CYMOTIVE
TRUSTED SMART MOBILITY

CANCAN

CAN-IN-CAN ATTACK FOR BYPASSING SECURITY

June 2022

Matan Ziv
Principal Cyber Security Researcher CYMOTIVE Technologies

CANCAN: Encapsulation of CAN-FD Messages for Circumvention of Security Measures

Matan Ziv
CYMOTIVE Technologies LTD
Tel Aviv, Israel
matan.ziv@cymotive.com

***Abstract* - "Don't look at the 'CANCAN' (Hebrew: pitcher), look at what's contained inside" is a Hebrew idiom, equivalent to the English idiom "Don't judge a book by its cover". The Controller Area Network (CAN) bus protocol allows communication between various components inside most modern-day vehicles. The introduction of the new Controller Area Network Flexible Data-Rate (CAN-FD) protocol allows for faster communication with a larger number of data bytes per message. As these protocols are used for passing critical messages between different components, many attacks were found, and many security measures were proposed to solve or restrict them. In this paper, a new way of compromising systems utilizing the CAN-FD protocol is presented. By introducing a crafted CAN-FD message encapsulating a legal CAN or CAN-FD message, components could potentially be made to accept the encapsulated internal message instead of the external message that was, in fact, sent on the bus. Furthermore, this paper will show how existing security solutions do not mitigate this attack and will propose effective mitigation solutions against it.**

Keywords – CANBUS, CAN-FD, Automotive Security

1 INTRODUCTION

CAN-FD introduces a new possibility of using a fast bitrate, for sending both DATA and CRC fields of the protocol messages. When a sampler using a lower bitrate examines a bitstream sent with a fast bitrate, it will sample a subset of the information as if it were slow bitrate bits. If these supposedly slow bits appear as a bitstream corresponding to a legal CAN (or CAN-FD) message, a CAN-FD controller may (in specific cases) accept this message – even though it was never actually sent on the bus (see Figure I).

This paper will discuss the usefulness of this case, the attack scenarios in which it may be used, and the technical details needed to build a usable encapsulating message.

2 PREVIOUS WORK

The following is an overview of the currently known attacks relevant to the CANBUS and CAN-FD communication. Several surveys were performed, aggregating the currently known attacks on the classic CANBUS protocol (e.g., [1], [2]). These known attacks can be split into two main types.

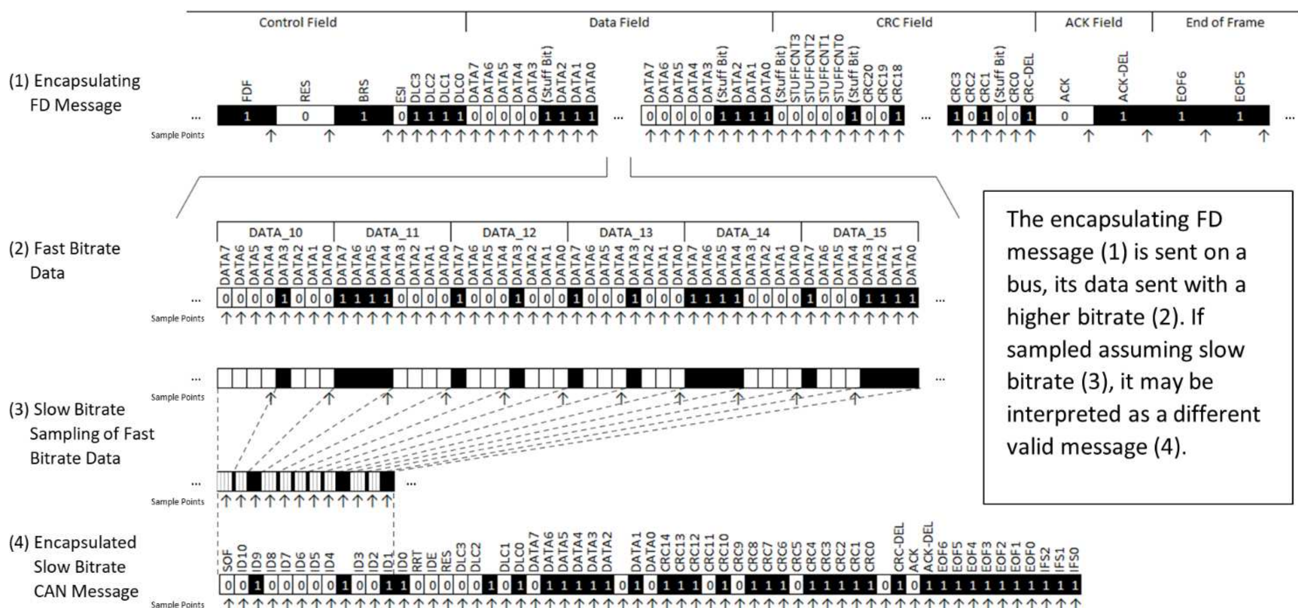

FIGURE I
SLOW SAMPLING OF A FAST BITRATE MESSAGE

It should be noted that most publications regarding security issues in the CAN-FD protocol seem to relist the same issues existing in the CANBUS domain. Efforts to locate any publications regarding security issues specific to the CAN-FD protocol were unsuccessful.

2.1 Can Frame Attacks

These are attacks utilizing valid sent messages. Such attacks will include the “Bus Flood Attack” (sending high priority messages, thus preventing other messages from being sent), “Spoofing” (the ability to send messages that should be sent by another entity), and “Sniffing” (listening to unencrypted data intended for another entity). These attacks may affect the availability, authenticity, and confidentiality of the system.

2.2 Can Protocol Attacks

These attacks involve the physical layer of the protocol and will require the ability to control the CANBUS by injecting specific bit values at specific times (“bit-banging”). Some of the proposed attacks will affect the availability of the system. These will include the “Bus-off Attack” (thoroughly explained in [3]), “Double Receive Attack”, and “Freeze Doom Loop Attack” (both presented in [4]). Another attack known as the “Janus Frame Attack” (reviewed in [5]) will use its control of the physical CANBUS to make different CAN controllers listening to the same bus see different valid payloads. This will break the atomic multicast feature of CAN (the implicit assumption that every device will see the same frame).

Notice that the “Janus Frame Attack” shares certain similarities with the proposed “CANCAN Attack”, as both involve making the bus simultaneously contain multiple valid payloads. However, the “Janus Attack” is a Can Protocol Attack, requiring that the attacker have access to the physical bus itself for changing the value sent on the bus faster than the sampling bitrate. The proposed “CANCAN Attack” may be categorized as a CAN Frame Attack, as it only requires sending a valid FD message over the bus, using the fact that an FD message’s data is naturally sent using a faster bitrate.

3 SYSTEM MODEL AND LIMITATIONS

- The fast bitrate used by CAN-FD is a multiplication of its slow bitrate. Specifically, this paper will assume (in most examples) a ratio of 4, as several real-world use cases were found to use this ratio.
- The controller will use the slow bitrate sampling to determine if the bus is IDLE (as discussed in 7.2). If the fast bitrate sampling is used instead, most of the attacks described here become unusable, at least directly. The attacks may still be possible given enough bus errors (see 7.1).
- A synchronization done on a recessive to dominant edge (1->0), which will later sample the bit ‘1’ will not be handled as an error. This adheres to the specification [6], but is not supposed to happen. Any controller that does handle this case as an error may break this assumption.
- The sampling of bits during the slow bitrate phase will always be at a point >=50% of the bit width. The specific sampling point will depend on the configuration of the controller. In specific cases (such as 9.3.1), there is a stricter assumption that the sampling point is in the 4th corresponding fast bitrate bit – meaning that the sampling will occur at a point >=75% of the bit width. In a case where the sampling point does not agree with this assumption, certain changes will need to be made to the proposed attacks for them to work.

4 GLOSSARY

- BRS – Bit rate switch. A bit in the CAN-FD CONTROL field, that, when set, will signal that the sending of the remaining CONTROL field bits, DATA field, and CRC field will be done using a higher bitrate. If cleared, all message bits are to be sent using the slower bitrate. When used throughout the paper, “BRS CAN-FD message” may denote a CAN-FD message with a set BRS bit.
- Fast Bit – a bit sampled by the faster bitrate sampler. This sampler is used for the DATA and CRC fields (and a subset of the CONTROL field) of a BRS enabled CAN-FD message.
- Slow Bit – a bit sampled by the slower bitrate sampler. This sampler is used for all CANBUS messages, for the ARBITRATION, ACK, and EOF fields (and a subset of the CONTROL field) of CAN-FD messages, and for all fields of a BRS disabled CAN-FD message. A sequence of “Fast Bits” sampled by a slower bitrate sampler will return a smaller number of sampled bits and will depend on the sampling and resynchronization rules of the sampler.
- Quiet Bits – the slow bits that must be prepended to the encapsulated message, for the controller to accept it. Its value will depend on the used attack scenario (as will be discussed in the attack scenario section 7), and will be equal to 11 recessive bits (in the Wakeup scenario 7.2) or 10 recessive bits (in the Bus Error scenario 7.1).

5 PAPER STRUCTURE

The paper will explain the usefulness of the CANCAN attack in section 6. The scenarios in which such an attack may be successfully performed are reviewed in section 7. Then, a deep dive into the different techniques for generating an encapsulating message is done in sections 8-10.

The attack setups used for demonstrating and verifying the proposed encapsulation techniques are presented in section 11, followed by suggestions of different protection methods against such an attack in section 12.

The paper will end with the conclusion in section 13, and the references in section 14.

6 ATTACK USEFULNESS

The type of attacker making use of CANCAN, is assumed to encounter the following situation:

- The attacker can send a subset of CANIDs to the target over a shared bus supporting the CAN-FD protocol. The target does not necessarily need to accept the message with the sent CANID, but it must reach the specific shared bus listened to by the target. An attacker may gain this capability by several methods, including (but not limited to):
  - Using a previous vulnerability to gain control over a component connected to the bus.
  - Connecting a physical CAN dongle to the OBD port of a vehicle.
- The attacker wants the target to receive a CANBUS message with a specific CANID (which may not be part of the subset of CANIDs that may be sent to the target by the attacker). An attacker may use this functionality in several ways, including (but not limited to):
  - Sending a diagnostic message (e.g., over the UDS protocol) which will enable some restricted functionality.
  - Injecting a signal with incorrect or malicious data which may itself trigger some functionality (e.g., a signal denoting engaged breaks).

If both the attacker and target reside in the same physics bus supporting the CAN-FD protocol, and there are no limitations on the messages the attacker may send, the attack is trivial and will not require CANCAN.

However, several security mechanisms exist which may prevent the straightforward attack. This paper will show how CANCAN may allow an attacker to circumvent these limitations.

6.1 Intrusion Detection/Prevention Systems Circumvention

An Intrusion Detection System (IDS) will attempt to recognize anomalies on the CANBUS. These anomalies may be based on incoming message properties such as time, CANID, and in certain implementations, the payload itself. When such an anomaly is recognized, the event is generally logged, but the attack is not prevented. Implementations may differ regarding access to the CANBUS.

An Intrusion Prevention System (IPS) will generally include all the mentioned functionalities of the IDS, with the addition of some blocking abilities. Several solutions will attempt to recognize a maliciously sent message in real time, making sure it does not reach its destination. This may be done in one of two ways. The first is by acting as a man-in-the-middle (MITM) between the attacker and the bus. As a MITM, the IPS will be able to refuse sending on a message going through it. The second way may be by injecting bits directly into the bus, while the malicious message is sent. This may mark the message as invalid for processing by any other components. Both options are used by the CAN-HG solution, as shown in [7].

The IDS/IPS filtering based on the CANID value may be trivially circumvented using the CANCAN method. An attacker may send a message with a legal CANID, that does not trigger the IDS/IPS, even if encapsulated in it is a second message which should be flagged by the IDS/IPS.

As the CANCAN method is currently unknown, implementations of IDS/IPS systems will not look for this kind of attack, and therefore will not stop it. A suggestion for the development of future IDS/IPS systems is discussed later, in the Protection section.

6.2 Security Gateway (GW) Routing

In many architectures used in the vehicle industry, there are multiple physical busses used in a single vehicle. When a component connected to one bus wishes to pass on a message to another bus, it will do so by sending the message to a gateway component (GW) connected to both busses, which will route the message to the second bus. The GW doubles as a security mechanism, as it can decide which messages to route based on parameters of the incoming message. These parameters may include simple ones such as CANID and length, but it may potentially use more sophisticated methods including deep packet inspection.

Assuming a GW that will base most of its filtering on the CANID of the message, one can picture the following scenario. An attacker connected to one bus, wishes to attack a component connected to another bus by sending a specific CAN message, with a specific CANID (denoted as *MALICIOUS_CANID*). However, the GW is configured to allow routing of only a specific CANID (denoted as *LEGAL_CANID)*.

Using CANCAN, the attacker may encapsulate the message with a CANID equal to *MALICIOUS_CANID* inside a second message with a CANID equal to *LEGAL_CANID*. The GW will forward the message to the bus connected to the component, and so (based on the specific attack scenario), the component will be attacked.

7 ATTACK SCENARIOS

The CANBUS and CAN-FD protocols were built to withstand bus errors. This resistance to errors, includes the objective of never allowing one message to appear as another – even in the case of a reasonable number of errors on the bus.

This is the main reason for using a CRC as part of the protocol, and the reason for the protocol changes made to the original CAN-FD specification (now denoted as non-ISO CAN-FD) when creating the current CAN-FD specification (the ISO CAN-FD) [8].

The CANCAN attack may break this assumption in case of a message parsing that starts in the middle of a sent CAN-FD message. This may happen in several possible scenarios.

7.1 Bus Error

Bus errors may cover a wide range of cases, depending on the cause of noise on the used communication bus. These errors may include:

- Bit flips – a result of a noisy bus, resulting in sampling of an incorrect value.
- Bit double sample – a result of a clock shift causing two sample points to occur during a single transmitted bit.
- Bit skipped sample – a result of a clock shift causing two consecutive sample points to occur before and after a transmitted bit, thus skipping it.

Any such error may cause a controller to assume the current sent CAN-FD message has finished sending, and the next message may begin parsing.

One example of such an error is a bit flip of the most significant bit of the DLC field (shown in Figure II). This will cause a DLC field of "1111" (denoting a 64-byte message) to appear as "0111" (denoting a 7-byte message).

Therefore, such an error would cause the message parsing to end earlier (in the middle of the DATA field). If an encapsulated message, prepended by 10 recessive bits (7 bits of End of Frame, and 3 bits of Intermission) was parsed starting at the end of the incorrectly read previous message, an encapsulated message will be read, and appear to be valid.

An attacker attempting to use the Bus Error scenario will likely attempt to send multiple encapsulating messages to the target, in the hopes that a specific error will occur, and the encapsulated message will be received.

7.2 *Wakeup*

When a CAN-FD controller wakes up, it has no knowledge regarding the current state of the bus on which it will start listening. It will start in operation mode "Integrating" in which it will [6, pp. 21-22] look for the "IDLE Bus sequence" – eleven consecutive recessive bits. After such a sequence, the operation mode will change to "Idle". In the "Idle" mode, the controller will be ready to receive the START OF FRAME (SOF), and switch to "Receiver" mode.

Therefore, if after waking up, the first IDLE Bus sequence is encountered as part of the encapsulating message bitstream (sampled at a slow bitrate), an encapsulated message following this sequence may be read, and seem to be valid.

An attacker attempting to use the Wakeup scenario will likely attempt to send multiple encapsulating messages to the target unit when it is expected to reset. Alternatively, if the attacker has the capability of causing a unit to reset (either directly, or by using another vulnerability to cause a crash resulting in a reset), multiple encapsulating messages may be sent right after the reset inducing message.

In both cases, the attack will be successful only statistically, and will therefore require multiple resets of the target unit.

7.3 *Mixed Network*

By default, CAN and CAN-FD nodes cannot operate on the same network. If naively connected, the CAN nodes will recognize the FD messages as errors, and therefore will not operate correctly.

Several solutions for using a mixed network were proposed (as can be seen in [9]). Such solutions may be vulnerable for the CANCAN attack, as they include CAN nodes which will only use the slow bitrate sampling. The specific way to utilize such an attack will depend on the specific mixed network solution used.

As the specific method used may vary, this paper will not further explore the exploitation of this scenario.

8 BASE ENCAPSULATION

The most obvious way of utilizing such an encapsulation, is through usage of the DATA field of a CAN-FD message with a set Bitrate Switch flag (BRS). Such an encapsulation will need to address several issues:

- Fast Bitrate Encoding – Both CAN-FD and CANBUS messages make use of bits sent at a slow bitrate (while a CAN-FD message allows sending the DATA and CRC fields using a faster bitrate, it will still use the slow bitrate for the rest of the message). Given that these slow bits will be sampled from the DATA field of a CAN-FD message, some method of encoding slow bits using fast bits should be used.
- Fast Bitrate Copying – When attempting to encapsulate a CAN-FD message with a set Bitrate Switch flag (BRS), the DATA and CRC fields of the encapsulated message will need to be sent using the fast bitrate. Therefore, these bits can be copied "as is" into the encapsulating message data.

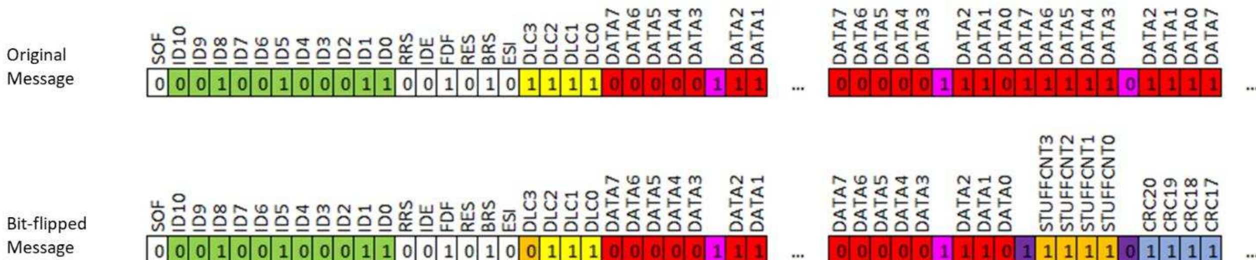

FIGURE II
BIT-FLIP OF DLC CAUSING SMALLER MESSAGE

Both first and second issues will require the ability to encode slow bitrate bits using the fast bits of the CAN-FD DATA field. Before proposing an encoding method, the Stuffing mechanism will be reviewed.

8.1 Stuffing

To make sure that the transmitter and receivers remain synchronized, it is crucial that an edge will be sent on the bus allowing a resynchronization of the controllers. Therefore, the CANBUS and CAN-FD protocols do not allow a sequence of more than 5 consecutive same value bits. This means, that after at most 10 bits (e.g., “0000011111”), a controller will encounter a recessive to dominant (1->0) edge, required for resynchronization.

To do this, every sequence of 5 consecutive same value bits will have a bit of the opposite value appended to it. For example, the sequence “000000” will be converted to the sequence “00000**1**0”. This process is called “Stuffing”.

It should be noted that stuffing will also consider the previously stuffed bits as part of the 5 consecutive bits sequence. This will allow what is called a “cascade effect”, in which a sequence of bits of the shape “0000011110000111100001111...” will be stuffed to “00000**1**1111**0**0000**1**1111**0**0000**1**1111…”. In the extreme case, the ratio of stuff bits to data bits will approach ¼.

A different type of Stuffing mechanism is the Fixed-Stuff-Bits (FSB). This type is used in the CRC field of FD messages. Every 5th bit (starting with and including the first bit) in the CRC field will be an FSB bit, with a value opposite the one preceding it. These bits are in constant offsets relative to the start of the CRC field and replace the usage of regular Stuffing inside the CRC field. As these bits are not part of the DATA field, they do not need special handling for Base Encapsulation, but will need handling in other types of encapsulations.

8.2 Fast Bitrate Encoding

The most obvious way to encode a slow bit, is by repeating a bit sent in the fast bitrate multiple times. The number of repetitions will be equal to the ratio of the fast and slow bitrates. So, for a ratio of 4, the translation table shown in Table I may be used.

TABLE I
NAÏVE TRANSLATION TABLE

{slow}”0”	{slow}”1”
{fast}”0000”	{fast}”1111”

This presents a problem – the sequence {slow}”00” will be encoded as {fast}”00000000”. However, the stuffing mechanism cannot allow this sequence of raw bits to be sent. Attempting to send {fast}”00000000” will be converted to the sequence {fast}”00000**1**000”.

It is important to note the distinction between the “data” of the encapsulating CAN-FD BRS message (the information that is sent, always a whole number of bytes), and the DATA field (containing actual raw bits sent on the bus, including “stuffed” bits). The proposed encoding should only consider the actual raw bits sent on the bus – not the “data” of the encapsulating message. However, the “data” will be used to build and send the encapsulating message, and so the raw bits and bytes must be converted back to “data”.

One way of encoding without encountering the “Stuffing” problem, is to encode the bit differently depending on whether this is the first bit in a sequence of consecutive same slow bits, or not. An example of such a translation table can be seen in Table II.

TABLE II
NO STUFFING TRANSLATION TABLE

	{slow}”0”	{slow}”1”
1st time	{fast}”0000”	{fast}”1111”
else	{fast}”1000”	{fast}”0111”

It is clear that this encoding method will not encounter the same stuffing problem, as no slow sequence can be encoded in a way that will include 6 consecutive same value bits.

As the slow bit will be sampled near the end of the sequence (during the 3rd or 4th fast bit), the sampled value will correspond to the slow bit value.

Introducing the new edges (the ‘0’ at the start of a slow ‘1’, or ‘1’ at the start of a slow ‘0’) adds a risk of causing a clock resynchronization, which might cause the sampling to go out of sync.

According to the synchronization rules [6, pp. 31-32], only recessive to dominant edges (1->0) may cause resynchronization. Moreover, an edge will be used in synchronization only if the value detected in the previous sample point differs from the bus value immediately after the edge.

In the non-first {slow}”1” case, the edge 1->0 adheres to these rules and thus causes a synchronization. But, as this edge happens exactly between two slow bits, this point in time is already the clock edge and no clock drift will occur.

In the non-first {slow}”0” case, the edge 1->0 will fail the second rule. Because the value after the edge is the same as the previous sampled value (notice that this is not the first slow 0 in a sequence, and therefore the previous sampled value was 0), no resynchronization will be performed.

Therefore, the proposed encoding will not cause any clock drift, and the slow bits can successfully be encoded using fast bits.

8.3 Ack Field

When encoding the encapsulated message, the ACK bit of the encapsulated message should receive special consideration. When a normal message is sent, the ACK bit will be sent with recessive bit, so that a receiving controller may mark the message as successfully received by pushing a dominant value into it.

However, in the case of an encapsulated message, its ACK bit resides in the DATA section of the sent FD message. If the value of the ACK bit in the encapsulated message is sent

as a recessive bit, and a receiver would push a dominant value while it is sent, the transmitter might notice that another unit is sending data at the same time as itself. This may be recognized as a bus error, causing the transmitter to stop sending the rest of the message.

At this stage, the encapsulated message may have already been successfully received – but the bus error may also cause it to be dismissed.

Therefore, it is better to set the ACK bit to dominant in the encapsulated message, circumventing this issue and allowing the transmitter to continue sending the encapsulating message successfully.

When attempted using the Wakeup Attack Setup (presented in 11.2), only encapsulated messages with a dominant ACK bit were accepted.

8.4 Encapsulation Steps

Considering all the mentioned issues, an encapsulation may be done following these steps:

Algorithm I – Base Encapsulation		
Input	:	• Encapsulated message • CANID of the encapsulating message • Length of the encapsulating message
Output	:	• FD Encapsulating message
Step 1	:	Create the FD encapsulating message with the input arguments (CANID and length). Its data should be the conversion of the encapsulated message bits (prepended by Quiet Bits) into fast bits using Fast Bitrate Encoding (8.2) on its slow bits. If needed, append or prepend padding for reaching the length of the encapsulating message.

8.5 Limits

The proposed Base Encapsulation attack method may be limited to the encapsulation of only specific messages. The calculation of these limits will depend on several factors – FD vs. CANBUS, extended CANID vs. non-extended CANID, and for FD messages, CRC17 vs. CRC21. The number of bits required for different cases can be seen in Table III.

Note that the presented table will assume a preamble of 11 bits, corresponding to the Idle bits of the Wakeup scenario (7.2). Calculation for the case of the Bus Error scenario (7.1) will require using the value of 10 bits instead.

The stuffing may add to the number of bits used. This change can be anywhere from 0 bits (in case no stuffing is needed) and up to 1 extra bit for every 4 bits (in the cascade case, see 8.1), causing the number of bits to be multiplied by the ratio 5/4.

The calculated theoretical length (depending on data length and stuff ratio) will be compared to the maximum available bits in the CAN-FD data section.

Note: The calculation will assume that a maximum of 64 bytes (64*8 bits) may be used in the data section of the encapsulating message. Theoretically, more bits can be used for this encoding, as the encapsulating message may also make use of stuff bits. Depending on the fast to slow bitrate ratio, they can sometimes be utilized. For example, if the fast to slow bitrate ratio was 5, encoding {slow}"01" as {fast}"0000011111" would utilize the stuff bit. However, for a fast to slow bitrate ratio of 4, no consistent way of utilizing these stuff bits was found.

TABLE III
MESSAGE TYPES TO CONTENT BIT LENGTH

Type	Content	#Bits (without DATA)
Base Can (Slow, need stuffing)	PREAMBLE+SOF+CANID+ +RTR+IDE+FDF+DLC+ +DATA+CRC	45
Base Can (Slow, no stuffing)	CRC_DEL+ACK+ACK_DEL+ EOF	10
Extended Can (Slow, need stuffing)	PREAMBLE+SOF+CANID1+ +SRR+IDE+CANID2+RTR+ +FDF+r0+DLC+DATA+CRC	65
Extended Can (Slow, no stuffing)	CRC_DEL+ACK+ACK_DEL+ EOF	10
Base FD (Slow, need stuffing)	PREAMBLE+SOF+CANID+ +RRS+IDE+FDF+res+BRS	28
Base FD (Slow, no stuffing)	ACK+ACK_DEL+EOF	9
Base FD (Fast, need stuffing)	ESI+DLC+DATA	5
Base FD (CRC17) (Fast, need FSB)	STUFF_CNT+CRC+CRC_DEL	22
Base FD (CRC21) (Fast, need FSB)	STUFF_CNT+CRC+CRC_DEL	26
Extended FD (Slow, need stuffing)	PREAMBLE+SOF+CANID1+ +SRR+IDE+CANID2+RRS+ +FDF+res+BRS	47
Extended FD (Slow, no stuffing)	ACK+ACK_DEL+EOF	3
Extended FD (Fast, need stuffing)	ESI+DLC+DATA	5
Extended FD (CRC17) (Fast, need FSB)	STUFF_CNT+CRC+CRC_DEL	22
Extended FD (CRC21) (Fast, need FSB)	STUFF_CNT+CRC+CRC_DEL	26

For every encapsulated message type, the number of fast bits required is calculated, using the following calculation:

$$L = (ss \cdot SR + sn) \cdot FSR + fs \cdot SR + [ff \cdot FR] \quad (1)$$

In this equation:

- L (length) – the encapsulating message's total data length. This value will be taken as 64*8, which is the maximum number of bits available for usage in the encapsulating message.
- ss (slow stuffing) – the number of slow bits for which stuffing is needed.
- SR (stuffing ratio) – the ratio between the number of bits needed for representing the data with stuffing bits and without them. This ratio will be examined in its two extremes, 1 and 5/4.
- FR (FSB stuffing ratio) – the ratio between the number of bits needed for representing the data with FSB bits and

without them. Its value is equal to 5/4. The result of the multiplication $ff{\cdot}FR$ will be rounded up.

- sn (slow no-stuffing) – The number of slow bits for which stuffing is not needed.
- FSR (fast to slow ratio) – The ratio of the fast and slow bitrates. It will be taken as 4.
- fs (fast stuffing) – the number of fast bits for which stuffing is needed.
- ff (fast FSB) – the number of fast bits for which FSB stuffing is needed.

This equation will be extended with a variable x denoting the variable length of the encapsulated message data. In case of an encapsulated CANBUS message, it will be inserted into the equation the same as the ss variable.

$$L = ((ss+x){\cdot}SR + sn){\cdot}FSR+fs{\cdot}SR+\lceil ff{\cdot}FR\rceil \qquad (2)$$

The equation used to calculate the maximum number of encoded bits possible in case of an encapsulated CANBUS message may be calculated by extracting the x variable, as seen in (3).

$$x = (L - fs{\cdot}SR-\lceil ff{\cdot}FR\rceil)/(FSR{\cdot}SR) - ss - sn/SR \qquad (3)$$

In case of an encapsulated FD message, x will be inserted into the equation the same as the fs variable.

$$L = (ss{\cdot}SR + sn){\cdot}FSR+(fs+x){\cdot}SR+\lceil ff{\cdot}FR\rceil \qquad (4)$$

The equation used to calculate the maximum number of encoded bits possible in case of an encapsulated FD message may be calculated by extracting the x variable, as seen in (5).

$$x = (L-(ss{\cdot}SR + sn){\cdot}FSR-\lceil ff{\cdot}FR\rceil)/SR - fs \qquad (5)$$

Using these equations, it is possible to extract the maximum number of data bits in the encapsulated message, using the two extreme values of SR. The values representing the maximum number of data bytes (for the cases of minimum and maximum stuffing) can be seen in Table IV.

TABLE IV
MESSAGE TYPES TO POSSIBLE ENCAPSULATED DATA LENGTH (BASE ENCAPSULATION)

Type	Max Stuff #Data bytes	Min Stuff #Data bytes
Base Can	6	8
Extended Can	3	6
Base FD (CRC17)	16	16
Base FD (CRC21)	24	32
Extended FD (CRC17)	16	16
Extended FD (CRC21)	20	32

It should be noted that the supported number of bytes is rounded down to the closest message length supported by the FD protocol (which include the lengths 0-8, 12, 16, 20, 24, 32, 48, and 64). The rounding should be down (as opposed to rounding up), because rounding up assumes extra bits may be added and used. As this is the maximum number of bits that may be represented, adding bits to a multiple of 8 is not possible.

As can be seen, not all possible messages can be encapsulated successfully. Some message lengths are impossible to send (using Base Encapsulation), while others depend on the specific data being sent.

9 END ENCAPSULATION

One way of better utilizing the encapsulating message is using not only its DATA FIELD, but also its later fields, including the CRC and ACK fields.

This will be quite different for the encapsulation of CAN-FD vs. Regular CAN-BUS messages.

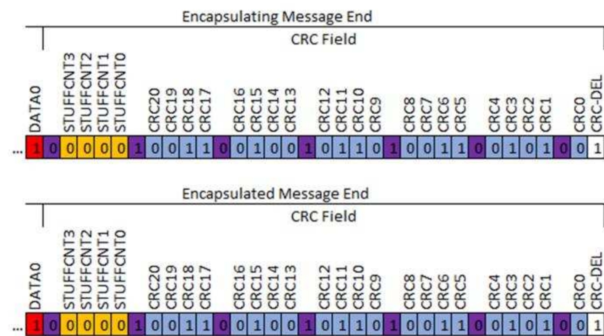

FIGURE III
CORRESPONDING END BITS, ENCAPSULATED FD WITH SAME CRC

9.1 *End Encapsulation of FD – Same CRC Length*

CAN-FD supports two types of CRC calculation, depending on the length of data. For data length up to and including 16 bytes a CRC of 17 bits is used, and for data length of 32 to 64 bytes a CRC of 21 bits is used. In the simplest case, it may be assumed that the encapsulating and encapsulated message use the same CRC length. This is a relatively simple case, as the last bits and fields of the encapsulated message perfectly align to the same bits in the encapsulating message (as shown in Figure III).

There are several issues that need addressing for successfully encapsulating such a message:

- Verifying a correct CRC value in the encapsulating message.
- Verifying a correct STUFF_CNT value in the encapsulating message.

These two issues will be addressed in detail, and then a step-by-step solution for generally creating the encapsulating message for a specific encapsulated message will be presented.

9.1.1 *Stuff Fixing*

The STUFF_CNT field will count the number of used stuff bits in the message, modulated by 8, and grey encoded. As this field is part of the CRC FIELD it must also be equal between the encapsulated and encapsulating messages. As all

the stuff bits from the encapsulated message also exist in the encapsulating message, the number of additional stuff bits must be a multiple of 8, so that this field will not change.

The most efficient way of adding stuffing is to use the “cascade effect” (see 8.1). This offers the highest ratio of stuff bits to data bits possible – a ratio approaching 1/4.

9.1.2 CRC Fixing

When attempting to encapsulate a CAN-FD BRS message with the same CRC-length as the encapsulating CAN-FD BRS message, the entire CRC and ACK fields of the encapsulating and encapsulated messages perfectly overlap.

Therefore, the CRC value of the encapsulating message will need to be equal to the CRC value of the encapsulated message.

As CRC is a linear calculation, it is possible to calculate what bits need to be changed to get a specific CRC value. Assuming there are some reserved (currently unused) bits in the DATA or even the CONTROL FIELD of the encapsulating message, it is possible to set (or clear) them to fix the CRC value. The number of needed bits may be up to the number of bits in the CRC polynom itself. The offset of the bits in the encapsulating message will need to span the entire vector space of the specific polynom. The details and implementation of the CRC fixing calculations may be found in papers (such as [10]), and open-source tools (such as [11]).

To fix the CRC, the proposed solution will first reserve bits in specific offsets of the message, and will temporarily assume they are clear. At a later stage, after setting the rest of the encapsulating message, these bits may be set in order to change the CRC value to the specific required value.

The most straightforward way of reserving bits is using CRC_LEN consecutive bits. A possible problem with this method, is that it may introduce extra stuff bits. If after doing the reverse CRC calculation these bits will be found to contain the sequence 00000 or 11111, they will change the value of the STUFF_CNT Value. Because fixing the CRC is done as the last stage, this will make the message format illegal.

Another way of doing this is to use non-consecutive bits. This may be more wasteful regarding bits in two ways:

- It may require adding bits to break potential sequences. For example, instead of using six consecutive bits xxxxxx, it is possible to use the sequence xxx01xxx in which it is not possible to cause stuffing – but the price is the usage of extra overhead bits.
- CRC_LEN bits may not be enough to span the entire vector space of the CRC Polynom. In other words, picking CRC_LEN non-consecutive bits, may never (even for all combinations of 1s and 0s) return a specific CRC value. Fixing this may require picking the specific used offsets in a way that does span the entire space, or adding extra offsets until the entire vector space is spanned.

A naïve and slightly wasteful approach is the following:

CRC_BITS = xxx01xxx01xxx01xxx...

Where x denotes the bits used for CRC fixing. By using a few more bits than CRC_LEN (e.g., CRC_LEN+2) one can be sure that the entire vector space is spanned.

9.1.3 Encapsulation Steps

The encapsulation will now make use of 3 extra steps, executed before the step of creating the final encapsulating message. These steps are needed for extracting and fixing the CRC and STUFF_CNT values. The encapsulation steps will now be:

Algorithm II – Same CRC Length FD End Encapsulation		
Input	:	• Encapsulated FD message • CANID of the encapsulating message • Length of the encapsulating message
Output	:	• FD Encapsulating message
Step 1	:	From the encapsulated message, extract the CRC and STUFF_CNT fields. These will be the expected CRC and STUFF_CNT values of the encapsulating message.
Step 2	:	Create an FD message with the same parameters as the encapsulating message (CANID and length), and with data that does not cause stuffing. Extract its STUFF_CNT field and calculate the number of stuff bits required to reach the expected STUFF_CNT value.
Step 3	:	Create an FD message with the same parameters as the encapsulating message (CANID and length). Its data should be the unstuffed conversion of the encapsulated message bits (up to and not including the CRC field, and prepended by Quiet Bits) into fast bits using Fast Bitrate Encoding (8.2) on its slow bits. Prepend bits using the cascade affect (9.1.1) adding enough stuff bits (as calculated in step 2) to fix the STUFF_CNT value. Prepend reserved bits for CRC fixing (9.1.2). If needed, prepend padding for reaching the length of the encapsulating message. From the created message, extract the CRC value, to use when later fixing the CRC.
Step 4	:	Create the FD encapsulating message, using the same data as the last created FD message, but replacing the reserved bits in order to set the CRC to its expected value extracted in Step 1.

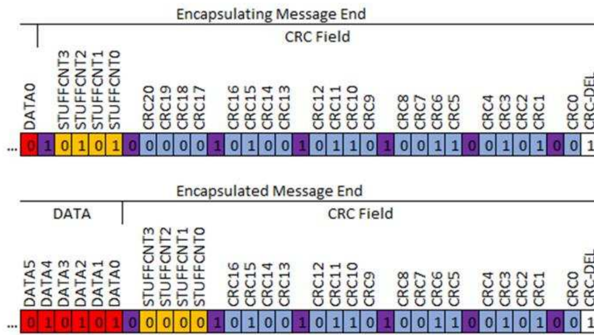

FIGURE IV
CORRESPONDING END BITS, ENCAPSULATED FD WITH SHORT CRC

9.2 *End Encapsulation of FD – Short CRC Length*

For an encapsulated message that requires a shorter CRC, the CRC fields will no longer be aligned. As the encapsulating message CRC is 4 bits longer, its extra 4 bits of CRC coincide with the encapsulated message STUFF_CNT field, and its STUFF_CNT field coincide with the last bits of the data (as shown in Figure IV). This will introduce new issues:

- The CRC and STUFF_CNT values of the encapsulating message will differ from the CRC and STUFF_CNT values of the encapsulated message.
- The end of the DATA field in the encapsulated and encapsulating messages, are no longer synchronized. This may cause data validity issues.

Addressing the first issue is quite simple. As shown in 9.1.2, it is possible to fix the CRC of the encapsulating message by introducing new CRC_BITS bits. The method for fixing the CRC does not change.

However, the data end validity issue cannot be easily addressed, as discussed in the next topic.

9.2.1 *DATA End Validity*

As the length of the CRC Field is different between the encapsulating and encapsulated messages, the DATA field end of the two messages no longer aligns.

The first Fixed Stuff Bit (FSB) right before the STUFF_CNT field in the encapsulating message now corresponds to a data bit in the encapsulated message. The STUFF_CNT itself will also correspond to the data bits of the encapsulated message. This may cause the encapsulating message to appear corrupt in several ways:

- The message is corrupt if the FSB is equal to the bit before it. As a stuff bit, it must be of the opposite value.
- The message is corrupt if the last value in the DATA field of the encapsulating message appears to be a stuff bit (which is not FSB). For example, the DATA field must not end with the sequence "000001", because the next bit (which is FSB) should already solve the stuffing. This means that if DATA5 (the sixth bit from the end) of the encapsulated message would be a stuff bit, the encapsulating message will be corrupt.
- The message is corrupt if the checksum value of STUFF_CNT is invalid. The STUFF_CNT's last bit is set in a way, that the checksum over the entire STUFF_CNT must be equal to zero.

This issue of the DATA end validity cannot be generally solved, so only a subset of encapsulated messages which do not cause this issue can be supported.

As this issue is caused by specific bits (DATA4, DATA5, and the sum of DATA3-DATA0 the encapsulated message), if the encapsulated message has freedom in choosing these bits, it can be easily fixed.

9.2.2 *Encapsulation Steps*

The encapsulation steps required are almost identical to the ones described for Same CRC Length encapsulation (9.1.3). The only difference is the addition to Step 1, verifying the validity of the encapsulating message DATA end. The complete encapsulation sequence will now be the following:

Algorithm III – Short CRC Length FD End Encapsulation		
Input	:	• Encapsulated FD message • CANID of the encapsulating message • Length of the encapsulating message
Output	:	• FD Encapsulating message
Step 1	:	Viewing the end of the encapsulated message as the fast bits of the encapsulating message, extract the expected CRC and STUFF_CNT fields. Verify that the encapsulating data end is valid (9.2.1). If the data end is invalid, retry with an encapsulated message with a different data end.
Step 2	:	Create an FD message with the same parameters as the encapsulating message (CANID and length), and with data that does not cause stuffing. Extract its STUFF_CNT field and calculate the number of stuff bits required to reach the expected STUFF_CNT value.
Step 3	:	Create an FD message with the same parameters as the encapsulating message (CANID and length). Its data should be the unstuffed conversion of the encapsulated message bits (up to and not including the end bits already encoded in Step 1, and prepended by Quiet Bits) into fast bits using Fast Bitrate Encoding (8.2) on its slow bits. Prepend bits using the cascade affect (9.1.1) adding enough stuff bits (as calculated in step 2) to fix the STUFF_CNT value. Prepend reserved bits for CRC fixing (9.1.2). If needed, prepend padding for reaching the length of the encapsulating message. From the created message, extract the CRC value, to use when later fixing the CRC.

Step 4 : Create the FD encapsulating message, using the same data as the last created FD message, but replacing the reserved bits in order to set the CRC to its expected value extracted in Step 1.

9.3 *End Encapsulation of CAN-BUS*

Encapsulating a CAN-BUS message as the end of a legal CAN-FD BRS message, introduces many problems. These are caused by the vastly different fields expected in the CRC fields, and the fact that they are sent using a different bitrate.

Such an encapsulation will need to address similar issues to those presented for the case of Short CRC Length CAN-FD encapsulation (9.2):

- The CRC and STUFF_CNT of the encapsulating message will differ from the encapsulating one.
- The FSBs and STUFF_CNT of the encapsulating message correspond to slow bits from the encapsulating message CRC bits, potentially causing a message to appear illegal.

As seen in Figure V, the FSBs and STUFF_CNT of the encapsulating message now correspond to the slow bits of the CRC and CRC-DEL values of the encapsulating message. Therefore, using the previously described Fast Bitrate Encoding (described in 8.2) will always fail in consistently creating valid FSBs. Another encoding must be used, assuring that FSBs will always contain a value opposite the one before them, and that the STUFF_CNT will always have a correct checksum.

9.3.1 *End Encoding*

The FSB constraint can be solved by creating a separate translation between slow and fast bits, depending on the offset of the FSB inside a 4-bits window. The translation may also depend on the specific slow bit offset for which the translation is done. For example, the translation of CRC5 and CRC0 of the encapsulated message both have the same FSB offset (offset 0), but they may be encoded differently if needed.

There are a couple of things to notice when building such a translation table:

- The usage of specific translation values may cause resynchronizations (in case of 1->0 edges before/after their expected time). This must be considered when verifying that the slow sampling of the values corresponds to the slow bitrate sequence of the encapsulated message. In case of resynchronization while encoding a specific slow bit, the next slow bit or bits may need to be encoded using a larger or smaller number of fast bits to fix the clock shift.
- As the STUFF_CNT field contains one parity bit, it must be taken into account when creating the translation of the slow bit corresponding to it. In the case of the proposed translation table, the optional values of STUFF_CNT are either “0000” or “1111”, which both have a valid parity bit.

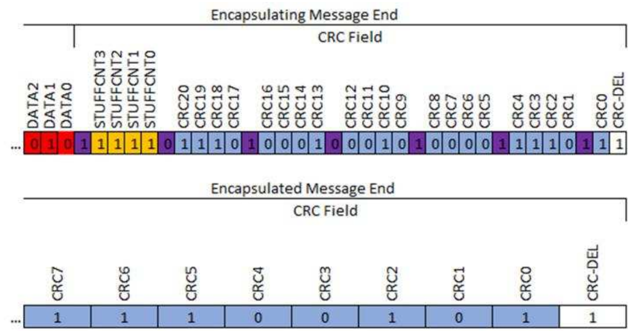

FIGURE V
CORRESPONDING END BITS, ENCAPSULATED CAN-BUS MESSAGE

An example of such a translation table (assuming the encapsulating message has a 21bit CRC) can be seen in Table V. In this table, *x* denotes the value equal to the opposite value of the previous fast bit. It is also assumed that the slow bitrate sampler will always sample the 4th fast bit.

TABLE V
END ENCAPSULATION OF CAN-BUS TRANSLATION TABLE
(ENCAPSULATING USING CRC21)

FSB offset	{slow}"0"	{slow}"1"
None	{fast}"0000"	{fast}"1111"
0	{fast}"x000"	{fast}"x111"
1	{fast}"0100"	{fast}"0111"
2	{fast}"0100"	{fast}"0011"
3	{fast}"0110"	{fast}"0101"

Using this translation table will not add any resynchronization causing a shift of the clock. This is due to the following resynchronization rules:

- Only 1->0 edges are used for resynchronization. Therefore “0000” followed by “0100” will not cause a resynchronization on the 0->1 edge.
- The value after the edge must be different than the previously sampled value for resynchronization to occur. Therefore “0000” followed by “1000” will not cause a resynchronization.
- Only one resynchronization may occur between two sample points. Therefore “0111” followed by “0100” will only resynchronize once, on the first bit of the second sequence. As this is already the expected edge location, this will not cause any clock shift.

Using this translation table, the expected values for the CRC and STUFF_CNT of the encapsulating message may be extracted. As done in the previous encapsulation algorithms described before, the rest of the data in the encapsulating message will be set so that the calculated CRC and STUFF_CNT are equal to the values extracted in this stage.

Notice that such a translation will automatically handle the FSB and STUFF_CNT validity issue, as the fast bit conversion sets all FSBs and STUFF_CNT to valid values.

Note: The proposed table is only an example of a translation table and is specifically intended for an encapsulating message using CRC21. For a shorter encapsulating message using CRC17, a different translation table may be needed, and even for encapsulating messages using CRC21 other tables can be found that work just as well.

9.3.2 Encapsulation Steps

The encapsulation steps will now include the end encoding in Step 1, but otherwise be identical. The complete encapsulation sequence will now be the following:

Algorithm IV – CANBUS End Encapsulation

Input	:	• Encapsulated CANBUS message • CANID of the encapsulating message • Length of the encapsulating message
Output	:	• FD Encapsulating message
Step 1	:	Using End Encoding (9.3.1), view the end of the encapsulated message as the fast bits of the encapsulating message. Extract the expected CRC and STUFF_CNT fields from these fast bits.
Step 2	:	Create an FD message with the same parameters as the encapsulating message (CANID and length), and with data that does not cause stuffing. Extract its STUFF_CNT field and calculate the number of stuff bits required to reach the expected STUFF_CNT value.
Step 3	:	Create an FD message with the same parameters as the encapsulating message (CANID and length). Its data should be the unstuffed conversion of the encapsulated message bits (up to and not including the end bits already encoded in Step 1, and prepended by Quiet Bits) into fast bits using Fast Bitrate Encoding (8.2) on its slow bits. Prepend bits using the cascade affect (9.1.1) adding enough stuff bits (as calculated in step 2) to fix the STUFF_CNT value. Prepend reserved bits for CRC fixing (9.1.2). If needed, prepend padding for reaching the length of the encapsulating message. From the created message, extract the CRC value, to use when later fixing the CRC.
Step 4	:	Create the FD encapsulating message, using the same data as the last created FD message, but replacing the reserved bits in order to set the CRC to its expected value extracted in Step 1.

9.4 Optimized Encoding

All of the described end encapsulation techniques have two overhead sources:

- CRC correction bits
- STUFF_CNT correction bits

The Fast Bitrate Encoding method (8.2) is not very efficient in the utilization of the encoded data bits. In the next section, a new and optimized encoding scheme will be proposed, which will attempt to handle the overhead sources without the addition of extra bits.

9.4.1 Stuff Fix Optimization

The original translation table takes care not to add any stuff bits. Instead, it is possible to create a different translation table, attempting to add as many stuff bits as possible without causing a resynchronization. One such example is shown in Table VI.

TABLE VI
FAST BITRATE ENCODING TRANSLATION TABLE, WITH STUFFING

State	{slow}"0"	{slow}"1"
after 1 {fast}'0'	{fast}"0000"	{fast}"0001"
after 2 {fast}'0'	{fast}"1000"	{fast}"0001"
after 3 {fast}'0'	{fast}"0010"	{fast}"0011"
after 4 {fast}'0'	{fast}"0110"	{fast}"0101"
after 5 {fast}'0'	{fast}"1010"	{fast}"1101"
after 1 {fast}'1'	{fast}"0000"	{fast}"1111"
after 2 {fast}'1'	{fast}"0000"	{fast}"0101"
after 3 {fast}'1'	{fast}"0000"	{fast}"0001"
after 4 {fast}'1'	{fast}"0000"	{fast}"0001"
after 5 {fast}'1'	{fast}"0000"	{fast}"0001"

This translation table may be used for encoding as many bits as needed to fix the number of used stuff bits.

9.4.2 CRC Fix Optimization

Rather than using extra bits as a placeholder for fixing the CRC, it is possible instead to utilize the unused fast bits of the Fast Bitrate Encoding. A new encoding may be proposed, similar to the original Fast Bitrate Encoding, but adding placeholders that can be used for CRC fixing. One such example is shown in Table VII.

TABLE VII
FAST BITRATE ENCODING TRANSLATION TABLE, WITH CRC PLACEHOLDERS

	{slow}"0"	{slow}"1"
1st time	{fast}"0x00"	{fast}"1x11"
else	{fast}"1x00"	{fast}"0x11"

In this table, x denotes a placeholder for future CRC fixing, as done in 9.1.2. Regardless of the value x will be in the end, this encoding does not add any resynchronization edges, and therefore will be parsed by a slow bitrate sampler as intended.

The final optimized encoding will use a mix of the two proposed translation tables. For the encoding of the first bits, the stuff fixing translation table will be used (9.4.1). Then, for the remaining bits, the CRC fixing table will be used.

In the following encapsulation steps, it is assumed that this is the way this encoding will be used.

9.4.3 Encapsulation Steps

The encapsulation steps will now change Step 3, removing the additional two bit-sequences (previously used for STUFF_CNT and CRC fixing), and replacing them with the usage of the Optimized Encoding. The following steps may be used for all three types of end encapsulation:

Algorithm V – Optimized Encoding End Encapsulation		
Input	:	• Encapsulated message • CANID of the encapsulating message • Size of the encapsulating message
Output	:	• FD Encapsulating message
Step 1	:	Viewing the end of the encapsulated message as the fast bits of the encapsulating message, extract the expected CRC and STUFF_CNT fields. Use End Encoding (9.3.1) in case the message is CAN. Verify that the data end is valid (9.2.1) in case of short CRC length FD.
Step 2	:	Create an FD message with the same parameters as the encapsulating message (CANID and length), and with data that does not cause stuffing. Extract its STUFF_CNT field and calculate the number of stuff bits required to reach the expected STUFF_CNT value.
Step 3	:	Create an FD message with the same parameters as the encapsulating message (CANID and length). Its data should be the conversion of the encapsulated message (prepended by Quiet Bits, and not including the end bits already encoded in Step 1) into fast bits. Use the Optimized Encoding (9.4) to both fix the STUFF_CNT value, and to reserve bits for CRC fixing. If needed, prepend padding for reaching the length of the encapsulating message.
Step 4	:	Create the FD encapsulating message, using the same data as the last created FD message, but replacing the reserved bits in order to set the CRC to its expected value extracted in Step 1.

9.5 Limits

Introducing the new Optimized Encoding will change the limits of the possible encapsulated messages. The calculations will be very similar to what was done in 8.5. However, as more than just the DATA field of the encapsulating message is now utilized, the only change needed is to the value of *L*. Its value will be increased by the number of extra fast bits from the CRC field (STUFF_CNT+CRC21+CRC_DEL+FSBs=33) and the number of slow bits in the end of the encapsulating message (ACK+ACK_DEL+EOF=9) multiplied by the fast to slow ratio (4). The new value of *L* will be 581 fast bits. As can be seen in Table VIII, the range of the now supported encapsulated messages is wider.

10 Multiple Encapsulation

It may be possible to encapsulate multiple messages in a single encapsulating message. Doing so is quite straightforward, and can make use of multiple Base Encapsulations, or using both Base and End encapsulations in tandem.

TABLE VIII
MESSAGE TYPES TO POSSIBLE ENCAPSULATED DATA LENGTH (END ENCAPSULATION)

Type	Max Stuff #Data bytes	Min Stuff #Data bytes
Base Can	8	8
Extended Can	5	8
Base FD (CRC17)	16	16
Base FD (CRC21)	32	48
Extended FD (CRC17)	16	16
Extended FD (CRC21)	24	32

One change introduced by multiple encapsulated messages as opposed to the encapsulation of a single message, is the number of prepended recessive bits required for the latter messages. As mentioned in the Bus Error scenario (7.1), two received messages are expected to have 10 recessive bits separating them – 7 End of Frame (EOF) bits, and 3 Intermission bits. So, in case of the Wakeup scenario (7.2), only the first encapsulated message will require 11 recessive bits, while the rest will require only 10 bits. In case of the Bus Error scenario, 10 bits will be used as the prepended recessive bits for all encapsulated messages.

11 Attack Setup

Two attack setups were used during the research, simulating two of the three attack scenarios mentioned in section 7.

11.1 Bus Error

This setup included 2 entities:

- *Attacker* – an embedded component, implementing the sending of CAN-FD messages by directly setting the output of a digital bit to the sent message bits (“bit-banging”).
- *Victim* – a CAN-FD controller, listening on the bus. This was done with both PCAN-USB Pro FD and several real controllers, manufactured by some of the top automotive chip vendors, including NXP, Renesas and ST.

The *Attacker* was programmed to send a specially built encapsulating message, which includes an encapsulated message that induces some response from the *Victim*.

In order to simulate a bit-flip, one of the DLC bits of the encapsulating message was flipped, and then sent on the bus. The attack was deemed successful if the *Victim* was seen to respond to the encapsulated message. This was successfully and deterministically demonstrated on several *Victim* units (detailed above).

The used *Attacker* was a Teensy 4.0 Microcontroller, running Micro-Python with CAN-FD simulating code.

11.2 Wakeup

This setup included 3 separate CAN-FD entities:

- *Attacker* - a CAN-FD controller, sending the attacking message quickly in a loop.
- *Listener* - a CAN-FD controller, listening on the bus.
- *Victim* - a CAN-FD controller, listening on the bus, and being continuously restarted. The used component was a PCAN-USB Pro FD utilizing an FPGA implementation of the CAN FD controller.

The reason a *Listener* is needed (besides being similar to the real-life case of other ECUs existing and listening on the same bus) is to prevent flooding the bus with retries. When the *Victim* is restarting, it will not be able to mark the ACK bit of the sent message, causing the *Attacker* to assume bus errors and retry sending the same message. Adding a *Listener* will solve this issue, as it will always set the ACK bit.

Every time the *Victim* would reset, it would have a statistical probability of starting its sampling during the sending of the encapsulating message. By logging the received messages, it was successfully demonstrated that the *Victim* received the encapsulated message in a certain percentage of the restarts. All the previously mentioned encapsulation techniques were checked and successfully verified using this setup.

The theoretical probability of success P will be equal to:

$$P = AT/TT \quad (6)$$

In this equation:

- P (probability) – the probability of successfully parsing the encapsulated message.
- AT (allowed time) – the time in the sending of a single encapsulating message where if the sampling begins, the encapsulated message will be successfully received. This will include all the time from 10 IDLE bits before the encapsulating message, and up to the encapsulated message IDLE bits. If sampling starts at any point in this time slot, it will wait until the bus seems IDLE, and accept the encapsulated message.
- TT (total time) – the total time of sending a message, including the time between messages.

For example, using end encapsulation for encapsulating an FD message, with CANID 0x123 and the data “AA” in an encapsulating message, with CANID 0x234, resulted in the data

“

7C F9 F3 E7 CF 97 55 11 17 17 17 77 77 77 77 77
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77
77 77 77 77 77 77 77 77 77 77 77 77 77 77 77 77
77 77 77 77 77 77 70 88 F0 8F 08 8F 70 8F 0F 0D

”.

If sent with a 2 ms wait between messages, on a bus with a slow bitrate of 500 kbps, and a fast bitrate of 2 mbps, the equation will predict 11% success. Attempting this with the setup, resulted in a similar value of about 13% success.

12 Protection

12.1 Controller Protections

As mentioned in the System Model and Limitations section of this paper (3), the CANCAN method requires that the controller do both of the following:

- The controller will use the slow bitrate sampling to determine if the bus is idle.
- A synchronization done on a recessive to dominant edge (1->0), which will later sample the bit ‘1’ will not be handled as an error.

A controller challenging these assumptions may be immune to this attack. Specifically:

- The controller will use the fast bitrate sampling to determine if the bus is IDLE, taking into account the fast bitrate to slow bitrate ratio. For example, instead of waiting for a sequence of 11 recessive slow bits, the controller will wait for a sequence of 11*4 recessive fast bits.
- When doing a synchronization on a recessive to dominant edge (1->0), it will be verified that the later sampled bit is ‘0’.

It should be noted that this solution may not be possible for the designer of a system using a specific hardware, as the controller logic may be built into the used system-on-chip and used as-is by the developers.

12.2 Encryption/Signatures

Addition of higher layer protocols, or changes to the CANBUS protocol may be used to add encryption and signatures to the sent messages. Several such methods were proposed, suggesting different encryption and signature protocols to be used. One such solution is the CANcrypt by ESAcademy, presented in [12]. Another solution is the Autosar SecOC protocol, as seen in [13].

As these methods essentially prevent the spoofing of messages, the CANCAN method may be rendered irrelevant. However, if these protection mechanisms are only added to a subset of the messages sent over the bus, the remaining messages may still be susceptible to this attack.

12.3 IDS/IPS

The most obvious way to recognize a potential attack using the CANCAN method is to analyze any sent message, attempting to simulate a slow bitrate controller starting at every sent bit offset. This may seem to require much time and computation efforts, as the complexity of such a task is O(N^2). However, diving deeper into the problem reveals it to be much easier.

Every encapsulated message must begin with an apparently IDLE bus (10 bits in the Bus Error scenario) followed by the Start of Frame (SOF) dominant bit. As the

CAN-BUS message itself cannot contain more than 5 consecutive same bits (which is the objective of the Stuffing mechanism) it is quite difficult to create two valid encapsulated messages with an overlap. For such a case to be valid, the IDLE Bus part (requiring 10 consecutive recessive bits) of one message must overlap with the stuffed bits part of the other message (allowing at most 5 consecutive recessive bits).

Moreover, the usage of resynchronization will tend (over time) to make the sampling points of optional encapsulated messages fall into specific offsets, making two overlapping encapsulated messages even more unlikely.

The problem may be separated into two stages – recognizing potential message SOFs and examining every potential message for inconsistencies.

As time is continuous, it is impossible to check all possibilities of the sampling start time. The proposed algorithm will assume that every fast bit is made of exactly (1 + seg1 + seg2)/*FastToSlowRatio* discrete time slots. As this time quantization is used by the FD protocol itself (when parsing the slow bits), a clock used by the FD controller is not required to handle any events occurring faster than this slot. It therefore makes sense that working under the same assumptions should not raise the probability of a false-positive, or false-negative identification, by much.

12.3.1 IDLE Bus Candidates

When attempting to search for a slow-bit Quiet Bits sequence in the CAN-FD fast-bit sequence, the following algorithm is proposed.

For every sampling start option, a candidate is created. For every candidate, sampling is resumed from the starting point, and resynchronizations are performed when the conditions are correct. The algorithm will attempt to count 10 consecutive recessive bits for every candidate. Any sampling of a dominant bit before this sequence is reached will remove the candidate. Whenever multiple candidates contain the same sampling point, only the candidate that already sampled more recessive bits is saved, and the rest are discarded. When a dominant bit is encountered after 10 consecutive recessive bits, this candidate is marked for further handling. Pseudocode implementing this logic is shown here:

```
1 :  final_candidates_set = empty set;
2 :  idle_candidates_set = empty set;
3 :  candidate_cnts = empty map;
4 :  for (idx, bit) in fast_bit_sequence
5 :  {
6 :    for candidate in idle_candidates_set:
7 :    {
8 :      // If 1->0 edge, and all other resync conditions
9 :      if resync_conditions(candidate, bit):
10:        do_resync(idx, candidate);
11:      if candidate.is_sample_bit(idx):
12:        if bit == '1':
13:          increment candidate_cnts[candidate];
14:        else (bit == '0'):
15:        {
16:          if candidate_cnts[candidate]>=10:
17:            do_hard_sync(idx, candidate)
18:            add candidate to final_candidates_set
19:
20:          remove candidate from idle_candidates_set
21:        }
22:    }
23:    if bit == '1':
24:    {
25:      for every sample point in the current bit:
26:      {
27:        create candidate with sample point
28:        add candidate to idle_candidates_set
29:        candidate_cnts[candidate] = 1
30:      }
31:    }
32:
33:    // If multiple candidates have the same sample point,
34:    // keep only the one with higher counter.
35:    for c1 in idle_candidates_set:
36:      for c2 in idle_candidates_set:
37:      {
38:        if c1==c2:
39:          continue;
40:        if get_sample_point(c1) != get_sample_point(c2):
41:          continue;
42:        if candidate_cnts[c1] < candidate_cnts[c2]:
43:          remove c1 from idle_candidates_set
44:      }
45: }
```

Following this algorithm, the variable *final_candidates_set* will contain a list of edge offsets from which messages may be potentially parsed. This algorithm will go over the message bits one bit at a time, doing O(*NumOfIdleCandidates*) calculations for every bit. Every iteration will keep at most 1 candidate for every sampling point, and every fast bit has '(1 + seg1 + seg2)/*FastToSlowRatio'* points in which sampling may take place. Therefore, the worst-case total timing will be: 'N*(1 + seg1 + seg2)/*FastToSlowRatio*'.

As this may be a bit time consuming, creating candidates with only a subset of the sampling points (line 25 of the pseudocode) may reduce the actual run time (if not the theoretical complexity supremum). In the extreme case, this subset may be reduced to one sample per bit, taken at a specific point – such as the middle of the bit.

12.3.2 CAN-BUS Message Candidates

For every one of the message candidates, the algorithm will simulate the slow bitrate sampling until either the CRC check is passed successfully, or some inconsistency is encountered in the message structure. Such inconsistencies will include:

- A sequence of 6 consecutive identical bits – will not happen in a legal message because stuffing will be added.
- For encapsulated CAN-BUS messages:

  - Incorrect CRC bit.
- For encapsulated CAN-FD messages:
  - DATA ending with stuffing – because STUFF_CNT will start with a FSB, DATA is never supposed to end with such a bit.
  - FSB equal to the previous bit.
  - Incorrect STUFF_CNT bit.
  - Incorrect CRC bit.

It should be noted that for a message candidate that has both FDF (the flag denoting CAN-FD messages) and BRS set, some of the sampling will actually be using a fast bitrate sampler, until after the CRC-DELIMITER field.

This algorithm will go over (at most) all N bits of the message, for every found message candidate, which gives a theoretical complexity of O(N^2) (as a message start edge may start at any fast bit).

But as detailed in the IDS/IPS introduction, it is quite hard to make multiple message candidates overlap. A scanned bit will most likely correspond to only 1-2 potential candidates, as most potential IDLE bus candidates were already dropped in the previous step (12.3.1), either because of an unexpected dominant bit, or because resynchronization caused them to have the same sample point as another candidate.

12.3.3 Messages to Check

As the CANCAN attack is a statistical attack, it is assumed that an attacker will send many crafted messages, hoping that one will be parsed in a malicious way. Following this assumption, it may be more economical to only do a statistical check as well.

An intuitive option for this may be to only examine a message that was received multiple times (with identical CANID and data), and over some threshold of #messages/time. This option may unfortunately be circumvented by an attacker using a collection of many different encapsulating messages, encapsulating the same internal message.

Instead, a statistical approach may be used. Messages will be picked at random times (using a Bernoulli distribution) to be checked. As the messages are picked in a similar way to the attack scenario, it is unlikely to be circumvented by an attacker.

13 CONCLUSION

In this paper, a novel attack circumventing many current CANBUS security measures was proposed. Relevant attack scenarios were presented, and the attack details and optimizations were meticulously explained. Several ways of protecting against the attack were proposed, both in the controller itself, and in external IDS/IPS.

Further research can expand on the possible ways of mounting such an attack based on bus errors and can review the different controllers used in the real world to check their susceptibility to this attack.

14 REFERENCES

[1] M. Bozdal, M. Samie and I. Jennions, "A Survey on CAN Bus Protocol: Attacks,," in *International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, Southend, Essex, UK, 2018.

[2] K. Tindell, "CAN Bus Security - Attacks on CAN bus and their mitigations," 14 Feb 2020. [Online]. Available: https://canislabs.com/wp-content/uploads/2020/12/2020-02-14-White-Paper-CAN-Security.pdf.

[3] K. T. Cho and K. G. Shin, "Error Handling of In-vehicle Networks Makes Them Vulnerable".

[4] K. Tindell, "New CAN Hacks," 20 Jan 2020. [Online]. Available: https://kentindell.github.io/2020/01/20/new-can-hacks/. [Accessed 2 Feb 2022].

[5] K. Tindell, "The Janus Attack," *CAN Newsletter,* pp. 10-11, April 2021.

[6] Bosch, "CAN FD Specification Version 1.0 (released April 17th, 2012)".

[7] K. Tindell, "CAN-HG overview: Augmenting Classic CAN for Performance and Security," 14 Dec 2020. [Online]. Available: https://canislabs.com/wp-content/uploads/2020/12/1905-2020-12-14-CAN-HG-overview.pdf. [Accessed 3 Feb 2022].

[8] A. Mutter, "CAN FD and the CRC issue," *CAN Newsletter,* pp. 4-10, March 2015.

[9] S. Monroe, D. Stout and J. Griffith, "Solutions of CAN and CAN FD in a mixed network topology," in *iCC*, 2013.

[10] M. Stigge, H. Plötz, W. Müller and J.-P. Redlich, "Reversing CRC - Theory and Practice," May 2006. [Online]. Available: https://sar.informatik.hu-berlin.de/research/publications/SAR-PR-2006-05/SAR-PR-2006-05_.pdf.

[11] resilar, "crchack," [Online]. Available: https://github.com/resilar/crchack.

[12] "CANcrypt," ESAcademy, [Online]. Available: https://www.cancrypt.net/. [Accessed 3 Feb 2022].

[13] T. Islinger, Y. Mori, J. Neumüller, M. Prisching and R. Schmidt, "Autosar SecOC for CAN FD," *CAN Newsletter,* pp. 44-45, Jan 2017.

[14] S. F. Lokman, A. T. Othman and M. H. Abu Bakar, "Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review," in *EURASIP*, 2019.