Testing the boundaries of the Parrot anti-spoofing defense system

Tsvika Dagan
Tel Aviv University
tdagan02@gmail.com

Avishai Wool
Tel Aviv University
yash@eng.tau.ac.il

Abstract

This paper describes an extended set of experiments testing the behavior of the *Parrot* system, that protects the CAN bus network of modern vehicles from spoofing attacks. Using a short carefully constructed flood of messages, the *Parrot* system drives the masquerading attacker's ECU to a *Bus-off* state. Previous work found *Parrot* to be very successful. We evaluated the system's performance and stability under various extreme conditions. We first observed that despite its general success, in some scenarios the defense system exhibited degraded performance. We discovered that the degradation is related to the low-level CAN encoding of message bit patterns, and specifically to the position of the so-called 'stuff-bit' that breaks up sequences of more than 5 bits of the same value. Through detailed testing we demonstrate that the observed degradation is in fact tied to a specific, yet common, CAN controller, that seems to diverge from the CAN specification under some stress conditions. In all cases in which the problematic controller was not in use by the attacker, the *Parrot* system successfully brought the attacker into a permanent *Bus-off* state. For environments in which the mentioned controller may be taken over by the attacker, we recommend *Parrot* configurations which can achieve the same desirable results.

1 Introduction

1.1 Motivation

Modern vehicles have multiple dedicated computers called electronic control units (ECUs), which are typically connected to each other over a Controller Area Network (CAN) bus. The CAN bus protocol was designed to be robust, but not to withstand cyber-attacks. The growing demand for new functionality that requires wireless communication, has opened vehicles to new attacks, by introducing a potential new connection between the vehicle's internal CAN bus and the outside world (through WiFi, Bluetooth, Cellular, etc.).

Over the last few years researchers have shown that these attacks are both feasible and severe, e.g., by taking control over a running Jeep through its 3G connection, before driving it into a ditch. The attackers first took over the exposed Infotainment system, and used it to inject spoofed messages into the internal unprotected CAN bus, taking control of other non-compromised ECUs.

This and other similar attacks show that extra protection is required, not only to detect, but also to protect the vehicle's CAN bus from such lateral movement, under the assumption that compromising an exposed ECU is a realistic scenario.

The recently introduced *Parrot* system provides such protection. and does not require any additional hardware, nor special network topology or additional key management. Hence it can to be added as a software-only patch to any existing ECU. In this paper we try to challenge and test the boundaries of this system, and answer some of the open questions that were raised along with its introduction.

1.2 Related Work

The research into CAN bus security has grown recently, due primarily to several demonstrations of the insecurity of existing in-car networks. Koscher et al. [13] were first to implement practical attacks on cars. Using CAN bus network sniffing, fuzzing and reverse engineering of ECU's code, they succeeded to control a wide range of the automotive functions, such as disabling the brakes, stopping the engine, and so on. Later Checkoway et al. [5] showed that a car can be exploited remotely, without prior physical access, via a broad range of attack vectors, such as Bluetooth, cellular radio and even TPMS (Tire Pressure Monitoring System). Valasek and Miller [18] demonstrated actual attacks on Ford Escape and Toyota Prius cars via the CAN bus network. They affected the speedometer, navigation system, steer-

ing, braking and more. In 2015 it was reported [12, 11] that they remotely disabled a Jeep's brakes during driving, and caused Chrysler to recall 1.4M vehicles. Foster and Koscher [9] have also reported of the potential vulnerabilities of the relatively new commercial OBD-II dongles (as used by insurance companies to track one's driving) which support cellular communication, which may be even exploited via SMS. A more recent hack was published by Keen Security Lab [20] on a Tesla electrical vehicle, in which the researchers took over control of the vehicle through a bug in the infotainment's browser, forcing the company to release an over-the-air software update.

One suggested approach to secure the CAN bus was to add some authentication to the messages on the bus by using a cryptographic Message Authentication Code (MAC). Several ideas were suggested, ranging from adding a part of a MAC tag to the actual message's data field, to splitting the MAC into several pieces and layers as offered by Glas and Lewis [10]. Another idea as suggested by Van Herrewege at al. [30] was to use a new light-weight protocol to better fit the CAN bus limitations. Their *CANAuth* protocol, also relied on the *CAN+* protocol of Ziermann et al. [31], which allowed them to split the authentication bits in between the sampling points of the bus. These solutions however require having a pre-shared key, which has its own key management challenges.

A similar approach was adopted by the AUTOSAR standard as defined by the Secure Onboard Communication (SecOC) mechanism [3], to add some authentication and replay prevention to the vehicle's internal networks.

A different approach to try and destroy non-legitimate spoofed messages, by transmitting an *active-error* flag (more on this in Section 2.1), was suggested by Matsumoto et al. [17]. However, their solution requires transmitting the *active-error* flag in violation of the CAN specifications. Therefore, their solution requires a modified CAN controller, which usually implies modified hardware.

A centralized approach to combine the two previous ideas (using MAC for authentication and the *active-error* flags) was suggested by Kurachi et al. [14] to reduce the need to use modified hardware and share a key between all ECUs. In this approach a centralized modified ECU was used to both authenticate and destroy non legitimate messages. The later work of Kurachi et al. [15] demonstrated an actual implementation of a central gateway to include the above mechanism.

Another evolution of [17] was the work of Ujiie et al. [29] which replaced the usage of the MAC with other, non cryptographic, message analysis algorithms. They also implemented and tested their model in a real vehicle, taking into account important technical details, such as the error counters behavior, etc.

Other works take advantage of carefully selected properties of the CAN bus protocol, in order to solve security related problems: The work of Mueller and Lothspeich [19] suggested a method of shared-key establishment; Demay and Lebrun [8] built the *CANSPY* auditing tool to facilitate working with the CAN bus.

There are several companies attempting to address various aspects of attacks on in-car networks [2, 28, 1, 27] —some are still young and provide minimal details about their specific offerings. Among them, Berg et al. of Semcon [4] suggested a secure gateway concept for protecting the CAN bus network from the infotainment domain. The concept is to use three layers: a network layer, a messaging layer and a service layer. The secure gateway is based on standard IP protocols with standard encryptions, and the communication with the CAN bus network is handled using vehicle network adaptors.

Our starting point in the paper is the *Parrot* system [7] which also utilizes the *active-error* flag — however, in contrast to Matsumoto et al. [17], obeys the CAN protocol rules, and hence, can be implemented as a software-only upgrade to existing ECUs. The recent work of Cho and Shin [6] showed a similar mechanism that was used for a Denial-of-Service attack.

1.3 Contribution

This paper describes an extended set of experiments testing the behavior of the *Parrot* system, that protects the CAN bus network of modern vehicles from spoofing attacks. Using a short carefully constructed flood of messages, the *Parrot* system drives the masquerading attacker's ECU to a *Bus-off* state. The initial work [7] found *Parrot* to be very successful. This time we evaluated the system's performance and stability under various extreme conditions. We first observed that in some scenarios the defense system exhibited degraded performance. We then discovered that the degradation is related to the low-level CAN encoding of message bit patterns, and specifically to the position of the so-called 'stuff-bit' that breaks up sequences of more than 5 bits of the same value. Through detailed testing we demonstrate that the observed degradation is in fact tied to the specific, yet common, NXP SJA1000 CAN controller, that seems to diverge from the CAN specification under some stress conditions. We tested the *Parrot* defense in multiple configurations: Using different Data fields in the spoofed messages; Using zero-length defense messages; When the first bit-difference between the attack and defense messages is in the rightmost message bits; Using different hardware choices for the attacker, defender, and neutral observer; And using different transmission rates. In all cases in which the problematic SJA1000 controller

was not in use by the attacker, the *Parrot* system successfully brought the attacker into a permanent *Bus-off* state. For environments in which the SJA1000 controller may be taken over by the attacker, we recommend *Parrot* configurations which can achieve the same desirable results.

Organization: In the next section we describe some preliminaries. We provide a short description of the *Parrot* system in Section 3. Section 4 describes the experimentation plans and the lab setup we used in our experiments. Sections 5 to 8 describe our experiments and discuss our findings, and Section 9 includes our conclusions.

2 Preliminaries

2.1 CAN Bus

The Controller Area Network (CAN) bus standard (developed by Robert Bosch GmbH [26]) is probably the most common protocol for in-vehicle communication. The protocol is a serial broadcast protocol which offers a reliable communication channel for the vehicle's Electronic Control Units (ECUs). The ECUs control the car's different subsystems (such as the engine control unit, the ABS system, etc). Modern vehicles typically have a few dozen ECUs.

Apart from the host processor, a typical ECU consists of a CAN controller, to implement and enforce the protocol. The controller is generally implemented by hardware, whereas the host processor is usually a microcontroller or full-fledged CPU running custom firmware and software.

Each CAN frame is identified by a message ID which is either 11 or 29 bits long; However CAN messages do not carry an identifier of the destination: each ECU unilaterally decides which message IDs to accept and act upon. Any ECU can monitor all the traffic that goes over the bus (including while it is transmitting).

The CAN protocol is a synchronous protocol, in which time is split into bit-time slots (of $1\mu s$ in the 1Mbps mode). When two ECUs start to transmit in the same slot, an arbitration procedure takes place where the message ID defines its priority: 0 bits are considered dominant over 1 bits, hence messages with numerically smaller IDs are prioritized over messages with larger IDs. Note that the zero-dominance property is not limited just to the message ID field and is maintained at all bit positions: if at any point in time a 0 and a 1 bit are transmitted simultaneously, the 0 bit dominates and the 1 bit is overwritten. The *Parrot* defense approach relies on this property.

In order to ensure enough signed transitions to maintain synchronization, a bit stuffing is applied, where a bit of opposite value is inserted after every five consecutive bits of the same value. This bit is automatically inserted and removed by the CAN controller of the transmitting/receiving ECU.

Each ECU maintains two internal error counters: TEC to count the errors observed during a transmission of a message, and REC to count the errors observed while receiving a message. Some error scenarios increase the related counter by one while others increase it by 8. Every successfully received message reduces the REC counter by one, and every successfully transmitted one, reduces the TEC counter by one. If either one of the ECU's error counters reaches 128, the ECU goes into an *error-passive* state. The ECU returns to the normal *error-active* state, when both counters go below the 128 threshold. An ECU reaches *bus-off* whenever its TEC counter reaches 256. In *error-passive* state the ECU is not allowed to transmit an *active-error* flag. In *bus-off* state the ECU is permanently disabled and is not allowed to transmit at all—typically until a reset.

There are five different types of errors: BIT, STUFF, CRC, FORM, and ACK. A *bit-error* occurs when a transmitting ECU monitors a different bit than it transmitted. A *stuff-error* occurs when six consecutive bits of the same value are monitored. A *form-error* occurs when some fixed-form fields contain illegal bit/s. A *crc-error* occurs when the calculated CRC is not equal to the transmitted one, and an *ack-error* occurs when the transmitter doesn't get an Ack on his message.

Figure 1 describes a data frame in a *standard-frame* (11 bits ID) format, where the 4-bit DLC field describes the number of bytes (0-8) that the data-field should contain. There is also an "extended-frame" in which the message IDs are 29 bits wide.

2.2 The Adversary Model

We assume that the attacker, *Eve*, "owns" one of the more vulnerable and exposed ECUs (meaning those which hold the capability to communicate with the outside world, e.g., through some wireless protocol), allowing her access to the internal CAN bus. From the owned ECU, *E*, *Eve* wishes to move laterally and take control of vehicle functions, which she can do by impersonating messages normally sent by another ECU, *A*. Sending fake messages allegedly from *A*, will spoof the victim ECU, *V*, to take an attacker-selected action.

The following scenario, as depicted in Figure 2, may better demonstrate the above: The attacker, *Eve*, first takes over the relatively exposed Infotainment system, INFS (*ECU E*). Having access to the bus, *Eve*'s attack software (loaded into *ECU E*) impersonates the ABS unit (*ECU A*), feeding the Engine control (ENG) unit (*ECU V*) victim with misleading data, which will make it eventually stop.

SOF	ID	RT	IDE	Res	DLC	Data	CRC	CDel	Ack	ADel	EOF	IS
0	#11	0	0	0	#4	#DLC	#15	1	#1	1	1111111	111

Figure 1: A standard data frame, with an 11-bit ID and a 4-bit DLC (length) field. The most common case is of DLC=8, having 64 bits of data.

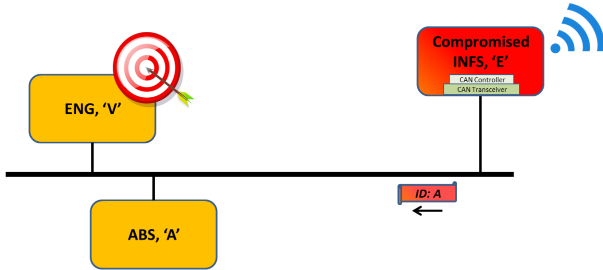

Figure 2: The adversary model. Note that both the CAN controller and transceiver are not compromised.

We assume that the attacker *Eve* has the following capabilities: She can take over an ECU, by loading malicious software into it, and change its behavior. The loaded software can transmit any desired message, at any chosen rate, masquerading as any message ID. *Eve* has full semantic understanding of the CAN bus communication, and of the contents and structure of valid messages in the system. However, crucially we assume that she cannot change the ECU's CAN controller low-level behavior which is typically implemented in hardware.

Note that these assumptions fit well with the published attacks of [5, 13, 18]: in all of them the attackers took over some ECU *E*, and injected their software into its logic, without manipulating the ECU's CAN controller hardware.

3 The Parrot Defense Mechanism

3.1 High Level Design

The *Parrot* system's [7] premise is that when *Eve* sends a spoofed message with an ID belonging to *Alice*, there is an ECU in the system that is able to recognize the spoof—and that is *Alice* herself. *Alice* doesn't require any cryptographic signature to detect the spoof, nor does she rely on any particular network topology: since the message ID belongs to *Alice*, and *Alice* did not transmit it, then certainly it is a spoof. All other ECUs, including the victim V, cannot tell the difference and will treat the spoofed message as valid. Thus, once *Alice* observes a spoof, she goes into a "parrot mode". In parrot mode, *Alice* tries to intercept all future spoofed messages, as soon as they are found on the bus, by launching a counter-attack, in order to silence the impersonating ECU. This second strike consist of a pulse of defending messages, transmitted at maximum speed by *Alice*, the owner of the spoofed message ID. The goal is that by generating collisions on the bus, between *Eve*'s spoofed messages and *Alice*'s defending ones, the attacker will drive itself into a *bus-off* state.

A key challenge is to avoid self-destruction during the counter-attack. Specifically, a bus collision between a spoofed message and a defensive one typically raises the error counters equally for *both* transmitting ECUs; Thus, in addition to driving the attacking ECU to *bus-off* state - the *Parrot* needs to ensure that the defending ECU does not end up in the same state. By careful design, relying on non-obvious low-level properties of the CAN bus standard, *Parrot* is able to consistently shut-down the attacker every time, while keeping both the defending and the surrounding ECUs operational.

3.2 Defense details

1. As soon as a *Parrot-equipped* ECU identifies a spoofed message (using one of its own IDs) which wasn't transmitted by itself, it transmits a pulse of *ND* defensive messages (*Dmessages*) at maximal speed as defined below, in order to intercept the next broadcasted spoofed message, and cause a collision.

   The size of the pulse, *ND*, is a configuration parameter of the *Parrot* system, and should be large enough to cover the expected time interval between the attacker's spoofed messages.

2. The defender continues to transmit *Dmessages* until it identifies a batch of sixteen collisions (or entering CAN *error-passive* state), which indicates a *Parrot* "collision detected" state.

3. At this point, the defender transmits 15 more *Dmessages*, in order to make sure that the attacker's CAN controller goes into *bus-off*.

The *Dmessage* should have the same ID as the spoofed ID, and the same length, i.e., the same DLC, as that of the spoofed message, with a data block of all-zeros. Alternatively, as we will see in Section 6, the *Dmessage* can

have a DLC of zero — with no payload, regardless of the length of *Eve's* spoofed messages. Further details of the system can be found in [7].

3.3 Limitations and open questions

In the introduction of *Parrot* [7] some limitations and open questions of the system were identified. In this paper we address two of these issues.

The need for speed: The *Parrot* system must transmit fast enough to almost saturate the bus for a short period of time. If the defender's CAN controller is not fast enough - [7] suggested using a helper ECU. In this paper we show that this mode-of-operation of the *Parrot's* system can be successfully used to reach the desired saturation of the CAN bus during the counter-attack, even when the defender uses a slower device such as the SJA1000.

DLC related issues: In [7] an open question remained of whether the *Parrot* system can be configured to trigger the collisions in the DLC field of the spoofed messages. In this paper we show that this is indeed a viable option, subject to some limitations.

4 Experimentation Plans

4.1 Overview

The original introduction of the *Parrot* system [7] described a set of successful experiments, which proved the validity of the system. However, despite the very promising results of [7] in which the *Parrot* system drove the attacker into *bus-off* with exceptional success, our preliminaries experiments occasionally showed some degraded results and unexpected behavior by some of the players. In these rare cases, we observed unusual error codes on the devices, and strange electrical signals on the bus. Specifically, we occasionally observed scenarios as in Figure 3, in which we see a prolonged *active-error* flag of 13 bits, instead of the common 12-bit echoed flag. Further inspection showed that in such cases, some spoofed messages were observed between the first batch of Parrot-generated collisions and the attacker's reaching *bus-off*, i.e., the defense was degraded.

The location of the interferences, and the changes in the behavior while using different data values in Eve's spoofed messages, led us to hypothesize that the unusual behavior is somehow related to the location of the stuff-bits in the encoded message. Our first set of experiments (Section 5) was designed to characterize this behavior and to explore its impact.

The second set of experiments (Section 6) was done to clarify whether it is possible to configure the *Parrot* system to trigger the collision in the DLC field, answering

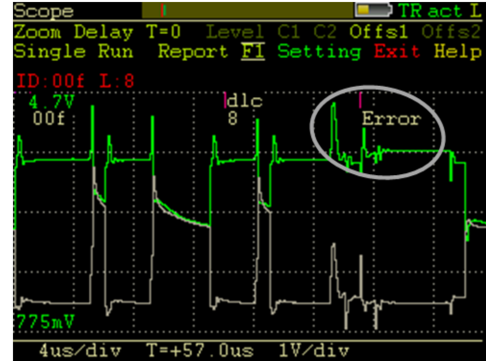

Figure 3: A screen-shot showing the CAN signals during a collision between a *DMessage* and a spoofed message with an MSByte of 0x40 (DLC=8). Notice the long burst of 12+1 dominant bits at the circled Error that seems to be a possibly malformed *active-error* echoed flag.

an open question that was raised in [7]. This set of experiments also strengthened our hypothesis on the stuff-bit effect.

The third set of experiments (Section 7) was designed to check the connection between the player's hardware and the stuff-bit effect, and characterize the problematic hardware configurations.

The last set of experiments (Section 8) was designed to test the behavior of the system in some edge cases, focusing on the message structure and the attacker's rate of transmission.

In all the experiments we counted the number of spoofed messages that pass the *Parrot's* first line of defense, i.e., those observed between the initial set of collisions (recall Section 3.2) and the attacker going into *bus-off*, in order to measure the success of the *Parrot* defense system. Note that ideally no spoofed message should pass this line during a desired operation of the system.

4.2 Lab setup

In all the experiments we used four CAN controller devices to simulate a CAN bus network of a real vehicle when under a spoofing attack. We also used one of our devices to play back real recorded traffic of an operating vehicle for additional realism. Our lab setup includes the following equipment from *Peak-system*:

- Two PCAN-USB devices [22] using the NXP SJA1000 CAN controller [24], [25].
- One PCAN-USB-FD device [23] using Peak's proprietary FPGA-based CAN controller.
- One PCAN-Diag-V2 hand tool device (HTD) [21] using the NXP LPC2292 built-in CAN controller.

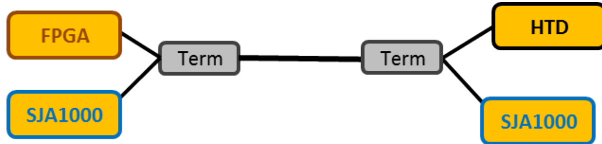

Figure 4: The general system diagram. *HTD* is the hand tool device; *FPGA* is the FPGA USB device; *SJA1000* are the standard USB devices

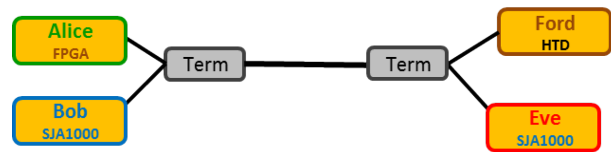

Figure 5: Setup of experiments 1 to 4

The three PCAN-USB devices are controlled via USB connections by a PC running Windows 8.1 using either the PCAN-View control software, or the PCAN-Basic software package's libraries and DLL (PCANBasic.dll).

The PCAN-View software provides a graphical interface (GUI) that can program the devices to transmit and receive CAN messages at predefined rates. The PCAN-Basic software package's libraries were used when more flexibility was required (such as for raising the controller's transmission rate, or for simulating *Eve's* role).

The hand tool device (HTD) was used to either play a specific role, or to play back a trace of CAN messages recorded on a Ford-Focus 2012 vehicle, [16], to make our environment more realistic. The built-in scope of the *HTD* was also used to capture some of the electrical signals from the bus, for detailed diagnosis.

All four CAN devices were connected, by their D9 connector, to a single terminated CAN cable (see Figure 4) to simulate the bus. For simplicity, we used a fixed 1Mbps bit rate in all of our experiments.

Each entity took a different role as required by the related experiment: The compromised ECU *Eve*, the defending ECU *Alice*, the victim ECU *Bob*, and when needed - the assisting ECU *Chester*. In most experiments, when not playing another role, the *HTD* was used to transmit the background messages. When applicable, non participating entities were simply used as observers to the system.

The results of our experiments were gathered from both the PCAN-View trace functions (GUI and files), the relevant dll based programs' interfaces, and the *HTD*'s scope.

5 The effect of the stuff-bit location

As mentioned in Section 4.1, the occasional unusual behavior of the system and its relation to the changing value of Eve's data field led us to hypothesize that the location of the stuff-bits in the *DMessages* affect the system. The following set of experiments was designed to characterize this behavior and to explore its impact.

The first experiment (Section 5.1), was designed to understand the relationship between the colliding bits position and the stuff-bit location. We did this by using a DLC of 8 (binary: 1000), and varying the most significant byte (MSByte) in *Eve's* spoofed messages. The second experiment (Section 5.2), helped us prove that the effect is indeed directly connected to the location of the stuff-bit; This was done by showing a one bit shift to the right in the effect, in a direct correlation to the change in the stuff-bit location. The shift was accomplished by using a DLC of 4 (binary: 0100) instead of 8 (binary: 1000), thus making the controller insert the stuff-bit one bit later.

In these experiments, we had *Alice* use the faster *FPGA* device with the Python implementation of the *Parrot*, in order to reach the desired maximal traffic density and complete the defensive mission on her own. In addition, we had the *HTD* transmit the recorded traffic file (see Figure 5). We used the following configuration for both sets of experiments:

- Eve (SJA1000): transmits attack messages with ID 00F, every 1 msec, having Data with a varying MSBtye (with a single non-zero bit, from 0x80 to 0x01), followed by 7 bytes of 0x00 (or 3 0x00 bytes in experiment 2).
- Alice (FPGA): transmits *Dmessages* with ID 00F, at the maximum allowed rate (about 8 messages per 1msec), having Data of 8 0x00 bytes (or 4 bytes in experiment 2).
- Bob (SJA1000): passive-reactive (not transmitting messages, but reacting to the error conditions).
- Background (HTD): transmitting the Ford trace file (2-5 messages per 1msec).

5.1 Experiment 1: using DLC=8

In this experiment we used a DLC of 8 (binary: 1000) for both *Alice* and *Eve*. Thus, the automatically inserted stuff-bit is expected after the second most significant bit (*msbit*) of the data field in *Alice's* all-zero *DMessage* avoiding five consecutive bits of zero (recall section 2.1). Another stuff-bit is expected after the 7th *msbit* of the same message. Figure 6 (Left) shows the results of the

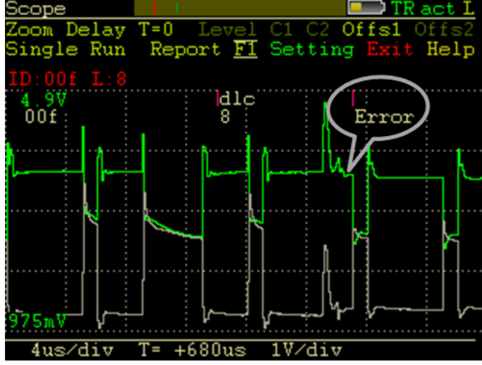

Figure 7: The unexpected interruption to the 17th message. Notice the extra dominant bit that is identified by the observing *HTD* as a *stuff-error* at the encircled *Error*, although the transmission continues as if the transmitter is unaware to the extra bit.

experiment. Notice how when the MSByte of *Eve's* message is either 0x40 or 0x02, the system behavior is degraded and several attack messages slip through the defense before the attacker reaches *bus-off*. With other MSByte values in the attack messages the results are as expected, and no spoofed messages pass the *Parrot's* defense. Closer examination revealed several unexpected phenomena:

- The length of the *active-error* flag in each of the initial 16 collisions, as observed by the *HTD*, was 13 instead of 12 bits long (as in Figure 3).
- The error that *Bob* observed raised his REC by +9 instead of by the expected +1.
- *Alice's* 17th *DMessage*, which should pass with no interference (since *Eve* should already be in *error-passive* state), seems to be interrupted, apparently by *Eve's* spoofed message (Figure 7) which is transmitted until the end, as if no error nor collision occurs.

5.2 Experiment 2: using DLC=4

In this experiment we used the same setup as in Experiment 1, except that we used messages with a DLC of 4 (binary: 0100), for both *Alice* and *Eve*, in order to make *Alice's* controller insert the stuff-bit one bit later. Figure 6 (Right) shows the effect when using this configuration. We now see the performance degradation when *Eve's* message has an MSByte of either 0x20 or 0x01. Note the clear shift of one place to the right in accordance with the one bit shift in the DLC field.

5.3 Discussion and preliminary conclusions

Based on these two experiments we see that there is a direct connection between the location of the stuff-bit in the *DMessage*, the position of the colliding bits, and the Parrot's success. Specifically the degradation seems to occur whenever the stuff-bit in the *DMessage* immediately follows the colliding bits. We also note that this effect only degrades the effectiveness of the system, but *Alice* still wins: *Eve* reaches *bus-off*, as desired, just slightly late.

In the following experiments we still need to understand whether the stuff-bit effect is a general phenomenon or is caused by a specific CAN controller, and if so — which one.

6 DLC related experiments

The next set of experiments was done in order to check the possibility of causing the collisions in the DLC field of the messages instead of in the data field, trying to answer one of the open issues that were presented in [7]. This experiment was also motivated by Cho and Shin [6] which specifically mentioned collisions in the DLC field of the messages.

For this purpose we let Alice use *DMessages* with DLC zero (no data) to intercept Eve's non-empty (DLC > 0) spoofed messages. Doing so causes the collision to happen between the first non-zero bit of *Eve's* DLC and *Alice's* all zero DLC bits, even before the data field (which is irrelevant in these experiments).

In addition, in these experiments, we also wanted to maintain a setup similar to the previous experiment (Section 5) to test whether the stuff-bit effect holds not only in the Data but also in the DLC field. This time the shift was accomplished by having one experiment with ID = 00F (binary: ..1111), and another with ID=00E (binary: ..1110), to make the controller insert the stuff-bit one bit earlier in the DLC field, shifting the effect one place to the left.

Each set of these two experiments was repeated 10 times, for each of the 8 possible non-zero values of *Eve's* DLC (from 1000 to 0001), where *Alice's* DLC was fixed to zero (0000). This experiment had the same set up (device-wise) as that of Section 5 (recall Figure 5).

6.1 Experiment 3: Varying DLC, ID=00F

Since we use an ID of 00F (1111) followed by 3 fixed zero bits (recall Figure 1), the automatically inserted stuff-bit is expected after the second *msbit* of the *DMessage's* DLC of zero.

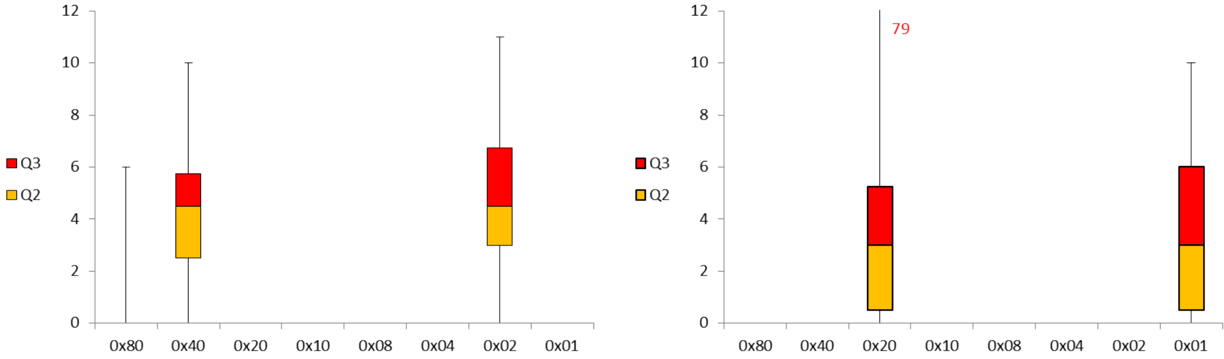

Figure 6: (Left) Experiment 1, DLC=8. (Right) Experiment 2, DLC=4. The number of observed spoofed messages to pass *Alice's* defense as a function of *Eve's* MSByte. The Median is indicated by the border between the bottom and the top boxes that indicate the second and third quartiles; The whiskers show the minimum and maximum values.

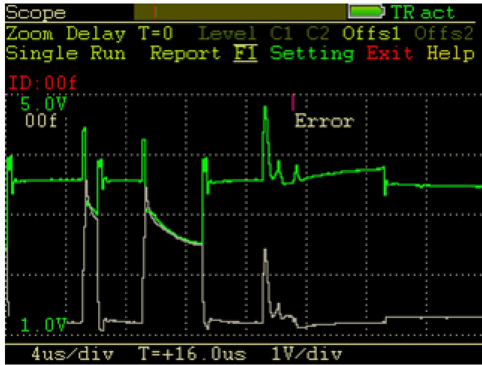

Figure 9: A collision between a DLC zero and a DLC 7 messages. Notice the prolonged sequence of 65 dominant bits of possibly the malformed *active-error* echoed flag.

Figure 8 (Left) shows the results of the *Parrot's* system when using this configuration for all 8 possible DLC values of the spoofed messages. We can see that when Eve's DLC=8 the Parrot defense works well. However, for DLCs of 7,6,5, and 4, whose leftmost colliding bit is just before the expected *DMessage's* stuff-bit, we see the same effect where several spoofed messages pass the defense.

Worse, when using a DLC of 7, the stuff-bit effect seems to be fatal to the system's operation: not only is the performance degraded, but actually the defender, *Alice*, reaches bus-off, letting *Eve* prevail.

Upon further investigation of the worst case (DLC=7) we saw in the *HTD's* scope a highly unusual prolonged *active-error* flag (Figure 9) of 65 dominant bits, instead of the normal 12 bits, which may be the reason behind *Alice's* fatal loss.

6.2 Experiment 4: Varying DLC, ID=00E

In this experiment we used the same setup as in Experiment 3, except that this time we used messages with an ID of 00E (..1110), for both *Alice* and *Eve*, in order to make the controller insert the stuff-bit one bit earlier in the *DMessage*. Figure 8 (Right) shows the results when using this configuration. We again see the performance degradation when using a DLC of 8 and the stuff-bit is expected just after the colliding bit. Note that this time the effect is evident only for a single value, since the DLC cannot exceed 8.

6.3 Discussion

The results from experiments 3 and 4 confirm our conclusion regarding the stuff-bit effect, and show that the same effect occurs in the DLC field as well. These experiments also show the the *Parrot* system can be successfully configured to triger the collisions in the DLC field. However, in special cases, when the collision occurs in the DLC field, the stuff-bit effect is amplified to a fatal reaction, so special care should be taken when using this configuration.

7 Switching the hardware

After characterizing the stuff-bit effect in the previous two sections, our next goal is to better understand the reason for this phenomenon and find the responsible entity: *Alice* or *Eve*. This was accomplished by switching the hardware between these two players.

The first set of experiments (Section 7.1) was designed to check whether switching between *Alice's* FPGA device and *Eve's* SJA1000 affects the results. This experiment required the *Parrot's* second mode-of-operation (with an assisting-neighbor), as defined in [7], since the

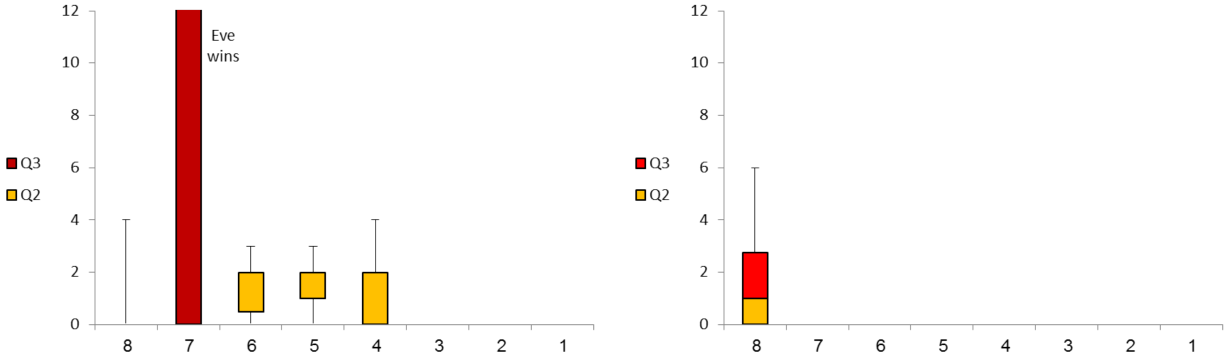

Figure 8: The number of observed spoofed messages to pass *Alice's* defense as a function of *Eve's* DLC value, when *Alice* is using empty *DMessages* (DLC=0). (Left) Experiment 3, ID 00F. Note the amplified effect when DLC=7 and *Alice* loses. (Right) Experiment 4, ID 00E. Note the shift of the effect.

slower SJA1000 device is not able to generate traffic fast enough to drive *Eve* into *bus-off*.

The second set of experiments (Section 7.2) helped us prove that the effect only occurs when the SJA1000 device serves as the attacker, *Eve*, eliminating the theoretical option that the FPGA is responsible if used by *Alice*. To check this we let *Alice* use the FPGA (as in Section 5.1), but had *Eve* use the *HTD* instead of the SJA1000 device.

7.1 Experiment 5: Varying DLC, Switched Hardware

This experiment is based on the same set-up as of experiment 3 (Section 6.1) except that this time we switched between the hardware of *Alice* and *Eve*. Since now the defender, *Alice*, used the slower SJA1000 device, we had to use an assistant (called *Chester* in [7]) to raise the density of the traffic after the initial interception of Eve's spoofed message. To achieve this we had *Chester* use the 2nd SJA1000 device to transmit some arbitrary background messages (3 per 1 msec) to reach, together with the *HTD* playing back the *Ford* file, the desired density of 5 messages per 1 msec. Specifically:

- Eve (FPGA): transmits attack messages with ID 00F, every 1 msec, with a changing DLC of 8 to 1 per set, and Data of all 0xFF bytes.
- Alice (SJA1000): transmits *Dmessages* with ID 00F, at the maximum allowed rate (about 7 messages per 1msec), with DLC=0 (no Data).
- Chester (SJA1000): Transmits three arbitrary[1] assisting messages (IDs 011/022/033), every 1 msec (DLC = 8, Data all 0x00).
- Background (HTD): transmitting Ford's trace file (2-5 msg per 1msec).

[1]These messages do not collide with the others since they have a different ID.

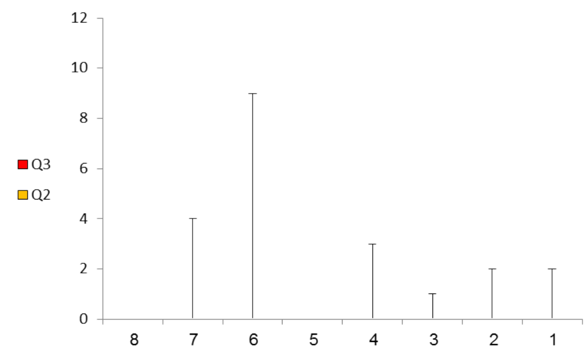

Figure 10: Experiment 5. The number of observed spoofed messages to pass *Alice's* defense as a function of *Eve's* DLC value, when *Alice* is using empty *DMessages* (DLC=0). The graph is a box-and-whiskers graph, except quartiles 2 and 3 are always zero — we only see the maximum whiskers. Note that unlike in Experiment 3 there is no evidence of the stuff-bit effect when the DLC is between 7 and 4.

The results from this experiment (Figure 10) show no sign of the stuff-bit effect: with a DLC of 7 to 4 (compare with experiment 3 and Figure 8 (Left)). Note that apart from some spurious outliers there is no evidence of degradation of the system's performance, nor any relation to the stuff-bit location in *Alice's DMessages*.

Also, this time, there was no indication of any odd increment to *Chester's* error-counter (recall the second observation in Section 5.1) throughout the entire experiment. This fits the original results from [7] and indicates that the stuff-bit effect is indeed related to the hardware in use.

7.2 Experiment 6: Eliminate the SJA1000

This experiment is based on the same set-up as of experiment 1 (Section 5.1) except that this time we let *Eve* use the *HTD* device (with NXP's LPC2292 built-in CAN controller) to transmit its spoofed messages every 1 msec while *Alice* uses the FPGA, and the observer *Bob* uses the SJA1000. The second SJA1000 device was also used for observation. Note that unlike previous experiments we couldn't transmit the Ford file in the background. Specifically:

- Eve (HTD): transmits attack messages with ID 00F, every 1 msec, having Data with a varying first byte of 0x80 to 0x00, followed by 7 bytes of 0x00.
- Alice (FPGA): transmits *Dmessages* with ID 00F, at the maximum allowed rate (about 8 messages per 1msec), having Data of 8 0x00 bytes.
- Bob (SJA1000): passive-reactive (not transmitting messages, but reacting to the error conditions).
- Bob2 (SJA1000): same as Bob.

As in experiment 5 (Section 7.1) we saw no indication of the stuff-bit effect. Moreover, this time the results were even closer to those of [7] — not a single spoofed message passed the first defense line of *Alice*. We omit the graph because it is empty. We note that this experiment also eliminates the theoretical possibility that the stuff-bit effect is related to the FPGA device when it is used by the defender, *Alice*, as it was in Experiments 1 to 4.

7.3 Discussion

The combined results from the above two experiments lead us to conclude that the stuff-bit effect is directly related to the attacker's hardware, and that it appears to exist when the attacker uses the NXP's SJA1000 controller. The other controllers we tested: Peak's FPGA, and NXP's LPC2292 built-in controller of the *HTD*, don't seem to exhibit this behavior.

8 Edge cases experiments

This set of experiments was designed to test the behavior of the system in some edge cases, focusing on the attacker's message structure and rate of transmission. The first experiment (Section 8.1) was designed to check the behavior of the system when the collision is in the least significant byte (LSByte) of the Data field. This is of interest since it moves the identification of the collision into the CRC field for both *Alice* and *Bob* (recall Figure 1).

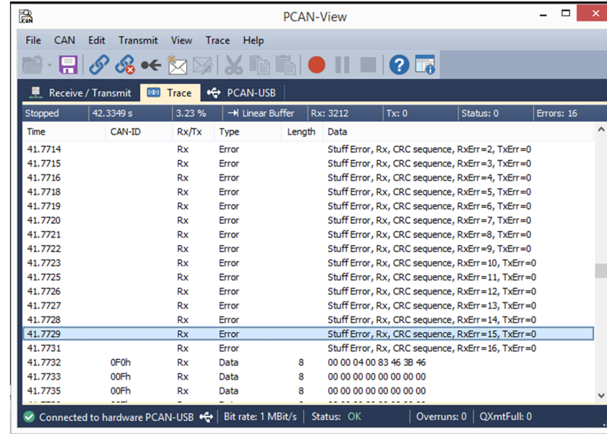

Figure 11: Experiment 7, *Bob's* GUI (using a spoof message with an LSByte of 0x02). Note that all of the reported stuff-bit errors are in the CRC field, as expected.

The second set of experiments was designed to check the ability of the *Parrot* system to handle more aggressive attackers who transmit their attack at their maximum possible rate.

8.1 Experiment 7: The CRC case

This experiment is based on the same set-up as of Experiment 1 (Section 5.1) with DLC=8 except that this time we had *Eve* use a varying LSByte Data field with leading zeros, triggering the collisions to happen on the border between the Data and the CRC fields. Note that *Alice* uses the same all-zero *DMessages* as always.

We found that when the collision occurs after the last stuff-bit of *Alice's DMessage* (after the 6th bit in the LSByte), as is the case when the spoofed message has an LSByte of either 0x02 or 0x01, *Eve's error-active* flag is indeed recognized in the CRC field of both *Alice* and *Bob*, making them raise the corresponding error (*Stuff-error* for *Bob* and *Bit-error* for *Alice*) in the CRC field instead (see Figure 11 for a screen-shot from *Bob's* terminal): in other words, the sequence of errors that the collisions trigger for both *Alice* and *Bob* is slightly different from what we observed in other experiments, but the outcome is the same: *Eve* goes into *bus-off*.

8.2 Maximum speed and Time measurements

In this set of experiments we checked the *Parrot* system's behavior when confronting an aggressive attacker that transmits at its maximum possible rate, either using the problematic SJA1000 hardware (Experiment 8A) or the FPGA (Experiment 8B). We also measured how long it took the *Parrot* system to drive *Eve* into *bus-off*, and

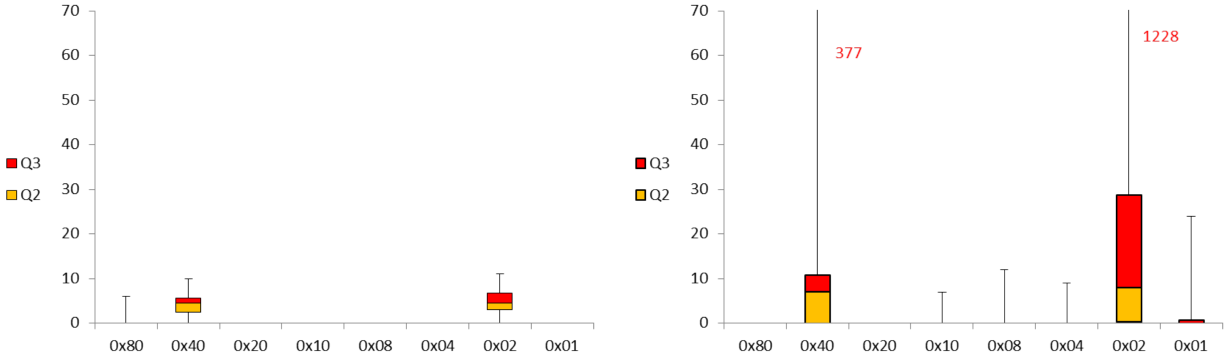

Figure 12: (Left) Experiment 1, moderate attack rate (1 per 1msec). (Right) Experiment 8A, with a maximum attack rate (7 per 1msec). The number of observed spoofed messages to pass *Alice's* defense as a function of *Eve's* MSByte. Note the amplified effect when using the higher rate.

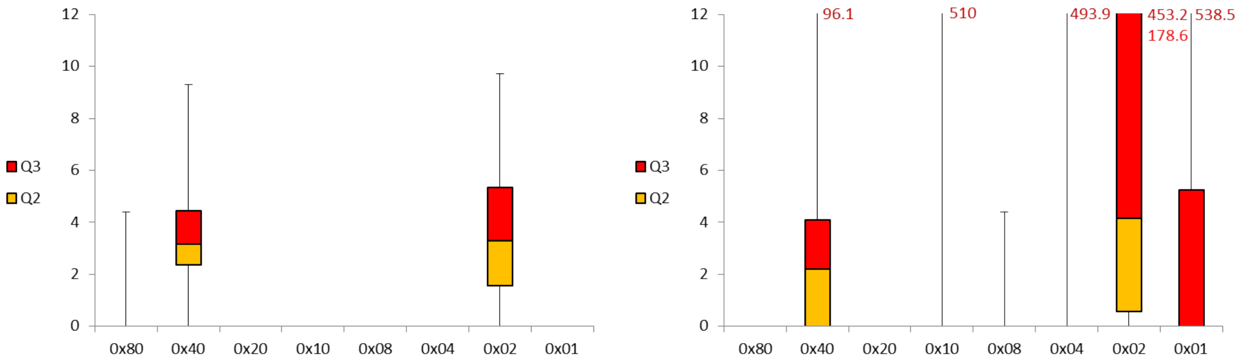

Figure 13: (Left) Experiment 1, moderate attack rate (1 per 1msec). (Right) Experiment 8A, with a maximum attack rate (7 per 1msec). The time in milliseconds between the end of the initial-set-of-16-collisions to *Eve's* last observed spoofed message (zero when none exist), as a function of *Eve's* MSByte. Note the amplified effect [and the additional maximum exceptions] when using the higher rate.

compared that to the results from the "less-agressive" experiments, 1 and 5. To estimate this time, we measured the time from the end of the initial set of 16 collisions (recall Section 3.2) until *Eve's* last observed spoofed message - indicating her entering *bus-off*. Note that we considered the time to be zero if no spoofed messages were seen after the initial-batch-of-16-collisions.

8.2.1 Experiment 8A: maximum speed using the SJA1000

This experiment is based on the same set-up as of Experiment 1 (Section 5.1) except that this time we had *Eve* transmit her attack at her maximum possible rate (of about 7 messages per 1msec on the SJA1000) using a similar dll-based program as that of *Alice's*. The following phenomena were observed:

- The degrading stuff-bit effect was amplified by a factor of about 10
- About ten percent of the experiments failed[2], probably due to the extreme stress on the system.

Figure 12 shows the results from this experiment (Right) in comparison to those of Experiment 1 (Left), which is a re-scaled copy of Figure 12 (Left). Figure 13 shows the measurements of the elapsed time, from the end of the initial set-of-16-collisions to *Eve's* last observed message, in comparison to similar measurements from Experiment 1: Note that the amplification is even bigger when considering the elapsed time, although the Medians of the results seem to stay similar.

[2]We counted a failure when one of the devices' programs got stuck, or collapsed, or that *Alice* failed to identify the spoofed messages.

8.2.2 Experiment 8B: maximum speed using the FPGA

This experiment is based on the same set-up as of Experiment 5 (Section 7.1) except that this time we had *Eve* transmit her attack at her maximum possible rate (of about 8 messages per 1msec using the FPGA). Note that this time *Eve's* rate was even faster than *Alice's* since she used the FPGA device. As in Experiment 5, we again found no sign of the stuff-bit effect, thus the only observed phenomenon was that of the failed execution (about 10 percent, as in Experiment 8A). However, the results in this experiment were even better than those of Experiment 5, making us omit its graph.

9 Conclusions and Future Work

In this paper we saw an extended set of experiments testing the behavior of the *Parrot* system, that protects the CAN bus network of modern vehicles from spoofing attacks. We evaluated the system's performance and stability under various extreme conditions. We observed a performance degradation that is related to the low-level CAN encoding of message bit patterns, and specifically to the position of the 'stuff-bit'. Through detailed testing we demonstrated that the observed degradation is tied to the NXP SJA1000 CAN controller, that seems to diverge from the CAN specification under some stress conditions. We tested the *Parrot* defense in multiple configurations: Using different Data fields in the spoofed messages; Using zero-length defense messages; When the first bit-difference between the attack and defense messages is in the rightmost message bits; Using different hardware choices for the attacker, defender, and neutral observer; And using different transmission rates. In all cases in which the problematic NXP controller is not in use by the attacker, the *Parrot* system successfully brought the attacker into a permanent *Bus-off* state.

For environments in which the SJA1000 controller is at high risk of being taken over by an attacker (such as when it is used by a relatively exposed ECU like the Infotainment), we recommend to configure *Parrot* to match the DLC value of its *DMessages* to that of the attacker's spoofed messages, and to avoid the potentially problematic DLC field. We note that an adaptive attacker may possibly bypass the defense in this mode, and that further investigation may be needed. For all other ECUs that show no similar behavior to that of the SJA1000, we recommend to configure the system to use empty *DMessages* to narrow down the attacker's possibilities and maintain the defender's tactical advantage.

A remaining challenge to the system is the possibility of an attacker to mount an adapted version of the *Parrot* system against a, possibly *Parrot*-protected, ECU of his choice in order to silence a genuine non-compromised ECU. This scenario can lead to a *parrots* fight, where both the defending and the attacking entities will try to silence each other using the same mechanism. Further investigation is needed to address this issue.

The bottom line from all our experimentation thus far is that the *Parrot* system seems to be a viable and practical software-only system that may be able to defend CAN bus networks from spoofing attacks in a wide range of scenarios, without need for any special hardware or topology. Further investigation is still needed in order to check the possibility to cover more complex scenarios.

References

[1] Argus Cyber Security Ltd. `http://argus-sec.com`, 2015. [Online; accessed 22-July-2015].

[2] Arilou. `http://ariloutech.com`, 2015. [Online; accessed 22-July-2015].

[3] AUTOSAR. AUTOSAR secure onboard communication (SecOC), version 4.3. `https://www.autosar.org/standards/classic-platform`, 2016.

[4] J. Berg, J. Pommer, C. Jin, F. Malmin, and J. Kristensson. Secure gateway - a concept for an in-vehicle IP network bridging the infotainment and the safety critical domains. In *13th Embedded Security in Cars (ESCAR'15)*, 2015.

[5] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[6] K.-T. Cho and K. G. Shin. Error handling of in-vehicle networks makes them vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[7] T. Dagan and A. Wool. Parrot, a software-only anti-spoofing defense system for the CAN bus. In *14th Int. Conf. on Embedded Security in Cars (ESCAR 2016)*, Munich, Germany, Nov. 2016.

[8] J. C. Demay and A. Lebrun. CANSPY: A platform for auditing CAN devices. In *Blackhat US 2016*, 2016.

[9] I. Foster and K. Koscher. Exploring controller area networks. *USENIX ;Login: magazine*, 40(6), 2015.

[10] B. Glas and M. Lewis. Approaches to economic secure automotive sensor communication in constrained environments. In *11th Int. Conf. on Embedded Security in Cars (ESCAR 2013)*, 2013.

[11] A. Greenberg. After Jeep hack, Chrysler recalls 1.4m vehicles for bug fix. http://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/, 2015.

[12] A. Greenberg. Hackers remotely kill a Jeep on the highwaywith me in it. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, 2015.

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

[14] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata. CaCAN—centralized authentication system in CAN (controller area network). In *12th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[15] R. Kurachi, H. Takada, T. Mizutani, H. Ueda, and S. Horihata. SecGW secure gateway for in-vehicle networks. In *13th Int. Conf. on Embedded Security in Cars (ESCAR 2015)*, 2015.

[16] M. Markovitz and A. Wool. Field classification, modeling and anomaly detection in unknown CAN bus networks. In *13th Embedded Security in Cars (ESCAR'15)*, Cologne, Germany, Nov. 2015.

[17] T. Matsumoto, M. Hata, M. Tanabe, K. Yoshioka, and K. Oishi. A method of preventing unauthorized data transmission in controller area network. In *IEEE Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2012.

[18] D. C. Miller and C. Valasek. Adventures in automotive networks and control units. http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf, 2014. [Online; accessed 22-July-2015].

[19] A. Mueller and T. Lothspeich. Plug-and-secure communication for CAN. *CAN Newsletter*, pages 10–14, 2015.

[20] D. Pauli. Hackers hijack Tesla Model S from afar, while the cars are moving. http://theregister.co.uk/2016/09/20/tesla_model_s_hijacked_remotely, 2016.

[21] PEAK-System. PCAN-Diag 2: Handheld device for CAN bus diagnostics. http://www.peak-system.com/produktcd/Pdf/English/PCAN-Diag2_UserMan_eng.pdf, 2015.

[22] PEAK-System. PCAN-USB: CAN interface for USB. http://www.peak-system.com/produktcd/Pdf/English/PCAN-USB_UserMan_eng.pdf, 2015.

[23] PEAK-System. PCAN-USB FD: CAN FD interface for high-speed USB 2.0. http://www.peak-system.com/produktcd/Pdf/English/PCAN-USB-FD_UserMan_eng.pdf, 2015.

[24] Philips Semiconductors. SJA1000 stand-alone CAN controller. Application Note AN97076, http://www.nxp.com/documents/application_note/AN97076.pdf, 1997.

[25] Philips Semiconductors. SJA1000, stand-alone CAN controller. Data Sheet, http://www.nxp.com/documents/data_sheet/SJA1000.pdf, 2000.

[26] Robert Bosch GmbH. CAN specification, version 2.0. http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf, 1991.

[27] Security inMotion. http://www.security-inmotion.com, 2015. [Online; accessed 22-July-2015].

[28] TowerSec. http://tower-sec.com, 2015. [Online; accessed 22-July-2015].

[29] Y. Ujiie, T. Kishikawa, T. Haga, H. Matsushima, T. Wakabayashi, M. Tanabe, Y. Kitamura, and J. Anzai. A method for disabling malicious CAN messages by using a centralized monitoring and interceptor ECU. In *13th Int. Conf. on Embedded Security in Cars (ESCAR 2015)*, 2015.

[30] A. Van Herrewege, D. Singelee, and I. Verbauwhede. CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[31] T. Ziermann, S. Wildermann, and J. Teich. Can+: A new backward-compatible controller area network (CAN) protocol with up to 16$\times$ higher data rates. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 1088–1093. IEEE, 2009.