Parrot, a software-only anti-spoofing defense system for the CAN bus

Tsvika Dagan and Avishai Wool

tdagan02@gmail.com, yash@eng.tau.ac.il
Tel Aviv University, Israel

***Abstract*—This paper describes a novel anti-spoofing system for in-car CAN bus networks. If an attacker compromises one of the car's electronic control units (ECUs), and from there tries to attack another, more critical, ECU, the *Parrot* system blocks this lateral movement. Unlike previous firewall-based solutions or cryptography-based solutions, the attack messages are identified and destroyed by the legitimate message ID's owner. Our method does not merely drop messages that are non-conforming with policy: the *Parrot* defense typically disconnects the compromised ECU from the bus. And unlike previous solutions, that require a modified controller (since they violate the CAN bus protocol), our method is able to shut down the attacker *while obeying* the protocol rules. Hence, the *Parrot* defense can be added as a software-only patch to any standard ECU. We implemented the *Parrot* system and tested its behavior in detailed experiments. With CAN controllers that are able to transmit fast enough we were able to disable the attacking ECU in 100% of experiments. For slower controllers, we showed a successful alternative.**

I. Introduction

A. Motivation

Modern cars have multiple dedicated computers under the hood called "electronic control units" (ECUs). These ECUs control all aspects of the car's operation: from the engine, breaking and steering controls to the car's entertainment systems. The ECUs are connected to each other in a network that typically uses the CAN bus protocol. CAN bus is a simple serial protocol, with absolutely no security components: it was designed under the assumption that all ECUs are legitimate, trustworthy, and operating according to their specifications. However, over the last few years researchers have shown that many ECUs are vulnerable to attack. Since CAN bus in itself is so naive, any attack on one ECU can immediately allow lateral movement, to attack other, more critical, ECUs; the subverted ECU can trivially spoof (masquerade as) any other ECU and cause significant damage. Replacing CAN bus with a more robust technology is probably a good idea. However, due to the huge investment made by manufacturers, and the decades it takes until old cars are scrapped, it is an important goal to improve the security stance of cars within the limitations of CAN bus. Thus finding methods to block the lateral movement, from the originally compromised ECU to others, is a major step in this direction.

B. 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. [10] 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. [4] 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). As a result, Checkoway et al. pointed to the need of using security practices in cars for restricting access and improving code robustness, similar to those in general purpose IT networks. Valasek and Miller [15] demonstrated actual attacks on Ford Escape and Toyota Prius cars via the CAN bus network. They affected the speedometer, navigation system, steering, braking and more. Recently it was reported [9], [8] that they remotely disabled a Jeep's brakes during driving, and caused Chrysler to recall 1.4M vehicles. Foster and Koscher [6] 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.

One suggested approach to secure the 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 [7]. Another idea as suggested by Van Herrewege at al. [26] 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. [27], 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 different approach to try and destroy non-legitimate spoofed messages, by transmitting an *active-error* flag (more on this in Section II-A), was suggested by Matsumoto et al. [14]. 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. In our work we also utilize the *active-error* flag—however, in contrast to Matsumoto et al. [14], our approach obeys the CAN protocol rules, and hence, can be implemented as a software-only upgrade to existing ECUs.

A centralized approach to combine the two previous ideas (using MAC for authentication and the *active-error* flags) was suggested by Kurachi et al. [11] 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. [12] demonstrated an actual implementation of a central gateway to include the above mechanism.

Another evolution of [14] was the work of Ujiie et al. [25] 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 [16] suggested a method of shared-key establishment; Demay and Lebrun [5] 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], [24], [1], [23] —some are still young and provide minimal details about their specific offerings. Among them, Berg et al. of Semcon [3] 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.

C. Contribution

Our starting point is the observation that an ECU (*Alice*) whose messages the attacker (*Eve*) chooses to spoof, is capable of identifying the spoof: *Alice* knows the IDs of all "her" messages; further, due to the broadcast nature of the CAN bus, Alice sees the spoofed messages. Thus she can detect one of her message IDs broadcast by another ECU. Importantly, Alice is able to identify the spoofed messages without relying on cryptography, and without requiring any particular network topology.

Our main contribution is a novel defense system we call "Parrot". Our method does not merely drop messages that are non-conforming with policy: the *Parrot* defense typically disconnects the compromised ECU from the bus. And unlike previous solutions, that require a modified controller (since they violate the CAN bus protocol), our method is able to shut down the attacker *while obeying* the protocol rules. After *Alice*, the owner of the spoofed message ID, identifies a spoofed message, her *Parrot* defense intercepts future spoofed messages, as soon as they are found on the bus. This is done by launching a counter-attack, in order to silence the impersonating ECU. The counter-attack consists of a short pulse of defending messages, transmitted at maximum speed by *Alice*. By generating carefully crafted collisions between the attacker *Eve*'s spoofed messages and *Alice*'s defending ones, we cause *Eve* to drive herself into a *bus-off* state. Our counter-attack aims to collide already with *Eve*'s second spoofed message, destroy it, and to disable the attacker.

The *Parrot* system requires no special hardware, or changes to the CAN controller, and therefore can be added as a software patch to any ECU with a standard CAN controller. This is due to the fact that the system is based on the standard CAN bus protocol as defined in [22], and relies on the required behavior of both malicious and defending entities.

A key challenge that our approach has to address is how 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—we need to ensure that the defending ECU does not end up in the same state. As we shall see, by careful design of our system, relying on non-obvious low-level properties of the CAN bus standard, we are able to consistently shut-down the attacker every time, while keeping both the defending and the surrounding ECUs operational.

We implemented the Parrot system and tested its behavior in detailed experiments. With CAN controllers that are able to transmit fast enough we were able to disable the attacking ECU (*Eve*) in 100% of experiments. For CAN controllers that are limited in their transmission rate we showed a successful alternative: if the combination of benign CAN traffic, the *Parrot* system's defense, and a helper ECU's traffic, produces a well timed pulse of high-enough bus load, we are again able to disable the attacker.

Organization: In the next section we describe some necessary details about the CAN bus protocol and our adversary model. In Section III we describe our basic counter-attack defense, and detail how and why it works. Section IV describes our experimentation with our defense system. In Section V we sketch an extended defense system that can protect ECUs with limited transmission speed, and we conclude with Section VI.

II. Preliminaries

A. CAN Bus

The Controller Area Network (CAN) bus standard (developed by Robert Bosch GmbH [22]) 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 micro-controller or full-fledged CPU running custom firmware and software.

The protocol itself includes four types of messages (frames): Data, Remote, Error, and Overload. Each Data 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.

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. Our 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.

Every ECU also monitors the bus while transmitting. 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, to limit its influence on the bus in case of a fundamental malfunction. 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 (more on this below). In *bus-off* state the ECU is permanently disabled and is not allowed to transmit at all—typically until a reset.[1]

A crucial property for our defense approach is the ECUs CAN controller's behavior when they detect an error: When identifying an error, an *error-active* ECU broadcasts an *active-error* flag, which consists of six consecutive dominant 0 bits, followed by eight recessive delimiter bits. This flag violates the bit stuffing convention to make sure that other ECUs identify the problem, raising in return, their own error flag: each ECU that identifies an *active-error* flag is obligated to transmit its own flag as an echo. An *error-passive* ECU is only allowed to transmit a *passive-error* flag, which consist of six + eight recessive bits only.

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 (with an exception for the arbitration procedure, the message acknowledgment, and the *passive-error* flag signaling). 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. The rules defining the error counter updates for each error are complex, we shall describe the relevant details as needed.

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.

[1]The CAN bus specification allows an ECU to return from *bus-off* state to *error-active* state under certain conditions, however our equipment alway required a *reset* to exit the *bus-off* state, unless specifically configured to auto-recover.

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

Fig. 1. A standard data frame, with an 11-bit ID and a 4-bit DLC (length) field. IS denotes the 3-bit *intermission* field.

The following CAN properties are relevant to our defense approach:

- Each ECU is pre-configured with a list of its own unique message IDs.
- Any ECU can monitor all the traffic that goes over the bus.
- The Intermission field: Data frames are separated from previous frames by an Interframe Space, which includes the *intermission* field, consisting of three recessive bits. No ECU is allowed to transmit a new Data frame during *intermission*, which means that the minimal required gap between every two transmitted messages is of three bit-time slots ($3\mu s$ if working at 1Mbps).
- Suspend transmission: After an *error-passive* ECU transmits a message, it has to send eight more recessive bits, after the minimal required *intermission* bits, before gaining the right to transmit a new message.

B. 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.

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 legal message, 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.

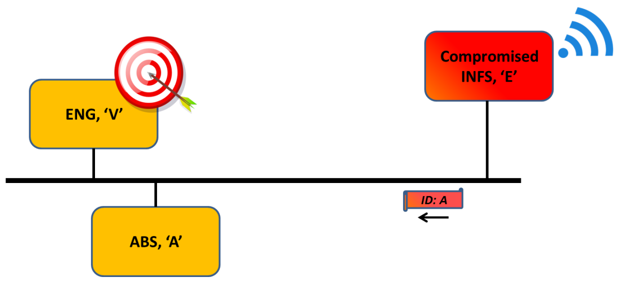


Fig. 2. The adversary model

Note that these assumptions fit well with the published attacks of [4], [10], [15]: in all of them the attackers took over some ECU *E*, and injected their software into its logic, without manipulating its CAN controller hardware.

III. The Basic Defense Mechanism

A. Overview

Our first observation 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. Our 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 that our approach has to address is how 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 - we need to ensure that the defending ECU does not end up in the same state. As we shall see, by careful design of our system, relying on non-obvious low-level properties of the CAN bus standard, we are able to consistently shut-down the attacker every time, while keeping both the defending and the surrounding ECUs operational.

Algorithm 1 The *Parrot* pseudo code

```
 1: procedure MAIN()
 2:     InitializeDefenseSystem()
 3:     while parrotOnGuard do
 4:         if suspectFound then
 5:             # identified a spoofed message with my
 6:             # ID
 7:             ENGAGE(spoofedID)
 8: procedure ENGAGE(SPOOFEDID)
 9:     # continue as long as we either intercepted
10:     # the spoofed message or give up
11:     while suspectFound and !collisionDetected do
12:         transmitNDmessages(ND)
13: procedure TRANSMITNDMESSAGES(ND)
14:     bound = ND
15:     for (i=0 ; i < bound ; i++) do:
16:         transmitDmessage()
17:         # After identifying a potential collision we
18:         # enter the final stage of our counter-attack,
19:         # and reset the flags to allow new suspect
20:         # identification.
21:         if collisionDetected then:
22:             collisionDetected=False
23:             suspectFound=False
24:             # transmit exactly 15 more Dmessages
25:             bound=i+16
```

B. 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 the defensive *Dmessages* until identifying 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 defensive *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 special data block of all-zeros. As we shall see, an all-zero data field is critical for our system's success. Algorithm 1 shows pseudo code of the *Parrot* system.

C. Why does it work

Suspect identification: We assumed that every ECU knows which Message ID "belong" to it, and since every ECU can see all the traffic that goes over the bus, a masqueraded ECU (*Alice*) can identify a spoofing attack, as soon as it sees the first message to include one of its own IDs. This causes the "suspectFound" flag to be set (line 4 in Algorithm 1).

Message interception - the first collision: After identifying the attack, *Alice* starts to transmit her pulse of *Dmessages* at maximal speed (procedure *Engage* in Algorithm 1), with the objective that the next transmitted spoofed message shall start at the same bit-time slot as one of her own *Dmessages*. Under the rules of arbitration and the synchronized broadcast nature of the protocol, sending a batch of *Dmessages* to include the same ID and DLC field as the next expected spoofed message, should result in a simultaneous transmission of the message by both *Eve* and *Alice*. Since the two messages are identical until the data field (recall Figure 1), both ECUs continue to transmit up until they transmit different bits in the data field. When this occurs, *Eve*'s CAN controller identifies a *bit-error* on the first recessive bit in her data field (since the *Dmessage* is of all-zeros). This makes *Eve* transmit an *active-error* flag (of six dominant bits), which in turn also raises a *bit-error* for *Alice* (by setting one of her recessive *stuff-bits* to zero). As a result, the collision raises the TEC counter of both ECUs by +8, and causes the neighboring ECUs to identify a *stuff-bit* error (which causes their REC counter to increase by +1), due to the violation of the stuffing convention. Figures 3 to 5 demonstrate this scenario.

The snow-ball effect: Since both *Eve* and *Alice* fail to transmit their messages, their CAN controllers try to automatically retransmit the same message as soon as possible. Since both hold the same ID and they are already synchronized, their next try starts again at the same bit-time slot, leading to a second collision of the same type. The collisions of the retransmissions continue up until after the 16'th collision, at which point both of their TEC counters reach the threshold of 128, and both are driven into an *error-passive* state. At this point, the *Parrot* system recognizes a "collisionDetected" condition (line 21 in Algorithm 1).

The 17'th silent collision: At this point, both *Alice* and *Eve* are in *error-passive* state, so both ECUs have to delay their retransmission, by an extra eight bit-times, under the *suspend-transmission* rule. Note that they remain synchronized for the next try. However, this time, when both start the retransmission, the equilibrium breaks. When the collision occurs, being *error-passive* forbids *Eve* from transmitting a regular *active-error* flag. *Eve* still raises her own TEC by 8, reaching the value of 136. Then *Eve* transmits a *passive-error* flag (of 14

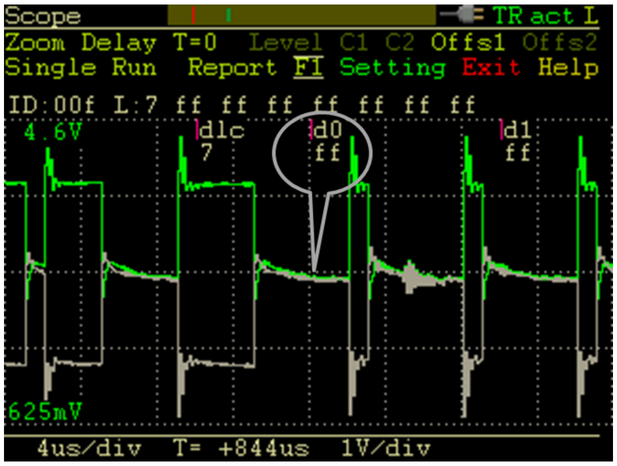


Fig. 3. The signal trace of a spoofed message, with ID:0x00F, a data length (DLC) of 7, and 7 0xFF data bytes. Notice the encircled *d0* red line, marking the beginning of the 1st data byte.

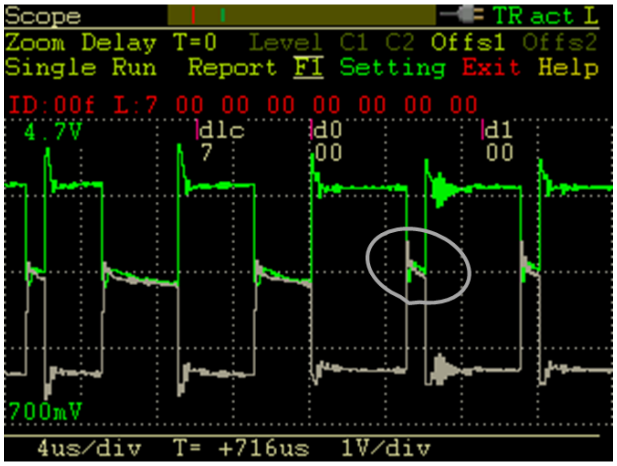


Fig. 4. The signal trace of the *Dmessage*, with the same ID as the spoofed message but with an all-zero data field. Notice the encircled recessive *stuff-bit* on the 6th bit of *d0*.

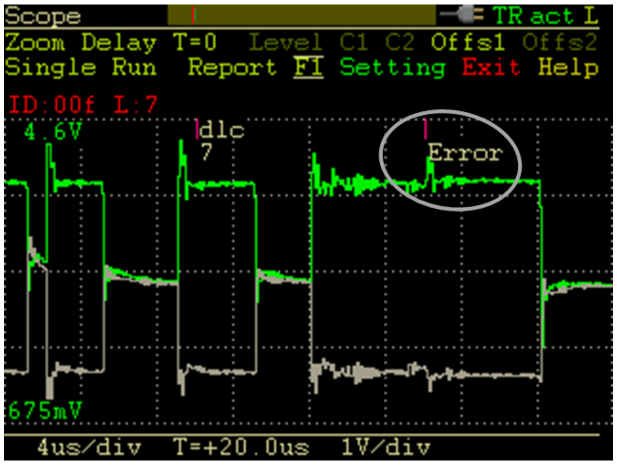


Fig. 5. A collision between the above two messages, as seen by a third passive observer. Notice that the collision begins only with the first different data bit (on *d0*), and that the dominant bits prevail. Notice also that the 12 dominant bits include the *active-error* echoed flag, identified by the observer as a *stuff-error* at the encircled red line.

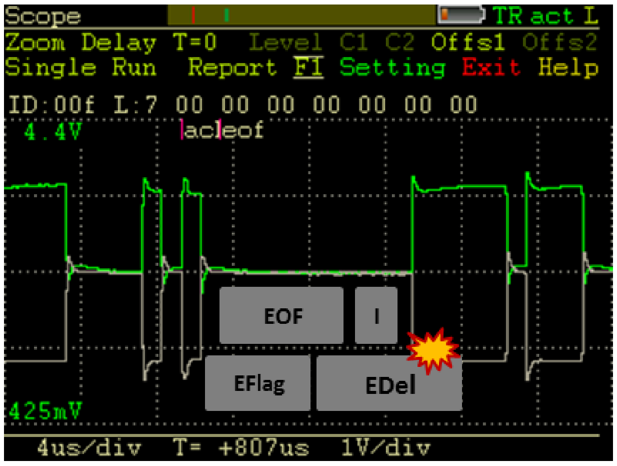


Fig. 6. The minimal gap of 11 bit-time slots, between the 17th silent collision and *Alice*'s next *Dmessage*. The grey rectangles illustrate both *Eve*'s (lower) and *Alice*'s (upper) transmitted fields. Notice the interruption in *Eve*'s error delimiter (EDel) block, as marked on the right.

recessive bits, as defined in section II-A). This doesn't disturb *Alice*'s transmission (since it consists of recessive bits only) letting her eventually finish her long-waited retransmission, lowering her TEC counter by one. Note that all 17 collisions in the "snowball" were caused by the successful interception of a single transmitted *Dmessage*.

From error-passive into bus-off: After the 17'th collision, *Eve* is waiting to complete the transmission of her *passive-error* flag, which requires having a minimal gap of six+eight recessive bits during the Interframe Space. This gap is needed in order to complete the transmission of both the 6 bits of the flag, and the 8 bits of the following delimiter.

However, since *Alice* continues to transmit at maximum speed, there is not enough idle time for *Eve* to complete her recessive sequence: *Alice*'s last retransmission of a *Dmessage* ends as just like all *data-frames* (recall Figure 1) with an Ack delimiter (1 bit), the EOF (7 bits) and the 3-bit *intermission*, totaling 11 bits. Thus *Alice*'s next *Dmessage* starts while *Eve* is still attempting to transmit her *passive-error* flag - interrupting the transmission (see figure 6). Since the interruption comes during the error delimiter transmission, *Eve* gets an internal *error-delimiter form-error*, which makes her TEC go up by 8. This scenario repeats itself for fifteen times, due to *Alice*'s next 15 *Dmessages*, causing *Eve*'s TEC counter to reach the desired *bus-off* threshold of 256 (136+8*15). On the other hand, *Alice*'s TEC counter continues to descend back toward the safe ground of zero. Furthermore, all other ECUs on the bus receive *Alice*'s successful transmissions and reduce their REC values toward zero accordingly.

D. Limitations and side effects

The need for speed: a critical piece of the *Parrot* defense is the ability of the defender, *Alice*, to cause the final 15 collisions for *Eve*'s *passive-error* flag transmission. Each of these collisions is caused by a separate message: hence, we need the bus to be almost completely saturated during the counter-attack pulse, once *Eve* goes into *error-passive* state. In Algorithm 1 we achieve this saturation by having Alice transmit *Dmessages* at maximum rate. However, we note that the same effect can be achieved if *any* messages interrupt *Eve*'s *passive-error* flag transmission.

The second-strike nature of our model: We assume that non-compromised ECUs on the bus are able to survive the first wave of attack (to absorb at least one maliciously spoofed message), before our system can react. Instead, if we were to allow modifying the CAN controller hardware, we could launch the counter-attack while the 1st spoofed message is being broadcast.

Fast response: Special care should be taken to ensure a fast enough response time between the spoof identification and the launch of the counter-attack. For highly sensitive ECUs who may not be able to withstand the first wave of attack (where a single spoofed message can cause a relatively serious damage, as maybe for the Airbag controller), we recommend deploying the *Parrot* defense as a hardware patch.

The adversary's possible recovery: The attacker's confinement to a *bus-off* state may only be temporary, since according to the standard an ECU is allowed to recuperate after some minimal idle time (of 128 occurrences of 11 consecutive recessive bits); in our experiments we did not observe such recuperations. However, even if *bus-off* is not permanent, the earned "safe-intervals" may be sufficient to maintain acceptable operation of the vehicle, as previously demonstrated by Ujiie at al. [25].

DLC related issues: Although previously gathered data from the work of Markovitz and Wool [13] show that all messages had a fixed length of 8 bytes, it is important to note that some changes may be required to the defense algorithm, in case of an adaptive adversary who may change the DLC field of the spoofed messages during his attack. If the lengths of the spoofed message and the *Dmessage* are different, we will have collisions in the DLC field, which has other effects on the error counters. We note that further investigation is required.

Neighboring ECUs: Neighboring ECUs should be configured in a way that "all-zero" messages (as the *Dmessages*) should be either ignored or treated as warning messages only. More on this in section V.

The effect on the genuine traffic: Our defense raises the bus load to maximum during the counter-attack pulse, subject to both the priority of the spoofed message, and to the predefined size of the defensive batch. Since *Dmessages* have the same priority (ID) as the original spoofed ones, the effect of both the defending and the spoofed messages on the network is equal to their relative "priority power" as defined by their message ID, and doesn't (mostly) affect other messages of higher priority.

IV. Experiments

We conducted several different experiments, in order to better understand the protocol and define our system.

A. Lab setup

Our lab setup includes the following equipment from *Peak-system*:

- Two PCAN-USB device [18] using the Phillips SJA1000 CAN controller [20], [21].
- One PCAN-USB-FD device [19] using Peak's proprietary FPGA-based CAN controller.
- One PCAN-Diag-V2 hand tool device (HTD) [17] using the NXP LPC2292 built-in CAN controller.

As a bus we used a single terminated CAN cable (see Figure 7). The 3 PCAN-USB devices were controlled via USB connections by a PC running Windows 8.1 with the *PCAN-View* control software. The *PCAN-View* software provides a graphical interface (GUI) that can program the device to transmit and receive CAN messages. In addition we used the *PCAN-Basic* software package's libraries and a DLL (*PCANBasic.dll*) to access to the devices' drivers. The hand tool device has a scope and a CAN protocol analyzer, with a graphical signed display - the screen shots in Figures 3–5 are all taken by the hand-tool device.

We used the four CAN devices connected through a CAN cable to simulate a small CAN bus. Each device was connected to both our computer (with a USB cable) and the CAN cable (using its D9 connector) of our simulated bus. 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*, and the victim ECU *Bob*. The forth entity was mainly used as a passive (Listen-Only-Mode: LOM) observer, to gather information on the related events (using the hand tool device's scope and tracing capabilities).

The results of our experiments were gathered from both the *PCAN-View* trace functions (GUI and files) and the *HTD*'s scope capability. A summary of our results is described in Table I (for 10 executions of 10 seconds each).

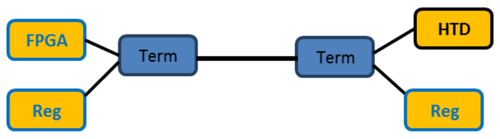


Fig. 7. The general system diagram. Notation: *HTD*: is the hand tool device; *FPGA*: is the FPGA USB device; *Reg*: are the standard SJA1000 USB devices

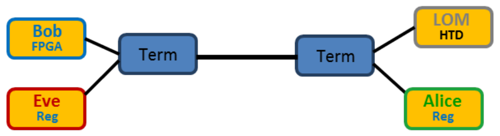


Fig. 8. Experiment 1: using a 1msec defending cycle

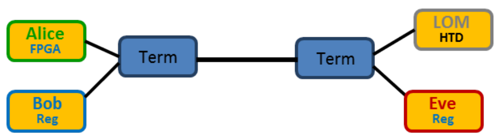


Fig. 9. Experiment 2: using the maximum possible transmission speed

TABLE I
EXPERIMENT RESULTS

Parrot CAN controller	*Dmessages* minimal gap	*Eve* msgs per second	*Eve* entrance to *bus-off*
SJA1000	$31\mu s$	1	0%
SJA1000	$31\mu s$	1000	0%
FPGA	$3\mu s$	1	100%
FPGA	$3\mu s$	1000	100%

B. Preliminary experiments

For the first set of experiments we programmed the USB CAN devices' using supplied PC GUI, which allows transmitting arbitrary messages at a maximum rate of 1000 messages per second. In this experiment both *Alice* and *Eve* used the regular USB *Reg* device, *Bob* used the *FPGA* device, and the *HTD* was in LOM, as depicted in Figure 8. We used the following parameters:

- Eve (Reg): transmits attack messages with ID 00F, every 1 sec (1000msec), with DLC=7 and Data of 7 0xFF bytes.
- Alice (Reg): transmits *Dmessages* with ID 00F, every 1msec, with DLC=7 and Data of 7 0x00 bytes.
- Bob (FPGA): passive-reactive (not transmitting messages, but reacting to the error conditions), and the *HTD* was in LOM.

With this setup we managed to get a collision between one of *Alice*'s and *Eve*'s messages, and we observed the snowball effect (batch of sixteen *bit-error* consecutive collisions) in the message's data-field, making both *Alice* and *Eve* raise their TEC counter by 8*16=128 (recall section III-C). However a transmission rate of 1000 *Dmessages* per second was too slow to produce the collision with Eve's *passive-error* flag. At this transmission rate *Alice* leaves a gap of approximately $300\mu s$ after her retransmission: more than enough for *Eve* to complete her retransmission and escape the drive to *bus-off*. We then repeated the experiment, this time programming the device via a Python program using Peak's *PCANBasic.dll*, which is not restricted to 1000 messages per second. The program implemented the behavior of the *Parrot* as defined in Algorithm 1, and allowed us to drive the Phillips SJA1000-based device to some 7000 messages per second, with a $31\mu s$ gap between messages - but this was still too slow to bring the attacking device to *bus-off*.

C. The FPGA experiment

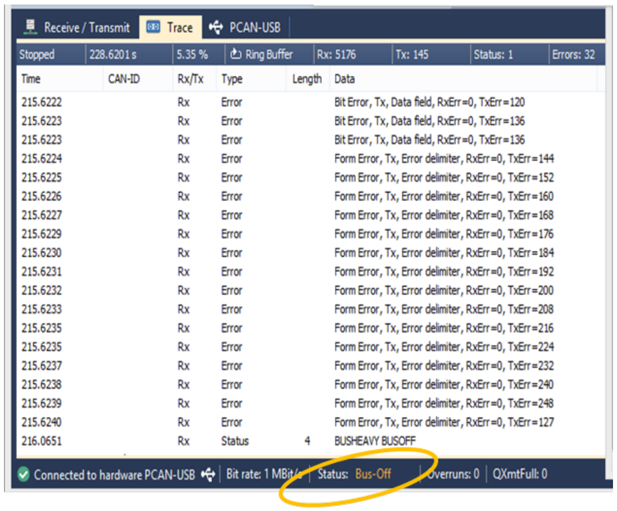
In this experiment we let *Alice* use the faster *FPGA* device, in order to reach the desired maximal traffic density and complete the defensive mission on her own. We again used the Python implementation of the *Parrot*, only this time, both *Eve* and *Bob* use the regular USB *Reg* device, and the *HTD* was in LOM. We used the following parameters:

- Eve (Reg): transmits attack messages with ID 00F, every 1 sec (1000msec), with DLC=7 and Data of 7 0xFF bytes.
- Alice (FPGA): transmits *Dmessages* with ID 00F, at the maximum allowed rate (about 8 messaegs per 1msec), with DLC=7 and Data of 7 0x00 bytes.
- Bob (Reg): passive-reactive (not transmitting messages, but reacting to the error conditions), and the *HTD* was in LOM.

With *Alice* transmitting at full speed, the *Parrot* defense was successful, and we consistently observed the behavior described in section III-C in all trails (see table I). Figure 10 is a screen shot of *Eve*'s terminal: note the last collisions with the first intercepting *Dmessage* (seen as *bit-errors*) driving *Eve*'s TEC to 136 - followed by 15 *form-errors*, as well as Eve's entrance to a *bus-off* state. Note that as in the previous experiment, both *Alice* and *Bob* were not affected by the second batch of collisions, and continued to lower their counters back to the safe ground of zero.

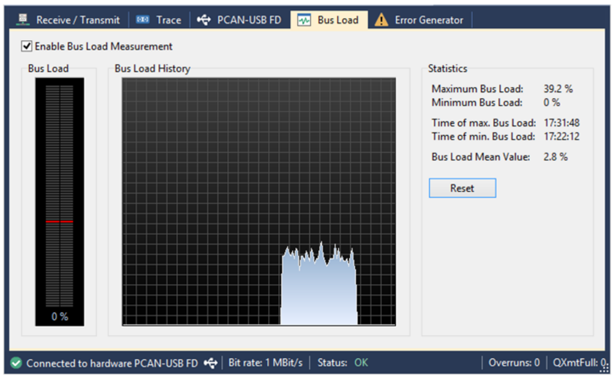
V. ASSISTING NEIGHBORS

We saw in the first experiment (Section IV-B) that some ECUs such as those using the Phillips SJA1000 CAN controller cannot transmit messages fast enough for a successful *Parrot* defense, for either technical (a single transmission register) or traffic-control (to prevent reaching full bus capacity) related reasons. However, note that to drive *Eve* from *error-passive* state to *bus-off*,



```
Stopped | 228.6201 s | 5.35 % | Ring Buffer | Rx: 5176 | Tx: 145 | Status: 1 | Errors: 32
Time       CAN-ID   Rx/Tx   Type     Length   Data
215.6222            Rx      Error             Bit Error, Tx, Data field, RxErr=0, TxErr=120
215.6223            Rx      Error             Bit Error, Tx, Data field, RxErr=0, TxErr=136
215.6223            Rx      Error             Bit Error, Tx, Data field, RxErr=0, TxErr=136
215.6224            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=144
215.6225            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=152
215.6226            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=160
215.6227            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=168
215.6229            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=176
215.6230            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=184
215.6231            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=192
215.6232            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=200
215.6233            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=208
215.6235            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=216
215.6235            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=224
215.6237            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=232
215.6238            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=240
215.6239            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=248
215.6240            Rx      Error             Form Error, Tx, Error delimiter, RxErr=0, TxErr=127
216.0651            Rx      Status   4        BUSHEAVY BUSOFF
Connected to hardware PCAN-USB | Bit rate: 1 MBit/s | Status: Bus-Off | Overruns: 0 | QXmtFull: 0
```

Fig. 10. Experiment 2 Eve's screen-shot results



```
☑ Enable Bus Load Measurement
Statistics
Maximum Bus Load:        39.2 %
Minimum Bus Load:        0 %
Time of max. Bus Load:   17:31:48
Time of min. Bus Load:   17:22:12
Bus Load Mean Value:     2.8 %
Connected to hardware PCAN-USB FD | Bit rate: 1 MBit/s | Status: OK | Overruns: 0 | QXmtFull: 0
```

Fig. 11. Bus load measurements from an operational vehicle

we just need a heavy bus-load and don't require more collisions with *Dmessages*.

A possible hypothesis is that the natural bus traffic in a real vehicle is dense enough. To check this hypothesis we used a CAN bus trace collected in [13] from a 2012 Ford Focus in various driving scenarios. We then converted this trace to let the HTD retransmit it into our simulated CAN bus, and measured the load on the bus. As Figure 11 shows, in the tested vehicle the bus load only reached 39%, with a median of 3 messages per 1msec (and a maximum of 5). This gives a typical gap of about 200μs between messages, which is not fast enough to collide with *Eve*'s *passive-error* flag (even with an SJA1000 defender).

However we can envision a system in which neighboring ECUs join the counter-attack, assisting the defender *Alice* to silence the attacker *Eve*. In order to evaluate the validity of this idea, we conducted the following experiment. In this experiment we let *Alice* use the regular PCAN-USB device based on the *SJA1000* controller.

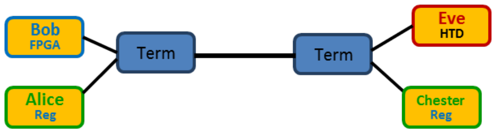


Fig. 12. Experiment 3, the assisting neighbor

Doing so allowed her to only reach a minimal gap of 31μs, which as we saw in section IV-B, is not enough to drive *Eve* into *bus-off* state. However, we added *Chester* as her assistant, to make sure that the minimal required gap is achieved by the combination of *Chester*'s and *Alice*'s transmitted messages. To demonstrate that we don't require anything special from *Chester*'s controller, we also let him use the regular (slower) PCAN-USB device. We had *Eve* use the HTD. We call *Chester*'s assisting transmissions *ADmessages*: These were his own all-zero *Dmessages* (with his own ID). The latter was chosen since also adding some extra potential protection in case of dealing with a more dynamic attacker who decide to attack several ECUs at a time.

In this experiment we used the following parameters:

- Eve (HTD): transmits attack messages with ID 00F, every 1 sec (1000msec), with DLC=7 and Data of 7 0x11 bytes.
- Alice (Reg): transmits *Dmessages* with ID 00F, at maximum speed (about 31μs gap), with DLC=7 and Data of 7 0x00 bytes.
- Chester (Reg): transmits *ADmessages* with an ID of 022, every $\sim 200\mu s$, with DLC=7 and Data of 7 0x00 bytes.
- Bob (FPGA): passive-reactive (not transmitting messages, but reacting to the error conditions).

We used the GUI to program *Chester* at a rate of five *ADmessages* per 1msec cycle. The *ADmessages* frequency was chosen so that *Chester*'s *ADmessages* gap < *Alice*'s *Dmessage* size + *intermission*. This was done to make sure that every *ADmessage* will be ready for transmission while *Alice*'s *Dmessage* is still being broadcasted. Since both *Chester* and *Alice A/Dmessages* are of 113 bits, the chosen frequency was sufficient (giving a gap of some 80μs). Note that in this experiment *Chester* used a lower priority *ADmessage* than both *Alice* and *Eve* (022 versus 00F), and that the data he transmitted was not important to our results.

Running the above experiment let us succeed in our defense by reaching the necessary minimal required gap of *intermission only* (as in the FPGA experiment in section IV-C) by creating a coupled batch of defensive *A/Dmessages* where *Alice*'s *Dmessage* waits for its turn while *Chester* is transmitting his own *ADmessage*, and vice versa. The resulting bus load was enough always to drive *Eve* into a *bus-off* state. Using a lower transmission frequency from Chester (4 or less messages per 1msec) was too slow to drive *Eve* into *bus-off*.

VI. Conclusion

In this paper we described a novel anti-spoofing software-only system for in-car CAN bus networks. The *Parrot* system blocks the attacker's lateral movement from a compromised ECU over the bus. Unlike previous firewall-based solutions or cryptography-based solutions, the spoofed messages are identified and destroyed by the legitimate message ID's owner, that can always detect a spoofed broadcast of one of its IDs. Our method does not merely drop messages that are non-conforming with policy: the *Parrot* defense typically disconnects the compromised ECU from the bus. And unlike previous solutions, that require a modified controller (since they violate the CAN bus protocol), our method is able to shut down the attacker *while obeying* the protocol rules. Hence, the *Parrot* defense can be added as a software-only patch to any standard ECU.

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] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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].

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

[17] 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.

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

[19] 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.

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

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

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

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

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

[25] 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.

[26] 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.

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