbH. 2018. CANoe 11.0. (2018). Retrieved April 16, 2018 from https://vector.com/vi_canoe_de.html

[6] Vector Informatik GmbH. 2018. CAPL Documentation. (2018). Retrieved April 16, 2018 from https://kb.vector.com/entry/48/

[7] Vector Informatik GmbH. 2018. VN1600 - Flexible, kosteneffiziente Bus-Interfaces mit USB-Schnittstelle für CAN, CAN FD, LIN, K-Line, J1708 und IO. (2018). Retrieved April 16, 2018 from https://vector.com/vi_vn1600_de.html

[8] Oliver Hartkopp. 2010. Readme file for the Controller Area Network Protocol Family (aka SocketCAN). (2010). Retrieved April 16, 2018 from https://www.kernel.org/doc/Documentation/networking/can.txt

[9] Oliver Hartkopp. 2010. slcan.c - serial line CAN interface driver (using tty line discipline). (2010). Retrieved April 16, 2018 from https://github.com/torvalds/linux/blob/master/drivers/net/can/slcan.c

[10] Bill Hass. 2018. pyfuzz_can. (2018). Retrieved April 16, 2018 from https://github.com/bhass1/pyfuzz_can

[11] Elm Electronics Inc. 2008. ELM327 - OBD to RS232 Interpreter. (2008). Retrieved April 16, 2018 from https://www.elmelectronics.com/wp-content/uploads/2016/07/ELM327DSF.pdf

[12] Microchip Technology Inc. 2005. MCP2515 - Stand-Alone CAN Controller With SPI Interface. (2005). Retrieved April 16, 2018 from http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf

[13] Arnaud Lebrun and Jonathan-Christofer Demay. 2016. Canspy: a platform for auditing can devices. (2016).

[14] Jan-Niklas Meier. 2014. Kayak. (2014). Retrieved April 16, 2018 from http://kayak.2codeornot2code.org/

[15] Rapid7. 2018. Metasploit. (2018). Retrieved April 16, 2018 from https://www.metasploit.com/

[16] RBEI and ETAS. 2017. BUSMASTER. (2017). Retrieved April 16, 2018 from https://github.com/rbei-etas/busmaster/

[17] Volkswagen Group Electronic Research. 2018. Linux-CAN / SocketCAN user space applications. (2018). Retrieved April 16, 2018 from https://github.com/linux-can/can-utils

[18] Christian Sandberg, Kasper Karlsson, Tobias Lans, Mattias Jidhage, Johannes Weschke, and Filip Hesslund. 2018. Caring Caribou. (2018). Retrieved April 16, 2018 from https://github.com/CaringCaribou/caringcaribou

[19] Alexey Sintsov. 2017. CANToolz - framework for black-box CAN network analysis. (2017). Retrieved April 16, 2018 from https://github.com/CANToolz/CANToolz

[20] Craig Smith. 2013. CAN Device Vehicle Research Server (OpenGarages.org). (2013). Retrieved April 16, 2018 from https://github.com/Hive13/CANiBUS

[21] Craig Smith. 2015. CAN of Fingers (c0f). (2015). Retrieved April 16, 2018 from https://github.com/zombieCraig/c0f

[22] Craig Smith. 2017. UDSim. (2017). Retrieved April 16, 2018 from https://github.com/zombieCraig/UDSim

[23] Michal Sojka, Pavel Píša, Ondřej Špinka, and Zdeněk Hanzálek. 2011. Measurement automation and result processing in timing analysis of a Linux-based CAN-to-CAN gateway. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, Vol. 2. IEEE, 963–968.

[24] M. Sojka, P. Píša, M. Petera, O. Špinka, and Z. Hanzálek. 2010. A comparison of Linux CAN drivers and their applications. In *International Symposium on Industrial Embedded System (SIES)*. 18–27. https://doi.org/10.1109/SIES.2010.5551367

[25] Espressif Systems. 2018. Espressif Systems ESP32. (2018). Retrieved April 16, 2018 from https://www.espressif.com/en/products/hardware/esp32/overview

[26] Brian Thorne. 2018. python-can - The can package provides controller area network support for Python developers. (2018). Retrieved April 16, 2018 from https://github.com/hardbyte/python-can

[27] Bert Vermeulen Uwe Hermann. 2018. Sigrok. (2018). Retrieved April 16, 2018 from https://sigrok.org/

[28] Guillaume Valadon and Pierre Lalet. 2018. Scapy. (2018). Retrieved April 16, 2018 from http://www.secdev.org/projects/scapy/

[29] Folkert van Heusden. 2014. O2OO. (2014). Retrieved April 16, 2018 from https://www.vanheusden.com/O2OO/

[30] Brendan Whitfield. 2016. python-OBD. (2016). Retrieved April 16, 2018 from https://github.com/brendan-w/python-OBD

[31] Peter Wu. 2018. Wireshark. (2018). Retrieved April 16, 2018 from https://www.wireshark.org/