Analyzing the Capabilities of the CAN Attacker

Sibylle Fröschle[1(✉)] and Alexander Stühring[2]

[1] OFFIS & University of Oldenburg, Oldenburg, Germany
froeschle@informatik.uni-oldenburg.de
[2] University of Oldenburg, Oldenburg, Germany
alexander.stuehring@informatik.uni-oldenburg.de

Abstract. The modern car is controlled by a large number of Electronic Control Units (ECUs), which communicate over a network of bus systems. One of the most widely used bus types is called Controller Area Network (CAN). Recent automotive hacking has shown that attacks with severe safety impact are possible when an attacker manages to gain access to a safety-critical CAN. In this paper, our goal is to obtain a more systematic understanding of the capabilities of the CAN attacker, which can support the development of security concepts for in-vehicle networks.

1 Introduction

The modern car is controlled by a large number of Electronic Control Units (ECUs), which communicate over an internal network of bus systems. One of the most widely used bus types is called *Controller Area Network (CAN)*. Recent automotive hacking [3,8,11] has shown that attacks with severe safety impact are possible when an attacker manages to gain access to a safety-critical CAN. Usually such an attack will require several stages. For example (c.f. Fig. 1): first, the attacker gains remote code execution on the telematics ECU via its cellular interface by exploiting a software vulnerability; this gives him access to the infotainment CAN. Second, the attacker compromises the gateway ECU that separates the infotainment CAN from the powertrain CAN. Third, he injects cyber-physical messages into the powertrain CAN: e.g. he can abuse messages that tell the power steering ECU to change the steering angle; such messages are usually sent from the Park Assist ECU during automatic parking.

There is currently much activity on how to complement automotive safety processes by security. Draft norms such as SAE J3061 prescribe a concept phase in which a cybersecurity concept must be developed that shows how risk is reduced to an acceptable level. In the example above, the cybersecurity concept might take a security-in-depth approach where the telematics and gateway ECU are hardened by traditional security mechanisms while the last stage is defended by CAN-specific IDS and/or safety measures, which e.g. enforce that certain commands (such as those that steer during automatic parking) are only executed at low speed. Measures to prevent the worst at the last stage are desired since they take weight from the outer layers concerning their *safety integrity levels*.

S.N. Foley et al. (Eds.): ESORICS 2017, Part I, LNCS 10492, pp. 464–482, 2017.
DOI: 10.1007/978-3-319-66402-6_27

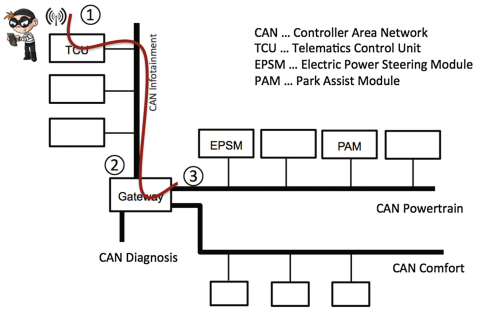

Fig. 1. Stages in automotive hacking

Thereby motivated, our focus here is on the last stage: once an attacker has made it to the last stage, what exactly are his capabilities? And how can they be captured in terms of abstract categories that can be used for a model-based evaluation of an automotive security concept? Since CAN is a broadcast network it is obvious that the attacker can eavesdrop and insert messages but less clear whether he can also delete or modify messages (such as the Dolev-Yao attacker).

Our contributions are as follows: (1) We motivate and define a threat model for the CAN attacker (Sect. 2.3). (2) We explore the capabilities of this CAN attacker (Sect. 3). Inspired by [14] we started out by systematically exploring whether and how the CAN attacker can realize the categories of the Dolev Yao attacker. This led us to identifying 6 categories of attacks that seem best suited for controller networks. Altogether, we show that by abusing error handling and configuration options at controller level the attacker has considerable power: he can silence or impersonate a target node as well as suppress and modify messages under certain conditions. For each basic category we show concrete attacks and demonstrate by experiment their feasibility. Many of our attacks are new. (3) We discuss the implications of these capabilities for automotive vehicles (Sect. 4). After presenting related work (Sect. 2.1) and the necessary background on CAN (Sect. 2.2) we proceed according to these contributions.

2 Background and Problem Statement

2.1 Related Work

Attacks on CAN. Security threats to automotive CAN networks have first been investigated by Hoppe et al. [5], based on automotive hardware in a lab. Koscher et al. [8] provide a comprehensive security analysis of two types of modern automobiles, which demonstrated the first attacks on real CAN networks with severe safety impact. While these attacks still required physical access to the cars via e.g. the OBDII diagnosis port, in another paper [3] it was shown that such attacks can also be done by hacking into the vehicle via its extensive attack surface. Since then many more attacks have been demonstrated by security experts

such as Miller and Valasek [11,20]. In [19] we have investigated what effect injecting sensor data has on a driving assistance system. Klebeberger et al. have pointed out in [7] that mechanisms implemented for safety, e.g. fault detection mechanisms may be abused by an attacker. In a recent paper [4] Cho and Shin have presented a bus-off attack, where similarly to one of our (independently designed) attacks collisions force a target ECU into bus-off.

IDS for CAN. There are several suggestions of how IDS (Intrusion Detection Systems) can be devised for in-vehicle networks. In [5] Hoppe et al. discuss IDS based on message frequency, obvious misuse of message-IDs, and other communication characteristics. Other approaches are based on entropy-based anomalies in the network [12], anomalies in the message frequency or other metrics [11,13,17], or are specification-based [9]. It remains to be investigated whether reliable IDS can be devised against attacks that are based on the generation of errors.

Crypto for CAN. The EVITA project has provided the first comprehensive security architecture for in-vehicle networks. The architecture is anchored in hardware security modules (HSMs), and realizes crypto-based security services such as secure boot, secure storage, and secure communication between in-vehicle components as well as for vehicle-to-x-communication [1]. Moreover, the hardware components have been evaluated for their use in the real-time critical in-vehicle environment [21]. By now most providers of automotive electronic components offer embedded security solutions such as automotive controllers with embedded HSMs or add-on security chips. Furthermore, cryptographic schemes for lightweight authentication over CAN have been developed (c.f. [15] and references therein).

However, it is not clear yet how the available components will be configured and employed as part of a comprehensive in-vehicle security concept that is economical, real-time suitable, and usable. Steps towards this are put forward in [10], and pursued by the SeSaMo project for embedded systems [16].

2.2 CAN - Controller Area Network

Controller Area Network (CAN) [2,6] is a bitstream-oriented broadcast bus with a maximal bit rate of 1 Mbit/s. The CAN protocol covers the physical layer and the data link layer. The physical layer can have one of two values: *dominant* or *recessive*. If two or more nodes transmit dominant and recessive bits at the same time then the resulting bus level will be dominant. This is for example realized by a wired-AND implementation. Hence, the dominant level is represented by a logical 0, and the recessive level by a logical 1. This electrical characteristic plays an important role for arbitration and error signalling.

Message Transfer. Each sender transmits their message without a destination address; rather every message contains an *identifier (ID)*, which indicates the meaning of the message. All nodes connected to the bus receive the message and decide by filtering on its ID whether the message is to be ignored or processed. The ID also assigns a priority to the message: the message with the smallest value of the ID has the highest priority.

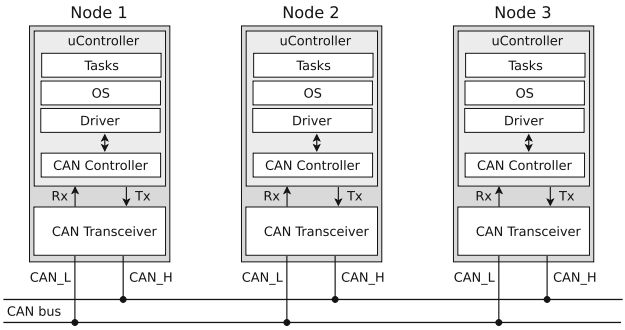

Fig. 2. A CAN network

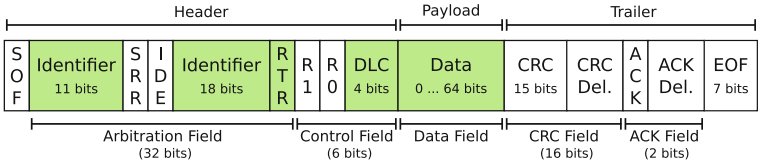

Fig. 3. Format of a data frame in extended format

CAN defines four different types of messages, called *frames*. The following two are particularly relevant here: a *Data Frame* carries data with a payload between 0 and 8 bytes; an *Error Frame* is transmitted to signal a bus error. Figure 3 depicts the format of a Data Frame (in Extended Format). The frame starts with the *Start of Frame (SOF) Field*, which is a single dominant bit. Then follows the *Arbitration Field*, which consists of a 29 bit *Identifier* and fixed-form fields. The *Control Field* contains the *Data Length Code (DLC)*, which records the number of bytes in the *Data Field*. Then follows the *Data Field* with 0 to 8 bytes of data. The *CRC Field* contains a cyclic redundancy check code calculated over the previous fields; followed by the *CRC Delimiter*: a single recessive bit. All receivers will acknowledge the successful receipt of the message. This is realized by the *Ack Field*: the transmitter sends a recessive bit in the *Ack Slot* while a receiver acknowledges a message by superscribing the Ack Slot by a dominant bit. The frame is concluded with the *End of Frame* consisting of 7 recessive bits.

Data Frames are always preceded by an *Interframe Space (IFS)*. The IFS consists of a fixed period of 3 recessive bits, called *Intermission*, in which no node is allowed to transmit a new frame, and is followed by a period *Bus Idle* of arbitrary length. In the latter any node can start to transmit a message (unless it is in an error state).

To resolve contention when more than one node wants to transmit, CAN uses *bitwise arbitration:* during transmission of the Arbitration Field every transmitter monitors the signal on the bus and compares it to the value of the bit it has transmitted itself. If the values are equal then the node will continue to send. If the node has sent a recessive bit but monitors a dominant bit on the

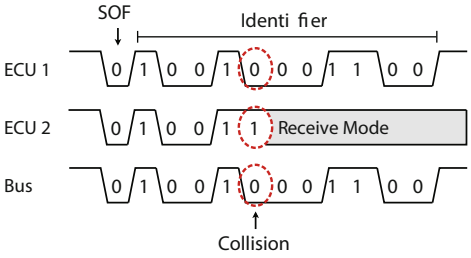

Fig. 4. Bitwise arbitration

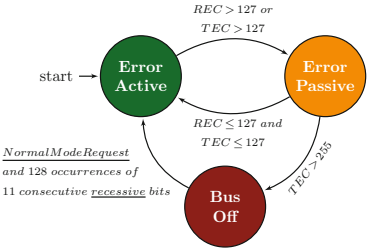

Fig. 5. Fault confinement

bus, it will withdraw from sending and become a receiver. Since the value 0 is represented by a dominant level thereby the conflict is resolved according to the priority - without losing information or time. An example is provided in Fig. 4. Frames that have lost arbitration or frames that are corrupted by errors will usually be *retransmitted automatically* (according to bitwise arbitration) until the transmission is successful.

To ensure that the nodes remain synchronized during the transmission of a message CAN uses the method of *bit stuffing*: after five consecutive bits of identical value are transmitted a complementary bit is inserted to enforce a change of signal level to synchronize on. Stuff bits are automatically inserted and removed by the transmitting and receiving controllers.

Error Handling and Fault Confinement. CAN provides several mechanisms for error detection, and distinguishes between five error types. The following three will be relevant later. When a node transmits a bit it also monitors the bus; a node detects a *bit error* when the monitored bit is different from the transmitted bit. (There are some exceptions as e.g. during arbitration.) All nodes also check whether bit stuffing is observed, and whether the form of fixed-form bit fields is observed. This will lead to a *stuff*, and *form error* respectively.

A node can be in one of three error states: *error-active*, *error-passive*, or *bus-off*. Figure 5 depicts the transitions between these error states. The transitions are governed by two error counters that CAN nodes keep: the *Transmit Error Counter (TEC)* and the *Receive Error Counter (REC)*. The counters are increased and decreased according to 12 rules specified in the CAN standard. Roughly, a node will increase its TEC by 8 when it detects an error during transmission, and decrease it by 1 after a successful transmission. A node will increase its REC by 1 when it detects an error during receiving, and further by 8 when it becomes clear (during error signalling) that it detected the error earlier than the other nodes. Decreasing is similarly as for transmission.

When an *error-active* node detects an error then it signals this by 6 dominant bits (*Active Error Flag*). This deliberately violates bit stuffing so that the other nodes will also detect an error. An *error-passive* node signals an error by 6 recessive bits, and then waits for 6 bits of equal polarity on the bus (*Passive Error Flag*). The violation of bit stuffing will only be noticed by other nodes when the error-passive node is a transmitter, otherwise the passive error flag

will not disturb the bus. Both error-active and error-passive nodes complete their Error Frame with the *Error Delimiter* of 8 recessive bits. (C.f. Fig. 6.)

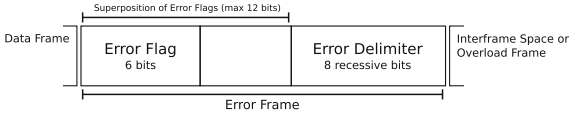

Fig. 6. Active error frame. A passive error frame is similar only that the superposition field can be longer than 12 bits: the node will wait for 6 bits of equal polarity

Moreover, after an *error-passive* node has been in the role of a transmitter it will extend the *Intermission* period by a *Suspend Transmission* period of 8 recessive bits before transmitting a further message (while it can receive).

2.3 Threat Model and Problem Statement

As motivated in Sect. 1 we are concerned with an attacker who has already compromised a node, say N_A, (such as the gateway ECU) in a safety-critical target CAN, say C_T, (such as the powertrain CAN). The goal of this attacker is not to hack into other nodes on C_T but rather to induce them to perform cyber-physical actions conveyed via C_T from his node N_A. Hence, we assume: *(1) The CAN attacker can host his own code on the compromised node N_A. However, we assume platform integrity for all other nodes of the target CAN.*

Moreover, we assume that the CAN attacker has compromised N_A remotely, and thus: *(2) The CAN attacker has no physical access to the vehicle instances that will be affected by his attack. He will launch his attack via malware on N_A* (which might be remotely controlled or not).

However, the attacker can *prepare* his attack with full access to a vehicle of the type he wishes to target. Security experts have demonstrated that in-depth knowledge about in-vehicle networks can be obtained by buying a car and reverse-engineering it. Stuxnet is also a point in case for sophisticated attack preparation. Hence, an automotive security concept should be based on the following overapproximation: *(3) The CAN attacker can prepare his attack off-line under a white box assumption and access to an instance of the vehicle type he wishes to attack. We assume he has full knowledge of message scheduling and architecture of the in-vehicle network.*

Automotive hacking so far has injected messages from a task layer (c.f. Fig. 2). However, if an attacker has compromised an ECU he usually also has access to lower software layers such as the interface to the CAN controller and configuration. So unless N_A is equipped with special security features, we assume: *(4) The attacker code can contain any command that controls CAN communication. This includes the basic functions for sending and receiving CAN messages but also standard functionality for controlling configuration and status registers. Hence, we assume the CAN attacker can make full use of the interface provided by the CAN controller of N_A.*

We consider the following problem: Given a target CAN C_T with a compromised node N_A and any number of honest nodes, which capabilities does the CAN attacker have apart from eavesdropping and inserting messages?

3 Attacker Capabilities

We now analyse the capabilities of the CAN attacker starting out from straightforward denial-of-service attacks to more targeted attacks. A detailed description of the experiments, the data, and a market analysis on features of CAN controllers is available on https://vhome.offis.de/pi/downloads/esorics2017/.

1. Blocking Messages by Priority. Similarly to a standard network attacker, the CAN attacker can disturb the target network by flooding it with messages. However, since CAN is priority-based the impact of flooding depends on the priorities of the messages involved.

Attack 1 (Flood to Block). *Let M be a message such that M is not sent by any honest node, and let $LP(M)$ be the set of all messages with priority lower than M. Then the attacker can block all messages in $LP(M)$: he simply floods the bus with the message M from his node N_A.*

When message M is flooded M will be ready for arbitration every time the bus becomes idle. Hence, messages by honest nodes can only win arbitration if they have a priority higher than M. Flooding with a message that is already allocated and regularly sent by an honest node would lead to bit errors (c.f. Sect. 3(4)). Our experiments show that the blocking indeed works reliably. Note that by flooding the bus with M such that $ID(M) = 0x0$ the attacker can block all messages of honest nodes (provided that $0x0$ is not already allocated).

2. Disrupting the Target Network. If the CAN attacker seeks to disrupt all behaviour on the target bus he can make use of functions that change the operating mode or the configuration of the CAN controller. Most CAN controllers have a feature that allows the attacker to induce a stream of dominant bits on the bus.

Example 1 (Test Mode). Many CAN controllers can be operated in a *Test Mode*, in which the state of the Rx pin of the CAN controller is clocked onto the Tx pin (c.f. Fig. 2). This mode is intended for testing the controller during circuit development. Invoking the Test Mode on a controller that is connected via a transceiver to a running CAN bus has a simple effect: once a dominant bit is received the transceiver continuously applies the dominant level on the bus.

Example 2 (GPIO Configuration). Usually, the Rx and Tx pins connecting the built-in CAN controller on the microcontroller to the transceiver are GPIO (General Purpose Input/Output) pins. They can either be assigned to a component

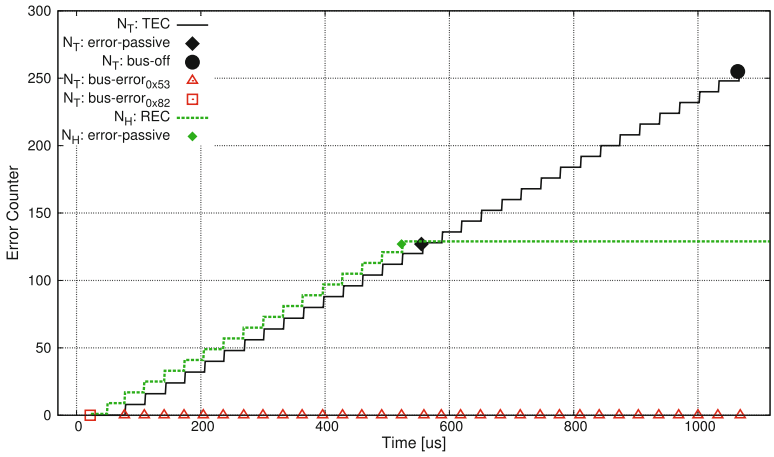

Fig. 7. Typical course of Attack 2, where N_T is a transmitter, and N_H is a receiver. Key to error codes: 0×82 is a stuff error in bit 28–21 of the Identifier Field; 0×53 is a form error due to "dominant for more than 7 bit times after active error flag"

such as the CAN controller or driven manually. The CAN attacker can access the GPIO configuration, disconnect the CAN controller from the IO pins, and interface directly with the CAN transceiver. This allows him to continuously send a dominant bit to the transceiver.

Our market analysis has shown that out of 23 microcontroller series with in-built CAN controller only one does not provide one of these features (7 support Test Mode, 18 GPIO configuration). We formulate and implement the attack based on the Test Mode, but the GPIO technique could be employed similarly.

Attack 2 (Disrupt by Dominant Bits). *Say the attacker wishes to disrupt the target CAN so that no messaging is possible at all. He simply invokes the Test Mode on his node N_A. He can stop the attack at any time by setting the operating mode back to normal. The bus communication will immediately be restored, but typically one node will be bus-off, and therefore remain silent.*

Once the attacker has invoked the Test Mode on N_A, and once one of the honest nodes has transmitted a dominant bit the behaviour on the bus will be reduced to a stream of dominant bits. A valid CAN frame can never contain more than 5 consecutive dominant bits: this either violates bit stuffing or the format of fixed-form fields. Hence, after at most 5 dominant bits each honest node will detect an error, and send an error frame. An error frame is completed with the Error Delimiter, which consists of 8 recessive bits. Since there are only dominant bits on the bus CAN fault confinement kicks in: after each additional 8 consecutive dominant bits each node will increase its error counter by 8. Hence, all honest nodes will quickly become error-passive (when their REC or TEC reaches 128). Nodes that were acting as transmitters when they first detected an

error will further go into bus-off (when their TEC reaches 256). Although there can be several transmitters during arbitration and it is possible that several nodes become bus-off in a well-scheduled CAN system with a usual load of about 80% one would expect that typically one node will be bus-off.

In each of the 10 experiments we conducted the receivers reach error-passive at $501^{\pm}1$ us after occurrence of the first error. The one transmitter either goes error-passive at the same time or after another $32^{\pm}1$ us. The latter applies when the transmitter detects a stuff error during arbitration; in this case the error counter is not increased as usual due to an exception of the CAN protocol [2]. Bus-off is reached by the transmitter after another $509^{\pm}2$ us. (Note that $512 \sim 16 \times 8$ bit-time.) Figure 7 shows a typical course of the attack. After the attacker resets to normal operating mode the error-passive nodes will be able to transmit their messages (at most subject to a suspension period of 8 bits) while the bus-off node will remain silent.

3. Silencing a Target Node by Dominant Bits. The disrupt attack is straightforward to implement and typically forces one node into the bus-off state in ≈ 1 ms (or 260 bit-time). However, the attack neither directs which node nor whether any node will become bus-off. Say the attacker wishes to silence a target node N_T. If he manages to synchronize the activation of the stream of dominant bits with the transmission of a message by N_T then he can force N_T bus-off in a targeted fashion. It turns out that there are several techniques to synchronize the attack with the transmission of a particular message.

Example 3 (ID Ready). Many CAN controllers have a feature called *ID Ready Interrupt*: an interrupt that is triggered as soon as the ID of a frame has been received while the rest of the frame is still in transit. Once the interrupt is raised the ID can be read from a register and compared to that of a target message, say M_T. A subsequent action of the attacker such as switching on the Test Mode will take effect while the rest of the message is still being transmitted. Our market analysis shows that 5 series of microcontrollers of 23 in total offer this feature.

Example 4 (Scheduling). If the CAN controller does not provide the ID Ready Interrupt he can make use of CAN message scheduling: CAN messages are typically sent with a fixed periodicity. Say message M_T has a period of t *ms*. The attacker waits to receive an instance of M_T, and can now predict that the next M_T will arrive after t *ms* plus some jitter. This will only work if the scheduling is precise up to the length of M_T.

Example 5 (Preceded IDs [4]*).* In [4] Cho and Shin define a *preceded ID* of a message M_T as the ID of a message that has completed its transmission right before the start of M_T. The attacker waits to receive the preceded ID message, and can now predict that M_T will be sent after 3 bit of Intermission. They also show that in real automotive CAN traffic preceded IDs often exist. Moreover, they show that preceded IDs can be fabricated when a target message M_T does not have them. This technique allows them to synchronize very precisely on the first bit of a target message.

We formulate and implement the attack based on ID Ready and Test Mode.

Attack 3 (Silence Target by Dominant Bits). *Say the attacker wishes to silence a target node, say N_T. He can achieve this as follows. He chooses a message M_T that is sent by N_T. In the task on N_A the attacker enables the ID Ready Interrupt, and programs the ISR that handles the interrupt as follows: the ISR compares whether the received ID matches ID(M_T). If this is true then the Test Mode will be invoked for ca. 280 bit-time. Then the operating mode is switched back to normal operation.*

In all of our 10 experiments N_T goes bus-off at $1010^{\pm}1$ us (or 253 bit-time) after occurrence of the first error. The course of the attack is similar to that depicted in Fig. 7 but without the offset due to the exception of stuff errors during arbitration: the first error is consistently a bit error in the DLC field.

4. Silencing a Target Node by Collisions. Arbitration in CAN is based on the assumption that it won't happen that two nodes send a data frame with the same ID at the same time: they would both win arbitration, and an error would occur in the DLC or Data Field unless they contain exactly the same payload. More precisely, letting i be the first bit position where the two frames differ, a bit error will be detected by the node that sends the frame with a recessive bit at position i. Figure 8 gives an example. In a real CAN system messages are allocated so that each ID is mapped to a unique node, from which messages with this ID will be sent. However, collisions can be deliberately caused by an attacker to force a target node into bus-off.

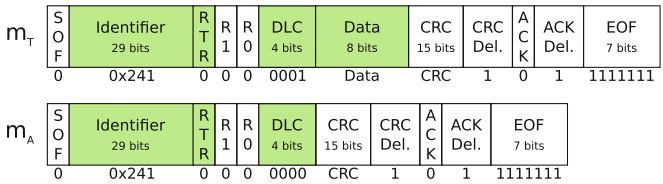

Fig. 8. The frames m_T and m_A will lead to a collision when they are transmitted at the same time. The sender of m_T will detect a bit error at the 4th bit of the DLC Field

Attack 4 (Silence Target by Collisions). *Say the attacker wishes to silence target node N_T. He picks a message m_T that N_T sends in a regular interval. He composes a message m_A such that sending m_A and m_T at the same time will raise a bit error at N_T. Hence, the attacker chooses m_A such that m_A has the same ID as m_T and there is a first bit position i at which m_A differs from m_T in that m_A has a dominant bit while m_T has a recessive bit. (This must necessarily be in the DLC or Data Field.) He then floods m_A from his node N_A.*

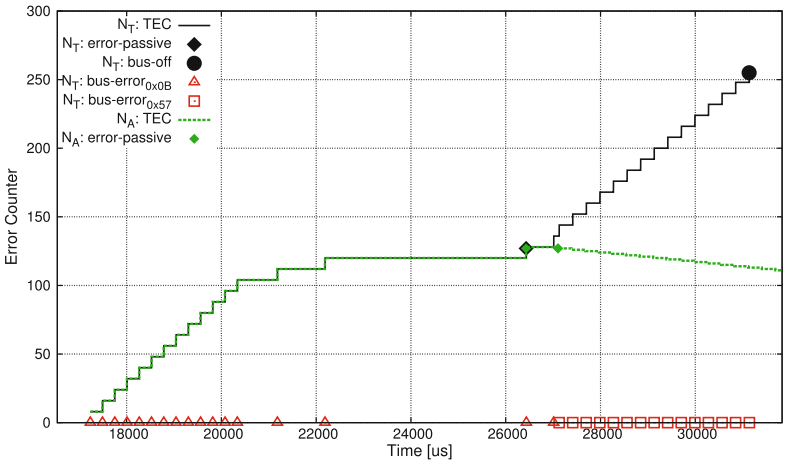

Fig. 9. Typical course of Experiment 4. Key to error codes: $0x0B$ is a bit error in the DLC Field; $0x57$ is a form error in the Error Delimiter

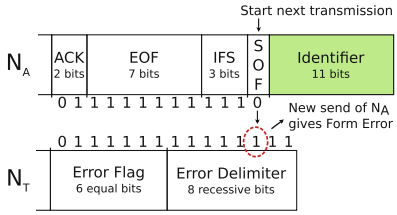

Fig. 10. Special situation of a form error in the Error Delimiter of N_T

The course of this attack is more complex, and goes over several stages. We explain the course of the attack when the TECs of N_A and N_T are initially 0.

Stage 1: Initially, both N_A and N_T are in the error-active state. Then as soon as N_T tries to transmit m_T there will be a collision (due to flooding of m_A). N_T will detect a bit error at position i as intended, and signal an active error flag. As a consequence all other bus nodes, including N_A, will also detect an error and signal error flags. Both N_A and N_T will increase their TECs by 8. After the Intermission period of 3 recessive bits N_A and N_T will try to retransmit m_A, and m_T respectively. This will again lead to a collision. This continues until both N_A and N_T go into error-passive (when their TECs reach $\geq$128, i.e. after 16 transmission attempts).

Stage 2: Both N_A and N_T are error-passive. They will both try to retransmit m_A and m_T at the same time, possibly after a Suspend Transmission period. Again N_T will detect a bit error. But this time N_T will signal a *passive* error flag. N_A's transmission will continue to dominate the bus, and neither N_A nor any other node will detect an error. N_A will transmit m_A successfully, and decrease its TEC by 1. In contrast, N_T will increase its TEC by 8.

Stage 3: N_A is now error-active again. N_T remains error-passive, and tries to complete its passive error flag while N_A is still transmitting m_A. The attacker can now profit from a "blind spot" of the CAN protocol [22] when the bus load is 100%. By how error signalling is defined N_T can complete its passive error flag only when it first detects 6 consecutive equal bits on the bus. This will only happen with the 5th bit of the EOF Field of m_A. As a consequence m_A (or a message of higher priority) will start to be transmitted while N_T is still in the field Error Delimiter of 8 recessive bits. This will cause a form error at N_T. (C.f. Fig. 10) Moreover, due to the flooding of m_A this situation will occur again and again whenever N_T tries to complete the next error frame. Hence, N_T will quickly go bus-off: when its TEC reaches $\geq$256, i.e. after 15 such form errors.

We have conducted 10 experiments (with the TECs N_A and N_T initially 0) that confirm that a target node can be forced bus-off in this way. In the 10 experiments the time from the first collision to bus-off of N_T ranges from 8,8 ms to 13,9 ms. The variation results from how many and which regular messages of the other nodes are interspersed. However, modulo the pattern of interspersed messages all experiments follow exactly the course explained above. Figure 9 shows the precise course of one of the experiments.

If the initial value of the TECs of N_A and N_T is not 0 the stages of the attack can be slightly different. However, Attack 4 is robust: we have confirmed by experiment that it works even in the worst initial situation when $TEC(N_A) \gg TEC(N_T)$. The flooding of m_A could easily be spotted by an IDS. However, the attack can be optimized to proceed more covertly: assume or ensure (by resetting $TEC(N_A)$ to 0) that $TEC(N_A) \leq TEC(N_T)$; use the technique of preceded IDs to precisely synchronize on the arrival of m_T rather than flooding to cause the first collision. Send one m_A synchronized with m_T. After m_A has finally been transmitted successfully send another m_A followed by a stream of messages that are as inconspicuous as possible but keep the bus busy until N_T is bus-off. The second m_A is only necessary when $TEC(N_A) < TEC(N_T)$: when N_A finally manages to transmit the first m_A, N_T might receive it successfully while in *Suspend Transmission*.

5. Suppressing a Target Message. In the previous two attacks the attacker deliberately causes an error while a target message M_T is being transmitted by a node N_T. Although this has the effect of suppressing this instance of M_T, the CAN features automatic retransmission and failure confinement together make it impossible to suppress M_T with a long-lasting effect while keeping N_T alive: either the automatic retransmission of M_T will be successful or N_T will accumulate errors and go bus-off. However, the newest version of the CAN standard [6] makes automatic retransmission optional: it may be disabled, or limited to a certain number of attempts. This can be exploited in the following attack:

Attack 5 (Suppress a Target Message). *Say the attacker wishes to suppress a target message M_T. Provided that the honest node that sends M_T, say N_T, has disabled automatic retransmission, he can achieve this as follows. In the task on N_A the attacker enables the ID Ready Interrupt, and programs the ISR*

that handles the interrupt as follows: the ISR compares whether the received ID matches ID(M_T). If this is true the Test Mode will be invoked for ca. 6 bit times. Then the operating mode is switched back to normal.

The attack proceeds exactly as Attack 3 apart from that now the Test Mode is invoked for only a few bits: just enough to prevent the successful transmission of M_T. Without retransmission the bus behaviour will be immediately back to normal. When the next M_T arrives it will again be captured by the ID Ready Interrupt and suppressed. N_T will increase its TEC with every suppression, and might go bus-off after a number of intervals. The exact number of intervals depends on how precise the suppression is, and on the number of messages (other than M_T) successfully sent by N_T: every successful transmission will decrease the TEC by 1. In every of our 10 experiments we manage to suppress 50 intervals of M_T. N_T goes bus-off only after $1000^{\pm}0,6$ ms from the first arrival M_T. Other variants of this attack not based on configuration features seem also possible; e.g. use the technique of preceded IDs to precisely synchronize on M_T and a collision to suppress it.

6. Modification Attacks. So far, we have only seen denial-of-service attacks against the target bus, a target node, as well as blocking and suppression of messages. However, the attacker is also capable of composite modification attacks:

Attack 6 (Impersonate Target Node). *Say the attacker wishes to impersonate a target node N_T. He can achieve this in two phases: first, he silences N_T by one of the 'Silence Target Node' attacks. Second, he injects the pattern of messages usually sent by N_T but modified by forged values.*

Naturally, Attack 6 can also be used when the attacker wishes to modify a particular message. Another way to achieve message modification, which does not require N_T to be forced bus-off, is this:

Attack 7 (Modify Target Message by Suppress and Inject). *Say the attacker wishes to modify a target message M_T. Provided that the node that sends M_T has disabled automatic retransmission, he can achieve this as follows: he runs one of the 'Suppress Target Message' attacks against M_T and after each suppression he injects a new instance of M_T with his own forged payload.*

In [19] we have employed yet another way to modify messages. Messages with data such as a particular sensor value are often not read on each arrival by higher layers but rather periodically from a dedicated receive buffer; it is allowed that a message can be overwritten by a new one with the latest sensor reading. But then a message M_T can be modified by deliberate buffer overwrite: the attacker hooks a new instance of M_T with his own payload onto the real instance of M_T. The disadvantage (for the attacker) is that this method will lead to messages with conflicting values on the bus, which can be detected by an IDS.

Summary and Attacker Model. We provide a summary in Fig. 11. For each attack we record time (how fast can the attack goal be reached?) or duration (how long can the effect be sustained?), which traces it leaves on the bus (in view of IDS), and which conditions are necessary to implement it. Most of our attacks only leave error traces on the bus, and it remains an open problem whether an IDS can be constructed to detect them reliably. This will require more research into which error patterns typically occur in real CAN systems. One further challenge is that our attacks can be varied so that the error patterns they produce will be less regular than the fastest or most straightforward versions we have discussed here.

Cat.	Attack		t or d	traces on bus	conditions
B	1	Block	any d	flooding	suitable M
D	2	Disrupt	any d	errors only	config
SN	3	Dominant Bits	$t \approx 260$ bit-t	errors only	config & synch-M
SN	4^*	Collisions	$t \approx 16$ msg-t	≤ 16 errors	synch-B
SM	5	Dominant Bits	$d \approx 32$ periods	periodic errors	config & synch-M & rt-off
SM	5'	Collisions	$d \approx 32$ periods	periodic errors	synch-B & rt-off
IN	6	3 & Inject	$d \approx 260$ bit-t	errors only	config & synch-M
IN	6^*	4^* & Inject	$t \approx 16$ msg-t	≤ 16 errors	synch-B
MM	7	5 & Inject	$d \approx 32$ periods	periodic errors	config & synch-M & rt-off
MM	7'	5' & Inject	$d \approx 32$ periods	periodic errors	synch-B & rt-off
MB	8	[19] Buffer	any d	conflicting M_T	buffer overwrite

d . . . duration t . . . time to achieve bit-t . . . bit-time mesg-t . . . time of a message
config . . . access to configuration synch-B . . . synchronization on first bit of message
synch-M . . . synchronization on message rt-off . . . retransmission off

Fig. 11. Overview. 4^* is the optimal variant of 4; $5'$ is the collision variant of 5

The impersonation attacks can easily be detected by the target node itself: while N_T cannot transmit messages while in bus-off it can receive messages, and hence, recognize when another node sends messages allocated to itself. However, N_T has no way of signalling this to other nodes unless there is an additional uncompromised channel available. This is similar for the (MM) modification attacks: although N_T could try to send a warning over the target CAN the attacker could suppress the respective message. Altogether, we derive the abstract model for the CAN attacker shown in Fig. 12. Finally, note that (I), (IN), (MM), and (MB) can be prevented by securing the messages (i.e. the payload) cryptographically but this is not possible for the other attacks.

1. Eavesdrop on all messages transmitted on the target CAN **(E)**. *IDS?* No.
2. Insert any message into the target CAN at any time **(I)**. (But transmission will be subject to arbitration.) *IDS?* No, if injection follows the usual message pattern.
3. Block a set of target messages $LP(M)$ for any duration, where M is a message not sent by honest nodes **(B)**. *IDS?* Yes, can detect flooding of M on bus.
4. Disable the target CAN for any duration **(D)**. *IDS?* Open (only errors).
5. Silence a target node N_T **(SN)**. *IDS?* Open (only/mainly errors).
6. Suppress any target message M_T up to ≈ 32 intervals if automatic retransmission is disabled on N_T **(SM)**. *IDS?* Open (only errors).
7. Impersonate a target node N_T **(IN)**. *IDS?* Open (only/mainly errors); or by signalling from N_T if an additional uncompromised channel is available.
8. Modify any target message M_T up to ≈ 32 intervals if automatic retransmission is disabled on N_T **(MM)**. *IDS?* open (only errors); or by signalling from N_T if an additional uncompromised channel is available.
9. Modify any target message M_T if buffer overwrite is possible **(MB)**. *IDS?* Yes, can detect conflicting messages on bus.

Fig. 12. Capabilities of the CAN attacker

4 Cyber-Physical Implications

We now discuss the implications of our attacks for automotive vehicles. For this we make use of the insights gained by automotive hacking for real vehicles: for the Ford Escape 2010 and Toyota Prius 2010 [11], the Jeep Cherokee [20], and the vehicle of [8]. It turns out that the Jeep and the Toyota both use a Renesas V850ES/FJ3-uController, which has the GPIO reconfiguration option, for at least some of their ECUs. We focus on cyber-physical attacks that manipulate steering and braking.

Steering has been manipulated based on Advanced Driving Assistance Systems (ADAS) such as Park Assist. Park Assist mainly involves two ECUs: the Park Assist Module (PAM) and the Electric Power Steering Module (EPSM), which controls the servo motor attached to the steering wheel. When park assistance is activated the PAM calculates the steering movement based on sensor inputs and sends messages over the in-vehicle network that ask the EPSM to realize the steering motion. These messages typically specify directly the required steering wheel angle. Some safety measures might be in place that prevent the EPSM from executing the request in any context.

For example, the EPSM of the Toyota only accepts requests to change the steering angle when the vehicle is in reverse gear and at low speed. Data such as current gear and speed are typically broadcast in regular intervals on the CAN bus to make sensor readings accessible to ECUs such as PAM and EPSM. It turns out that the EPSM of the Toyota obtains the data for the safety checks via the same CAN bus as the steering commands. Miller and Valasek managed to override these checks as follows: a forged message for current gear was hooked in front of the steering message while forged speed values had to be continuously injected. This led to some ECUs become unresponsive. (C.f. [11].)

Example 6 (Steer Toyota covertly at any speed: avoid flooding). A more subtle attack could make use of the 'Impersonate Node' attack: the attacker silences the ECU responsible for broadcasting the current gear, and the ECU for broadcasting the current speed respectively. He then injects forged gear and speed packets that mimic the usual patterns of sending them. It is plausible that the impact of silencing these ECUs is no worse than the ECUs becoming unresponsive in the attack by Miller & Valasek (which was perhaps due to collisions). Moreover, if these ECUs do not use automatic retransmission for the gear and speed packets a 'Modify Message' attack is also possible, which might avoid any side effects.

The PAM of the Jeep Cherokee sends a CAN message with the following information: status, i.e. park assist on or off, torque to be applied, and a counter value. The message is not only sent when Park Assist is active but in a regular interval. This allows the EPSM to recognize when messages are injected that are conflicting with those sent by the real PAM, in which case Park Assist will go offline. Valasek and Miller have subverted this safety measure as follows: they first start a diagnosis session with the PAM, which will stop it from sending messages. However, since a diagnosis session can only be opened at low speed this restricts their attack to a speed of no higher than 5 mph. (C.f. [20].)

Example 7 (Steer Jeep at any speed: without diagnosis session). The attack can be improved by employing one of the 'Impersonate Node' attacks. This will silence the PAM without having to open a diagnosis session. Hence, the speed constraint does not apply (unless there are further checks), and the attack will remain covertly when IDS against diagnostic messages is used.

One of the most severe attacks against a vehicle is to disable its brakes while driving. Most vehicles have a diagnostic command that induces the ABS ECU to bleed or release the brakes with the effect that the driver cannot apply the brakes at all. 'Disabling brakes' has first been realized in [8], and also against the Ford [11] and the Jeep Cherokee [20] based on such commands. In the Ford and the Jeep this only works at low speed, enforced by the diagnostic session that needs to be opened first.

Example 8 (ABS: protected by direct line to sensor). It seems plausible at first that 'disabling brakes' can be leveraged to full speed by forging speed packets by means of an 'Impersonate Node' or 'Modify Message' attack as discussed in Example 6. However, it seems unlikely that this is possible: the wheel speed sensor is usually directly connected to the ABS ECU, and speed packets are broadcast from there to other ECUs. Hence, one would expect that the ABS ECU itself cannot be fooled by wrong speed packets forged over CAN.

Another potentially severe attack is to suddenly engage the brakes while driving. This has also been implemented based on diagnostic messages [8,11]. Another way to realize this is to exploit cyber-physical messages that are part of Collision Prevention Systems (CPS) [11,20]. Such systems can send messages to the ABS ECU that induce it to brake. This has been demonstrated against

both the Toyota and the Jeep. The Jeep has a safety measure analogous to that for Park Assist: the CPS module sends a message regularly, and the ABS ECU checks whether there are conflicting messages, in which case the ABS turns off CPS entirely. Valasek and Miller override this safety feature as before, by putting the CPS ECU into a diagnostic session, which restricts the attack to low speed.

Example 9 (Sudden brakes for Jeep at full speed). Analogously to Example 7 this attack could be improved by an 'Impersonate Node' Attack: to work without speed constraint and only based on messages that are used during normal operation.

Messages that are part of CPS have to work at any speed, and hence, a safety measure based on speed checks is not an option here. The same is true for Lane Keep Assist (LKA). Lane Keep Assist will detect when the vehicle is in danger to veer from the lane, and intervene in the steering to correct this. This system involves the LKA module, a camera that detects the lines of the lane, and the EPSM. Similarly to Park Assist the LKA module transmits a steering request to the EPSM. While the Toyota's camera is directly connected to the driving support ECU the Jeep's Forward Facing Camera Module (FFCM) is a node on the CAN bus. The following demonstrates that even if cyber-physical messages are cryptographically protected one still has to guard against indirect attacks based on sensor data transmitted over a bus.

Example 10 (Steer Jeep by Faking the Environment). Silence the FFCM by a bus-off attack. Then play in a pattern of FFCM messages that mimic the values sent when the vehicle ventures off the lane. The LKA system will "correct" the steering correspondingly.

5 Conclusions

We have derived an abstract model for the CAN attacker, and demonstrated its usefulness by a discussion of potential implications for real cars. In future we will employ this model in our model-based safety and security analysis [18]. We do not consider this model to be static. In particular, it has to be extended by cryptographic mechanisms that might be available and timing information. Also, we expect there will be more Disrupt Attacks, e.g. based on a change of bit rate or polarity. However, we hope that the categories are stable.

Our analysis has revealed new attacks: all our attacks are new apart from the obvious 'Flood to Block'. Bus-off by collisions has also been shown in [4]. However, our (independently designed) Attack 4 works much faster: it only needs one interval compared to approx. 17 in [4]. The analysis has shown several directions for further experimental exploration such as a more systematic understanding of synchronization, and how real error traces look like in view of IDS. We will also explore whether small changes to CAN such as removing the 'blind spot' would make it easier to detect some types of attacks.

Acknowledgement. This work is supported by the *Niedersächsisches Vorab* of the Volkswagen Foundation and the Ministry of Science and Culture of Lower Saxony as part of the *Interdisciplinary Research Center on Critical Systems Engineering for Socio-Technical Systems.*

References

1. Apvrille, L., El Khayari, R., Henniger, O., Roudier, Y., Schweppe, H., Seudié, H., Weyl, B., Wolf. M.: Secure automotive on-board electronics network architecture. In: FISITA 2010 World Automotive Congress, vol. 8 (2010)
2. Bosch. CAN Standard. Bosch (1991)
3. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T.: Comprehensive experimental analyses of automotive attack surfaces. In: 20th USENIX Security, SEC 2011, p. 6 (2011)
4. Cho, K.-T., Shin, K.G.: Error handling of in-vehicle networks makes them vulnerable. In: 2016 ACM SIGSAC Computer and Communications Security, CCS 2016, pp. 1044–1055. ACM (2016)
5. Hoppe, T., Kiltz, S., Dittmann, J.: Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In: Harrison, M.D., Sujan, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 235–248. Springer, Heidelberg (2008). doi:10.1007/978-3-540-87698-4_21
6. ISO. Road vehicles controller area network (can) – Part 1: Data link layer and physical signalling. ISO 11898-1:2015 (2015)
7. Kleberger, P., Olovsson, T., Jonsson, E.: Security aspects of the in-vehicle network in the connected car. In: 2011 IEEE Intelligent Vehicles Symposium (IV), pp. 528–533 (2011)
8. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental security analysis of a modern automobile. In: IEEE Security and Privacy (2010)
9. Larson, U.E., Nilsson, D.K., Jonsson, E.: An approach to specification-based attack detection for in-vehicle networks. In: 2008 IEEE Intelligent Vehicles Symposium, pp. 220–225. IEEE (2008)
10. Lima, A., Rocha, F., Völp, M., Esteves-Veríssimo, P.: Towards safe and secure autonomous and cooperative vehicle ecosystems. In: Cyber-Physical Systems Security and Privacy, CPS-SPC 2016, pp. 59–70. ACM (2016)
11. Miller, C., Valasek, C.: Adventures in automotive networks and control units (2013) http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf
12. Müter, M., Asaj, N.: Entropy-based anomaly detection for in-vehicle networks. In: Intelligent Vehicles Symposium, pp. 1110–1115. IEEE (2011)
13. Müter, M., Groll, A., Freiling, F.C.: A structured approach to anomaly detection for in-vehicle networks. In: Information Assurance and Security (IAS) 2010, pp. 92–98. IEEE (2010)
14. Pöpper, C., Tippenhauer, N.O., Danev, B., Capkun, S.: Investigation of signal and message manipulations on the wireless channel. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 40–59. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23822-2_3

15. Radu, A.-I., Garcia, F.D.: A lightweight authentication protocol. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016, Part II. LNCS, vol. 9879, pp. 283–300. Springer, Cham (2016). doi:10.1007/978-3-319-45741-3_15
16. Sojka, M., Krec, M., Hanzálek, Z.: Case study on combined validation of safety & security requirements. In: SIES 2014, pp. 244–251. IEEE (2014)
17. Song, H.M., Kim, H.R., Kim, H.K.: Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In: Information Networking (ICOIN) 2016, pp. 63–68. IEEE (2016)
18. Strathmann, T., Fröschle, S.: Towards a model-based safety and security analysis. In: Model-Based Development of Embedded Systems (MBEES) (2017)
19. Stühring, A., Ehmen, G., Fröschle, S.: Analyzing the impact of manipulated sensor data on a driver assistance system using OP2TiMuS. In: Design, Automation and Test in Europe (DATE 2016) (2016)
20. Valasek, C., Miller, C.: Remote exploitation of an unaltered passenger vehicle, August 2015. http://illmatics.com/Remote%20Car%20Hacking.pdf
21. Wolf, M., Gendrullis, T.: Design, implementation, and evaluation of a vehicular hardware security module. In: Kim, H. (ed.) ICISC 2011. LNCS, vol. 7259, pp. 302–318. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31912-9_20
22. Yang, F.: A bus off case of can error passive transmitter. EDN Technical paper (2009)