Hardware Module-based Message Authentication in Intra-Vehicle Networks

Eric Wang
Thomas Jefferson High School for Science and Technology
6560 Braddock Road
Alexandria, Virginia 22312
2017ewang@tjhsst.edu

William Xu
Thomas Jefferson High School for Science and Technology
6560 Braddock Road
Alexandria, Virginia 22312
2017wxu@tjhsst.edu

Suhas Sastry
Thomas Jefferson High School for Science and Technology
6560 Braddock Road
Alexandria, Virginia 22312
2017ssastry@tjhsst.edu

Songsong Liu
George Mason University
4400 University Drive
Fairfax, Virginia 22030
sliu23@gmu.edu

Kai Zeng
George Mason University
4400 University Drive
Fairfax, Virginia 22030
kzeng2@gmu.edu

ABSTRACT

The Controller Area Network (CAN) is a widely used industry-standard intra-vehicle broadcast network that connects the Electronic Control Units (ECUs) which control most car systems. The CAN contains substantial vulnerabilities that can be exploited by attackers to gain control of the vehicle, due to its lack of security measures. To prevent an attacker from sending malicious messages through the CAN bus to take over a vehicle, we propose the addition of a secure hardware-based module, or Security ECU (SECU), onto the CAN bus. The SECU can perform key distribution and message verification, as well as corrupting malicious messages before they are fully received by an ECU. Only software modification is needed for existing ECUs, without changing the CAN protocol. This provides backward compatibility with existing CAN systems. Furthermore, we collect 6.673 million CAN bus messages from various cars, and find that the CAN messages collectively have low entropy, with an average of 11.915 bits. This finding motivates our proposal for CAN bus message compression, which allows us to significantly reduce message size to fit the message and its message authentication code (MAC) within one CAN frame, enabling fast authentication. Since ECUs only need to generate the MACs (and not verify them), the delay and computation overhead are also reduced compared to traditional authentication mechanisms. Our authentication mechanism is implemented on a realistic testbed using industry standard MCP2551 CAN transceivers and Raspberry Pi embedded systems. Experimental results demonstrate that our mechanism can achieve real-time message authentication on the CAN bus with minimal latency.

ICCPS 2017, Pittsburgh, PA USA

DOI: http://dx.doi.org/10.1145/3055004.3055016

CCS CONCEPTS

•Security and privacy →Authentication; •Networks →Security protocols; •Computer systems organization →*Embedded systems;*

KEYWORDS

Controller Area Network (CAN); Intra-Vehicle Network; Authentication; Message Compression; Realtime

ACM Reference format:
Eric Wang, William Xu, Suhas Sastry, Songsong Liu, and Kai Zeng. 2017. Hardware Module-based Message Authentication in Intra-Vehicle Networks. In *Proceedings of The 8th ACM/IEEE International Conference on Cyber-Physical Systems, Pittsburgh, PA USA, April 2017 (ICCPS 2017),* 10 pages.
DOI: http://dx.doi.org/10.1145/3055004.3055016

1 INTRODUCTION

The Controller Area Network (CAN) was invented by Bosch GmbH [3] in order to provide reliable, fast communication between ECUs in automotive networks. However, it was not designed for security, and as such remains vulnerable to various attacks from both physical and wireless interfaces. Although the majority of cars are vulnerable to attacks through physical media, such as the On-Board Diagnostics port (OBD-II) [23], recent developments in automotive technology have made cars increasingly connected with each other, mobile devices, and infrastructure via wireless interfaces [6, 11]. This connected car technology enables functions such as cooperative adaptive cruise control, telematics, and traffic management [1, 9]. However, at the same time, it opens new attack vectors for the CAN, through wireless interfaces or OBD-II access. For instance, attackers can exploit the cellular data link connecting to the telematics ECU to send malicious messages into the CAN bus, to take control of other ECUs and the vehicle as a whole [4, 10, 12, 14]. These types of vulnerabilities pose great risks to drivers and passengers, as even small disruptions of car control can cause lethal results.

One way to improve the security of CAN bus intra-vehicle networks is to add message authentication. However, implementing authentication protocols in automotive networks also introduces

delay (overhead). Critical messages must be transferred and processed as fast as possible, and extra overhead could prove fatal in certain circumstances. Therefore, any practical authentication system must keep overhead and delay at a minimum. Furthermore, there are already millions of cars in use, so it would be unrealistic to recall cars and change their ECUs to add authentication mechanisms or modify the CAN protocol itself. A desirable solution should be backwards-compatible with existing CAN systems, with minimal changes to ECU software and the CAN protocol.

To address the above challenges, we propose a real-time authentication mechanism for securing in-car CAN bus communications. Our major contributions are summarized as follows.

- We propose the addition of a secure hardware module (the SECU) onto the CAN bus. The hardware module can perform key distribution and message verification, and can destroy malicious messages before they are fully received by ECUs. The module significantly enhances the security of in-car network communications and reduces the overhead of key management. With this SECU, only software updates are required for existing ECUs. They only need to compute MACs, and they do not need to conduct verification. Therefore, delay and computation overhead on message verification are also reduced compared with traditional message authentication protocols.
- We collect 6.673 million CAN messages from various cars and conduct entropy and pattern analysis of the messages. We find that the CAN messages collectively have low entropy, with an average of 11.436 bits. This finding supports our proposal for CAN message compression, which allows us to significantly reduce the message size to fit the message and its MAC within a single CAN frame, thus enabling fast authentication.
- In order to find an optimal compression method, we test various coding schemes and find that Huffman coding with interframe compression meets message space and storage space requirements.
- We develop a new ECU synchronization process to allow more efficient transfer of compression trees.
- We implement our authentication mechanism on a testbed using industry standard MCP2551 CAN transceivers and Raspberry Pi embedded systems.
- We conduct experimental evaluation using the testbed. Evaluation results show that our mechanism can achieve real-time message authentication on the CAN bus with minimal latency.

2 RELATED WORK

Although instances of cyberattacks on vehicles are fairly recent, there are already some proposed models or mechanisms for authentication and security in automotive networks. To date, however, we are not aware of an approach that successfully provides low latency, reliability, cost efficiency, and security.

One notable exploration of authentication in intra-vehicle networks is the CANAuth protocol [21]. CANAuth is built upon the CAN+ protocol [24], which allows for transmission of up to 16 bytes of additional data per byte of CAN data. However, to use CAN+, one must install special CAN+ transceivers on every ECU in one's car, severely diminishing CANAuth's ease of implementation. This renders CANAuth impractical for cars already on the road, since doing so would require accessing numerous ECUs embedded in the critical systems of the car.

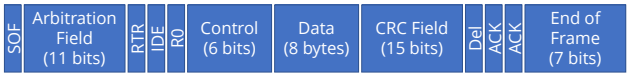

Figure 1: CAN Bus Frame

Another system, proposed by Mundhenk et al. [17], focuses on adapting traditional encryption standards to automotive networks. This system encounters the same issue as CANAuth: It requires the installation of hardware modules on each ECU, making implementation more difficult.

There are several software-based solutions proposed for intra-vehicle network security. The most notable of these is VeCure [22], which utilizes precomputation of MACs to achieve delays of as low as 50 microseconds. Despite this, VeCure relies on sending two messages in immediate succession, but sequential receipt is not guaranteed due to the possibility of a higher-priority message arriving between the two messages. Furthermore, VeCure's delay, though small, is still far greater than that of an unmodified CAN bus. Other software-based solutions include those of Schweppe et al. [18] and Glas et al [7]. Schweppe et al. have focused on creating secure key-distribution and secure communication channel protocols. Glas et al. have explored different placements for MACs. Both of these software-based solutions introduce substantial latency.

A promising approach to resolving the issue of processing power is to add a more powerful node to the in-car network that would perform security functions. Seifert and Obermaisser propose that such a "security gateway" be deployed in the intersection of all the different buses in the in-car network [19]. This hardware node would introduce a single point of failure - if it failed, then the entire in-car network would fail. Also, the use of this security gateway does not prevent individual networks from being compromised, as it only filters traffic between the different types of networks [15][16]. Authentication between a user device and an in-vehicle gateway node is studied in [10]. However, the case in which the ECUs are compromised is not considered in [10].

3 SYSTEM AND ATTACK MODEL

Here we briefly illustrate the CAN architecture and protocol. We also introduce our attacker model.

3.1 System Architecture of CAN

As a broadcast network, the ECUs in a CAN are laid out in an arrangement following the ISO 11898-2 [13] standard (also called high speed CAN), the most commonly used architecture in automotive and industrial applications. High speed CAN connects all ECUs on a linear, two-line bus terminating on either end with 120Ω resistors connecting the two lines. The two lines have a base, recessive voltage of +2.5V with the CAN high line increasing to +3.5V and the CAN low line decreasing to +1.5V for dominant bits [5].

There are four types of CAN messages: data frames, remote frames, error frames, and overload frames. Data frames, as shown in figure 1, are the most common type of CAN message. Data frames consist of one dominant 0 bit denoting the start of frame, followed by an 11-bit arbitration field or message identifier (ID). The next three bits are used for protocol purposes that we do not discuss in this paper. The following four bits comprise the Data Length Code (DLC), which signifies the length of the data field in bytes. The eight-byte data field follows the DLC, and is succeeded by a 15 bit CRC and a one-bit delimiter. The message ends with nine bits that are used to acknowledge the reception of the message and mark the end of the message. Seven bits of interframe space follows before the next message is sent. Remote frames are simply data frames with an empty data field, and are sent to request data. Error frames are sent to signify that an error was detected on the CAN bus. Overload frames are sent when an ECU cannot process messages fast enough, and requests for the last message to be repeated. Improvements in ECU technology have eliminated the need for overload frames [3].

3.2 Attack Model

We assume the attacker's goal is to send malicious messages into the CAN bus to gain control of the car or interfere with normal operation of the car. We consider two types of attackers: the outside attacker and the inside attacker.

The outside attacker can be a malicious device which is attached to the CAN bus. It can be a malicious ECU attached to the CAN bus or a compromised OBD-II dongle. We assume there are no shared secrets between the outside attacker and any ECUs on the CAN bus.

The inside attacker can be a compromised ECU, which has a shared secret key with the SECU, but not with any other ECUs. We will explain why it is the case in more detail when we present the design of our authentication mechanism in Section 4.2. Therefore, the inside attacker can generate legitimate messages.

We assume both outside and inside attackers can replay messages transmitted by other ECUs or inject arbitrary messages into the CAN bus. They can launch the following attacks.

- Collision attacks, in which they generate a large amount of message and MAC pairs in order to have some messages pass the authentication.
- Bit injection attacks, in which the attacker overwrite recessive bits (low voltage) with dominant bits (high voltage) to modify the contents of a message. Bit injection is limited to overwriting recessive bits.
- In-protocol denial-of-service (DoS) attacks, in which the attacker attempts to send messages at a high data rate in order to prevent other legitimate communications. We call it "in-protocol", since we assume the attackers still follow the CAN medium access control, CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance).

We do not consider jamming attacks, where attackers send noise signals into the CAN bus to disrupt all communications. This kind of attacker does not comply with CSMA/CA and can send the noise signals at any time no matter there is any communication in the network or not. In this case, a CAN bus jamming detection and isolation mechanism has to be implemented or the car has to be pulled over for safety, which is outside of the scope of this paper.

4 PROPOSED AUTHENTICATION MECHANISM AND APPROACH

In this section, we will present our authentication mechanism that can be implemented in existing intra-vehicle CANs. Our mechanism includes adding a hardware-based security module onto the CAN bus, updating software on the existing ECUs in the intra-vehicle networks, a compression method to reduce the message size, applying truncated SHA-3 for MAC, and a synchronization method to distribute Huffman trees to the corresponding ECUs.

4.1 Design Goals

The main reason most of the attacks discussed in Section 3.2 can be successful lies in the fact that the CAN lacks message authentication. When adding a message authentication mechanism to the existing CAN, we aim to achieve the following design goals:

- Compatibility: The authentication mechanism should be compatible with the existing CAN protocol. It should not require any changes to the CAN protocol, which has been widely adopted and is difficult to modify for deployed vehicles.
- Easy deployment and maintenance: It should have low deployment overhead and low maintenance overhead. It should not require any hardware changes or replacements of the existing ECUs in the intra-vehicle networks. It is preferable that the ECUs only incur software updates, and that these software updates should also be efficient.
- Fast/Real-time: The authentication mechanism should not introduce non-tolerable delay that affects the normal operation of the car. That is, it should be fast and real-time, with minimum latency.
- Low cost: The overall cost to implement the entire authentication mechanism should be low, in terms of hardware and software cost as well as labor.
- Low key management overhead: The key distribution in the initialization phase should be efficient. The key management overhead should be low when ECUs are added onto or removed from the CAN bus.

4.2 Authentication and the SECU Module

To achieve the above design goals and defend against the attacks discussed in Section 3.2, we propose an in-car security module, named the Security ECU (SECU). It will act as an authentication module to verify CAN messages, detect/block malicious messages, and facilitate key distribution.

Our basic idea is as follows: The SECU will share a unique key and a counter with each ECU on the CAN bus. When a legitimate ECU sends a message, it will first compress the message and then generate a MAC of the counter and the secret key. The counter will be increased by one for each transmitted message. The ECU then fits the compressed message and the MAC into one CAN frame, and sends it onto the CAN bus. The SECU will perform the message verification on behalf of the intended receiver(s) of the message. If

the verification passes, the receiver(s) simply decompress the message and use it as a normal CAN message. If the verification fails, the SECU will corrupt the CAN frame before it is fully received by the intended receiver(s). The corrupted CAN frame will be ignored by the intended receiver(s) as if it was never received. Therefore, a malicious message generated by the attacker will inflict no damage on the system.

In order to realize this idea, we need to address the following technical issues: key distribution and management, message compression, and quick malicious message detection and corruption. We present our solutions to these issues in the following subsections.

4.3 Key Distribution and Management

With our system setting and authentication design, an ECU only requires key sharing with the SECU, and not with other ECUs. This design significantly simplifies key distribution and management. Existing solutions require key sharing between the sending and receiving ECUs, which produces more overhead on key distribution and management. In the initialization phase, SECU can generate unique keys and distribute them to each ECU in the CAN. We assume this phase is secure, which can be ensured by a technician in a mechanic shop or car dealership. When a legitimate new ECU is added to the CAN, it only needs to obtain the necessary secret key(s) from the SECU.

4.4 Feasibility of Message Compression

The key challenge of adding a MAC to a CAN message is that it should only introduce a short delay without measurably affecting the operation of the car. If we can fit the CAN message and MAC in one CAN frame, we will achieve our goal. Since the CAN frame has only 8 bytes, we need to use a short MAC, but at the same time, provide enough security strength.

First, we explore the feasibility of fitting the message and its MAC in one CAN frame. To do this, we collect real CAN bus data from various cars and conduct entropy analysis on the data. We find that the entropy of the CAN bus messages is around 12 bits, which demonstrates that the transmission data has low entropy and thus is capable of compression. More detailed information about entropy analysis of CAN bus messages will be presented in Section 5.2.

4.5 ECU Update Protocol

Immediately after the SECU is added to the CAN, it will collect extensive CAN data during vehicle operation. This data will be used to build the Huffman trees (which will be detailed in Section 4.7) and will ensure that initial latency is at a minimum when the system starts functioning. Once sufficient data is collected, the SECU will update all ECUs in the CAN to communicate using our protocol. (See Figure 2.) Unique secret keys randomly generated by the SECU will be assigned to each message ID, and are known to only the SECU and the ECUs that send messages of that message ID. Key distribution and Huffman tree generation should be completed in a secure area such as the manufacturer's test site, since they could be points of attack.

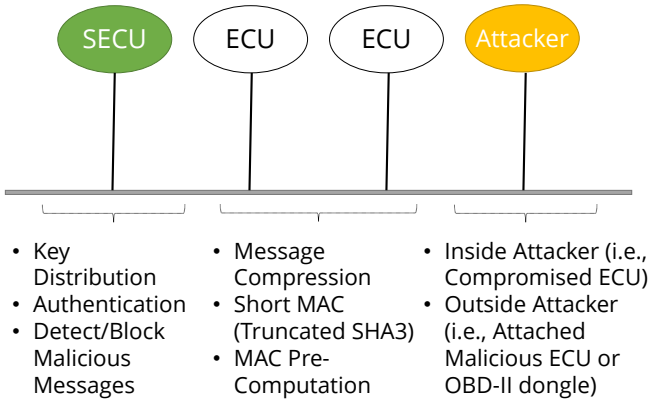

Figure 2: SECU Module

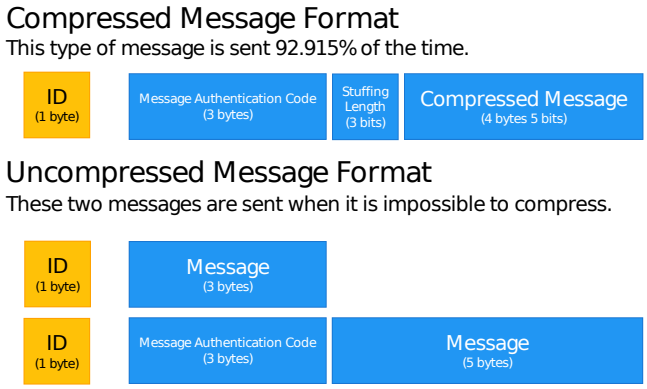

Figure 3: Different types of CAN data messages of modified protocol

4.6 MAC Generation

The update will provide ECUs with MAC generation capabilities, and will create counters for each message ID. Every time an ECU sends a message, it increments the counter associated with the ID of that message. The SECU does the same when it receives the message. Each ECU stores only the secret keys and counters corresponding to the message IDs that it sends. Since the SECU verifies incoming messages, ECUs do not need to store secret keys and counters corresponding to message IDs that they receive. The SECU stores a look-up table with a list of message IDs as the keys and each corresponding secret key and counter as values.

To generate a MAC, an ECU hashes the relevant secret key and counter together. The counter is necessary to prevent replay attacks. The ECUs and the SECU precompute hashes for future counter values that are stored in lookup tables to save time when they send a message. This is possible because the only way for attackers to record and reuse MACs is easily detectable to the SECU, allowing it to interrupt malicious messages and invalidate the stolen MAC.

4.7 Compression Methods

A simple lookup table method requires more storage space than is practical for use in ECUs. We tested several different compression algorithms, recorded the sizes of compressed messages and the sizes of the compression dictionaries, and found that Huffman coding was the optimal choice. We generated the Huffman trees from CAN data logs using bytes as symbols, and encoded each CAN message separately. We generated a Huffman tree for each unique message ID, so each ECU would only have to store the Huffman trees for the message IDs that it communicates with. Refer to Section 5.

From our pattern analysis, we found that there were many cases where messages of the same ID varied only slightly over time. Videos have similar temporal redundancy, which the MPEG compression algorithm utilizes through interframe compression, where some frames consist of the difference from a reference frame prior to the current frame. Our implementation of interframe compression also utilizes temporal redundancy, sending the change in each byte from message to message.

After data collection has finished, interframe compression is applied on the messages within each unique message ID, and Huffman encoding is performed on the result. To avoid potential errors, all 256 possible bytes are encoded into each Huffman tree, including those with a frequency of zero. For efficiency, we impose a maximum compressed message length, $N = 64 - (MAC + SL)$ bits. All messages with compressed versions longer than this length will not be compressed. SL signifies the stuffing length and is 3 bits long; this is the number of zero bits that were stuffed in front of the encoded message during compression to make the final data field a whole number of bytes (i.e. if the encoded message is E bits long, then $SL = 8 - (E \bmod 8)$ in binary, a value from 000 to 111). The bit stuffing is necessary due to the CAN protocol mandating that a whole number of bytes be sent in the data field, which is not necessarily the case for bit strings returned by compression algorithms.

Compressed messages will consist of one CAN message of the format $MAC + SL +$ encoded message. The previous message will be stored for each unique message ID to allow for the change values to be processed. Each of the eight bytes of the stored message is considered independently. If there is no previous stored byte for that byte of the message, the current byte is taken to be the actual byte value, not the change value. This procedure takes care of the fact that message lengths vary even within the same message ID. Uncompressed messages, whose compressed counterparts are longer than the maximum compressed length, will consist of two CAN messages: the first with the first three bytes of the uncompressed message, and the second with the 3 byte long MAC plus the remaining (up to) five bytes of the uncompressed message. For both types of messages, the MAC is placed before the message in order to give the SECU more time to authenticate the message and destroy it if it is invalid. Refer to Figure 4 for a diagram of message formats.

The MAC for compressed messages will comprise only the SHA-3 hash of the message ID's counter and secret key. The MAC for uncompressed messages will consist of the SHA-3 hash of the first 3 bytes of the uncompressed message along with the message ID's counter and secret key. ECUs will be able to identify an uncompressed message from the DLC in the control bits of a CAN message. If the length of the data field is equal to that of a MAC (i.e. three bytes), then that message contains the first three bytes of the uncompressed message, and the next message of that message ID will contain the MAC plus the remaining (up to) five bytes of the uncompressed message. This uncompressed message represents the actual data values, as opposed to the change in values for compressed messages. The MAC is generated by hashing the first three bytes of the uncompressed message in addition to the secret key and counter to prevent spoofing of the uncompressed message, since the first three bytes are sent in a separate CAN message. The remaining (up to) five bytes of the uncompressed message are protected from spoofing by being in the same message as the MAC itself. The uncompressed message protocol is structured as described in Figure 4 in order to allow for the SECU to have enough time to verify and destroy malicious messages.

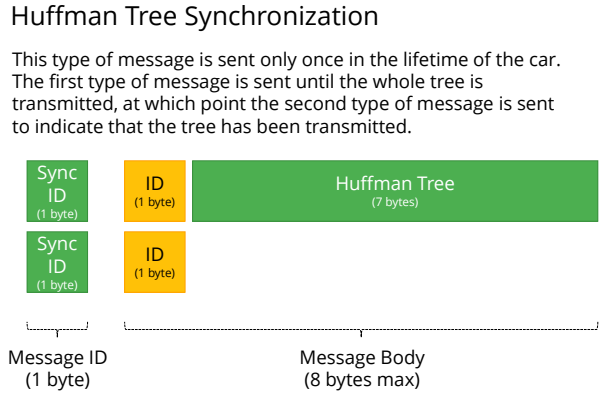

Figure 4: Compressed sync messages

4.8 Synchronization Methods

To reduce the processing power and memory strain on normal ECUs, data collection and Huffman tree generation are performed on the SECU. Thus, for a short duration after installing the SECU, the vehicle will continue to use the unmodified CAN protocol while the SECU performs data collection on normal CAN bus traffic. After data collection, interframe compression, and Huffman coding are finished, the Huffman trees are serialized to be sent within CAN messages for distribution to the other ECUs. Huffman trees are full, so they can be serialized by storing preorder traversal of the tree along with a bit identifying each node as a parent node or leaf node. The zero bit will represent a parent node, so when deserializing, the next bit in the data field will be read as a node identifier bit. However, when a one bit representing a leaf node is read during deserialization, the following 8 bits will be read as the value of the node in the Huffman tree.

There are at most 256 possible bytes that can exist as leaf nodes in a Huffman tree in our protocol. In a full tree, if there exist N leaf nodes, there must be $N-1$ parent nodes. Thus, $256+255 = 511$ bits ≈ 64 bytes are required as node identifier bits, so $64 + 256 = 320$

Figure 5: SECU Testbed

bytes are required to serialize the Huffman tree. Since we generate a unique Huffman tree for each message ID, 255 Huffman trees must be sent, and a message ID must be sent for each sync message. (From analysis of our collected CAN data, we found that the Subaru, Honda, Toyota, and Lexus had 24, 28, 35, and 78 unique message IDs respectively. Thus, we found it reasonable to assume a maximum of 255 unique message IDs, since one ID will be used for sync messages.) This results in $320 * 255/7 \approx 11700$ messages needed to send every Huffman tree. From our message analysis, we found that for any given message ID, at most approximately 100 messages are sent per second. Therefore, in order to not slow down bus traffic drastically, we assume a rate of 50 sync messages being sent per second after data collection and encoding are finished. Thus, only $11700/50 = 234$ seconds ≈ 4 minutes is necessary to finish sending all Huffman trees. It is also important to note that this slowdown is only a one-time occurrence.

Once Huffman trees are completely distributed, the SECU will send a unique message to signal the completion of Huffman tree distribution and the initiation of compressed message sending. This message contains the sync ID in place of a normal message ID with no data following it. Refer to Figure 4 for a diagram of sync messages. Using interframe compression, any new message can now be generated with a different permutation of change values. The Huffman trees encode every possible byte value, so all potential change values have a corresponding Huffman code.

4.9 Malicious Message Interruption

Interruption of malicious messages is conducted by the SECU, which sends high voltage through the CAN high line when an invalid MAC is detected. In the CAN protocol, high voltage in the CAN high line represents dominant bits. Since the end-of-frame for all messages must consist solely of recessive bits, writing dominant bits over the recessive bits causes the end-of-frame to be corrupted. The interrupted message is thus treated as a corrupted message and ignored by all ECUs.

5 RESULTS AND ANALYSIS

To test our proposed method for securing the CAN, we created a model composed of both software programs that we coded to implement the changes that we proposed to the CAN protocol, and a hardware testbed on which we ran our programs and simulated a real CAN.

5.1 System Implementation

Most of the message processing code was written in Python for easier prototyping. A commercially available implementation would be written completely in C to make message processing as fast as possible. We wrote a Python extension in C to compute SHA-3 (Keccak) hashes [2] in our testbed. Our implementation differs from the official SHA-3 implementation in that we modified it to increase speed and to generate a hash of the desired length. This SHA-3 implementation is a C implementation, which we used in order to gather statistics about how fast the algorithm would perform in a car equipped with an SECU.

Our testbed modeled a standard CAN bus with three ECUs. It incorporated two Raspberry Pi 2 Model B boards and one Raspberry Pi 3 board running the Raspbian operating system, with the Raspberry Pi 3 acting as the SECU and the Raspberry Pi 2's acting as regular ECUs. All three Raspberry Pi's utilized a Raspberry Pi to Arduino (Arduberry) shield and an Arduino CAN Shield to communicate using the CAN protocol. The CAN shields were connected by wires in order to simulate a CAN bus. An Arduino script facilitated communication between the Python scripts running on the Raspberry Pi and the hardware on the CAN Shield. This system was used to ease prototyping. A faster implementation could be achieved by using field-programmable gate arrays. Although there are actually 2048 possible message IDs in the CAN protocol, the testbed we are using only supports up to 256, and all our subsequent calculations were done with this limit. However, CAN implementations often have far fewer than even 256 message IDs, as shown in a recent analysis of Mini-Cooper CAN message IDs, which found only 15 unique message IDs [20].

We verified that the ECUs within our testbed could communicate correctly by having one model ECU send captured CAN data over the bus to the other, which wrote the data to the terminal. We demonstrated normal traffic with model ECUs communicating with each other, and then repeated this test using our security measures. We recorded live CAN data using the OBD-II diagnostic port of a 2011 Toyota Camry and replayed it on our testbed. We also collected CAN data from a 2011 Honda Accord, a 2010 Lexus GS350, a 2015 Subaru Forester, and a 2007 Jeep Liberty.

5.2 Entropy Analysis

Message collection was conducted through attaching a Raspberry Pi 2 Model B to each car's OBD-II port using a DB9 to OBD-II adapter. Messages were logged using a script that read messages from the DB9 port into a text file. We had the most complete data for the 2011 Toyota Camry, so it was used for the majority of our analysis.

Message collection for the Jeep Liberty, Honda Accord, Toyota Camry, and the Subaru Forrester was conducted during driving on local roads, as well as while the car was parked. Message collection for the Lexus GS350 was conducted while the car was moving at slow speeds (< 5 *mph*), and while it was parked.

We calculated message entropy using the Shannon entropy definition $H(x) = -\sum_{i=0}^{N-1} p_i \log_2 p_i$ on the messages we collected from the vehicles. The results are shown in Table 1. The average message entropy across all five cars was 11.436 bits, which was sufficiently low that we found the use of compression algorithms viable.

5.3 Experiment Results

In Table 2, it appears that as the number of rounds per hash and the number of packets increases, the time required per round decreases. This is not true. The total time values in the tables include the constant startup time required in addition to the actual time taken to produce the hashes, which has a linear relationship with the number of rounds and MACs generated. This is demonstrated

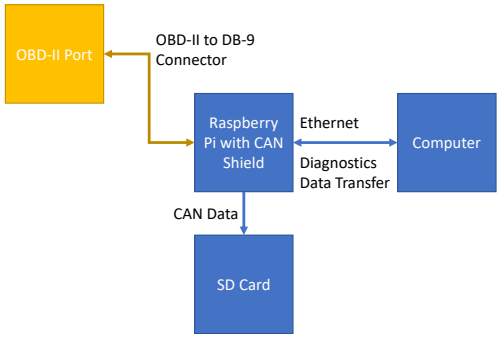

Figure 6: Message Collection

Table 1: Entropy Analysis

Vehicle	Entropy (bits)
Subaru Forester 2015	12.564
Toyota Camry 2011	12.792
Honda Accord 2011	12.217
Lexus GS350 2010	8.498
Jeep Liberty 2007	11.111

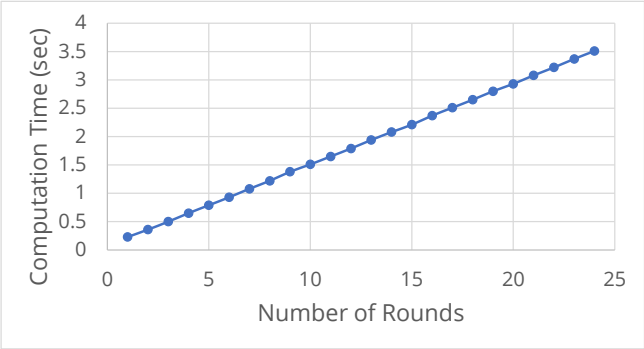

Figure 7: The time to compute all 53,920 MACs for the real CAN data

to be true in Figure 7, which is a plot of the hash time in seconds with respect to the number of rounds the hash performed. As the number of rounds and packets increases, the time per round approaches the actual value of time taken for each round, roughly 2.7 μs per round.

A MAC collision occurs when the same MAC is generated from the secret keys and counters of two distinct messages. We performed MAC collision testing to ensure that MAC collisions did not appear too frequently. When MAC collisions occur too often, malicious users can attempt to reverse the hash or perform replay attacks. As described in Section 5.5, replay attacks are sufficiently mitigated by the CAN protocol. Reverse engineering of the hash to find secret keys and counters is largely prevented by our implementation as well as the nature of Keccak, since each message ID has its own secret key and counter, which increases the difficulty

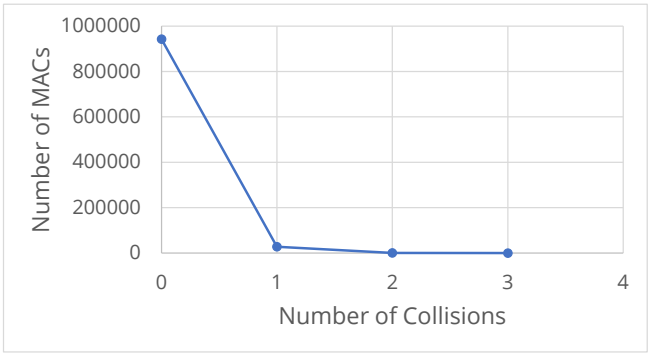

Figure 8: The number of collisions for MACs generated from the real CAN data

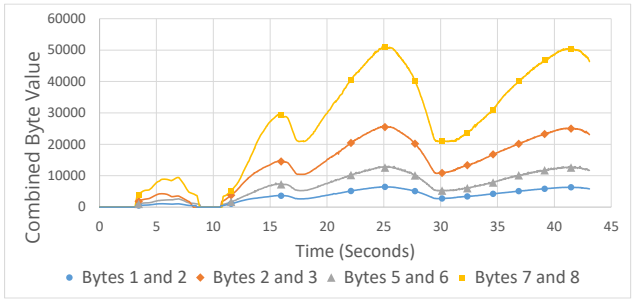

Figure 9: Combined byte values of a message that forms a gradually changing interframe-compressible pattern

of reversing the hash, and because it is extremely difficult to determine the data that was hashed from the hash itself, even if there are many collisions [2]. Figure 8 and 10 show that for randomly generated data there tends to be a normal distribution as expected, but real, captured data is skewed right, having a large amount of MACs with few collisions. To calculate the times in the tables below, we used the "time" command in the Linux terminal taking the sum of user and system time as the time the program used to run since that is better for benchmarking. This is because the real time field shows the actual elapsed time and not solely the time spent on the program, which would be the sum of user and system time.

We also tested various compression algorithms (Table 3) and parameters within those algorithms (Table 4) to find the most promising method. The algorithms were all able to perform in a reasonable amount of time, with the exception of arithmetic encoding. Of all of the different compression methods we tested, a combination of Huffman and interframe compression had both the smallest average message size and required storage space. We were able to compress 92.915% of our sample data to within 4 bytes and 5 bits, which is the limit for compressed message size in our protocol. Larger messages were left uncompressed and had to be sent as two messages. Even taking this into account, the average message size was only 1.830 bytes.

Through our entropy analysis, we found that for some message IDs the contents of the messages were almost always identical. Thus, we created separate compression dictionaries for each ID.

Table 2: MAC Generation Summary Data Table

Data Type (Rounds)	Number of Packets	Average Hash Time (s)	Time Per Packet (μs)	Time Per Round (μs)
Captured Data (13)	53,920	1.957	36.29	2.792
Captured Data (24)	53,920	3.526	65.39	2.725
Randomly Generated (13)	1,000,000	35.695	35.695	2.746
Randomly Generated (24)	1,000,000	64.896	64.896	2.704

Table 3: Compression Method Comparison

Method	Size (Bytes)	Storage (Bytes)
Uncompressed	6.482	0
Lookup Table	3	6654502
Arithmetic	4.003	2424
Shannon-Fano	6.482	12384
Huffman	3.733	84520
Huffman + Time	1.830	79912

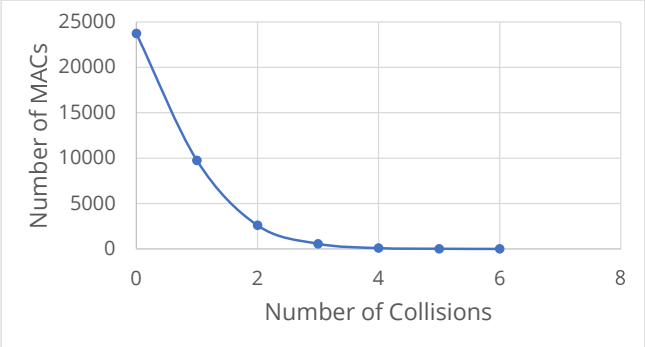

Figure 10: The number of collisions for MACs generated from the randomly generated CAN data

Table 4: Huffman Implementations

Type	Avg Size (bits)
Body as Symbol - Single Tree	44.182
Body as Symbol - Individual Trees	52.565
Byte as Symbol - Single Tree	29.860
Byte as Symbol - Individual Trees	24.963

When examining message bodies, we found that values would increase or decrease by small amounts (see Figure 9), which can be efficiently compressed using interframe compression.

5.4 Performance Analysis

The goal of our MAC generation script is to minimize the time it takes to generate the MAC while maximizing the security of the MAC. For our implementation, the minimum size of the data field is 4 bytes. Thus, the shortest possible message that could be sent along the CAN containing a MAC would be 76 bits long, and assuming a high speed CAN bus with a baud rate of 1 Mb/s, that would

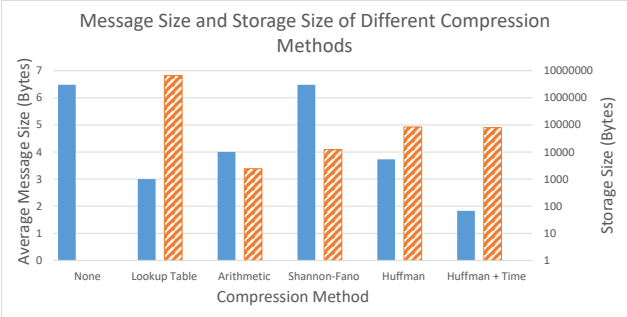

Figure 11: Storage space usage of different compression algorithms

mean that it would take at least 76 microseconds to send each message. The data we collected on our MAC generation script in Table 1 shows that it would take approximately 36.29 μs on average to complete 13 rounds of the SHA-3 hashing function and produce a MAC. Since 36.29 μs is significantly less than 76 μs, our hashing function can generate MACs faster than messages can be sent. This means that all MACs can be precomputed before they are required, resulting in minimal latency due to MAC generation.

We ran 10,000 iterations of our Huffman encoding and decoding algorithms, rewritten in C for speed, on the data collected from various vehicles. We found that the Huffman encoding takes 2.92 μs, while the decoding process takes 2.08 μs. The reason the encoding algorithm takes longer than the decoding algorithm can be explained by the fact that messages that cannot be compressed are still run through the encoding algorithm, while the decoding algorithm only decompresses messages that are proven to fit within 4 bytes when compressed. Assuming the worst case of an uncompressed message, adding together the speed of our MAC generation (36.29 μs), our encoding algorithm (2.92 μs), the total processing time is 39.21 μs. Since the message is not compressed, no decoding is necessary. However, encoding is still necessary to verify that the message cannot be compressed. For compressed messages, the MAC can be precomputed and the total processing time is 5.00 μs.

Our solution is faster than other proposed solutions, such as VeCure, whose total processing time is ~50 μs. VeCure itself functions faster than a majority of existing solutions [22].

As shown in Figure 11, the average compressed message size for messages compressed using Huffman coding and interframe compression is drastically lower than the average compressed message size for any other compression algorithm tested. It also requires less than a tenth of the storage space needed for the original lookup table method. Lempel-Ziv compression, which is based on

run-length encoding, does not compress CAN messages well since CAN messages are so short. Arithmetic coding functions similarly to Huffman, but encodes and decodes much slower when compared to Huffman coding. Shannon-Fano compression performed quite well, but it was still less efficient that Huffman. Since every other compression algorithm resulted in higher message sizes without a substantial benefit in storage space saved, we found that Huffman coding and interframe compression was the best compression algorithm for use in our protocol.

We determined that a 3 byte long MAC is sufficiently secure against brute-force attacks in an automotive environment. An average entire CAN message implementing our security solution is roughly 96 bits long. Assuming a bitrate of 500 kilobits per second, the most common bit rate for automotive applications, that 11 consecutive recessive bits occur at the end of every message (they may be even less common), and that the attacker is able to send messages at every possible opportunity within the CAN protocol, this means that they are able to make 32 attempts every (128 + 32) * 96 / 500,000 = 0.031 seconds. For a 3 byte long MAC, there are 2^{24} possibilities, so an average of 2^{23} attempts is necessary to brute force the MAC. Therefore, it takes approximately 2^{23} / 32 * 0.031 = 8053.064 seconds ≈ 2.237 hours on average to brute force a MAC in our system. An attacker will not have this much time, since the MAC changes every time a message with the same ID is sent, due to the counter value incrementing. Replay attacks are prevented by the same incrementing counter.

5.5 Discussion on Overhead and Security

Based on our compression dictionary specifications, we can estimate hardware requirements for the SECU. The maximum number of message IDs we have observed in a car is 40, and the worst case for each Huffman tree results in a tree with 511 nodes. Since each node requires approximately 9 bytes on a 32-bit system (1 byte for the data, 4 bytes for each pointer), the SECU will need at least 9 * 40 * 511 = 183,960 bytes of storage space. The SECU also needs to store secret keys, but 40 8-byte-long secret keys comprises only 320 bytes, which is small compared to the storage space needed for the compression dictionaries. Processing capabilities must be at least that of a Raspberry Pi 2 Model B, because our results were generated on that device, and devices of lower processing power may not perform well enough to allow proper implementation of our protocol.

We are able to generate MACs without hashing their associated messages because the ability of attackers to record and replay MACs is severely limited by the CAN bus and our protocol. Counters increment every time a message with a specific ID is sent, so an attacker must simultaneously record a legitimate MAC and destroy that message to prevent this. However, if the SECU detects a corrupted message, it will send a top priority message for all IDs causing all ECUs to increment their counters, preventing this attack entirely.

Our solution may appear to have a single point of failure, but this is not the case. The security features would no longer work, but the driver could still drive the car. A true single point of failure would cause the entire system to fail if the point of failure could not function. Because of this, our system is superior to other centralized security systems for the in-car network. Where other systems have a true single point of failure, our system, if rendered nonfunctional, will not inhibit CAN bus traffic, though that traffic will not be secure.

We consider attacks made using a compromised ECU to inject messages into the CAN bus. An inside attacker, one that gains complete control over the compromised ECU, will be able to obtain all secret keys and counters that are stored on the ECU. However, since the most easily compromised ECUs generally deal with entertainment and non-essential systems, the message IDs that correspond with the compromised keys and counters will generally not be of value to malicious users that want to gain control of critical systems. Section 3.2 describes the capabilities of an attacker that has compromised an ECU such that they are able to inject customized messages into the CAN bus.

Message injection through an ECU would require a valid MAC. Reversing the hash is extremely difficult, due to the security of the SHA-3 (Keccak) hashing algorithm. However, since the MACs are relatively short, an attacker could potentially conduct a brute-force attack to cause a MAC collision. The CAN protocol limits the effectiveness of such an attack. The Transmit Error Counter is a counter implemented on the firmware level that increments by 8 on an ECU whenever a message that is transmitted by that ECU receives an error frame. Invalid MACs will cause the SECU to send error frames. Once the counter exceeds 255, after only 32 messages, the ECU is turned to the Bus Off state, disallowing it from further CAN bus communication until "after 128 occurrences of 11 consecutive 'recessive' bits" (approximately 128 messages) occur on the bus.

The CAN protocol can prevent in-protocol DoS attacks. Attackers cannot conduct an in-protocol DoS attack using remote frames, since data frames are dominant to remote frames. Overload frames cannot be used to continually delay messages, as the CAN protocol limits ECUs to a maximum of two overload frames. The Transmit Error Counter prevents DoS attacks using data and error frames in the same way that it prevents brute-force attacks.

Bit injection attacks are also prevented by the CAN protocol. This attack requires the corresponding CRC to be bit injected to properly match the transformed message. The CAN protocol prevents this attack because transmitting nodes monitor the bus to make sure the message on the bus is the same message that they are transmitting. During a bit injection attack, the transmitting ECU would detect a bit error and throw an error flag.

6 CONCLUSION AND FUTURE WORK

Our solution improves upon the many proposed solutions addressing the concerns of car security. It compresses messages so that the latency introduced by MACs is significantly reduced. MACs are precomputed to further reduce latency. A unique approach of jamming the CAN bus line is utilized by the SECU to temporarily take control of the entire network. Our solution is not only secure, but also faster than previous implementations of intra-vehicle network security. We improved upon our solution by conducting entropy analysis and modifying our protocol based on the results. However, there are certainly ways in which to expand and improve our

research. Our protocol might require additional memory and processing power for ECUs to decode messages.

We demonstrated that our solution causes low overhead, which allows the SECU to interrupt malicious messages. We were unable to trigger the friendly jamming immediately after processing a message due to hardware limitations of CAN transceivers. CAN transceivers store complete messages in a buffer and only send them once the message CRC has been processed. This prevents interruption of those messages, which would otherwise be possible if the transceiver instead sent bits directly. Therefore, custom hardware is necessary to allow software access to incomplete CAN messages. Due to cost limitations, we were unable to purchase this hardware. If manufactured in bulk, however, this hardware would become inexpensive.

Once that custom hardware is available for use, we recommend implementation in a vehicle for final testing, followed by releasing the solution for use in cars by the general public. We anticipate that our protocol will successfully prevent future attempts at hacking the central control networks of automotive vehicles.

ACKNOWLEDGMENTS

The vast majority of this work was done by Eric Wang, William Xu, and Suhas Sastry during the summers of 2015 and 2016 as part of the Aspiring Scientists Summer Internship Program (ASSIP) [8] at George Mason University.

REFERENCES

[1] Alipour-Fanid, A., Dabaghchian, M., Zhang, H., and Zeng, K. String stability analysis of cooperative adaptive cruise control under jamming attacks. In *Workshop on Security Issues in Cyber Physical Systems (SecCPS)* (2017), IEEE.

[2] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. The Keccak sponge function family, 2011. *Submission to NIST's SHA-3 competition* (2011).

[3] Bosch, R. CAN specification version 2.0. *Rober Bousch GmbH, Postfach 300240* (1991).

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

[5] Corrigan, S. Introduction to the Controller Area Network (CAN). *Texas Instrument, Application Report* (2008).

[6] Diewald, S., Möller, A., Roalter, L., and Kranz, M. Mobile device integration and interaction in the automotive domain. In *3rd International Conference on Automotive User Interfaces and Interactive Vehicular Applications (AutomotiveUI 2011)* (2011).

[7] Glas, B., Guajardo, J., Hacioglu, H., Ihle, M., Wehefritz, K., and Yavuz, A. Signal-based automotive communication security and its interplay with safety requirements. In *Proceedings of Embedded Security in Cars Conference* (2012).

[8] GMU. The George Mason University Aspiring Scientists Summer Internship Program. http://assip.cos.gmu.edu/, 2016 (accessed March 2, 2017).

[9] Häberle, T., Charissis, L., Fehling, C., Nahm, J., and Leymann, F. The Connected Car in the Cloud: A Platform for Prototyping Telematics Services. *IEEE Software 32*, 6 (Nov 2015), 11–17.

[10] Han, K., Potluri, S. D., and Shin, K. G. On authentication in a connected vehicle: Secure integration of mobile devices with vehicular networks. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)* (April 2013), pp. 160–169.

[11] Harding, J., Powell, G., Yoon, R., Fikentscher, J., Doyle, C., Sade, D., Lukuc, M., Simons, J., and Wang, J. Vehicle-to-vehicle communications: Readiness of v2v technology for application. *National Highway Traffic Safety Administration* (2014).

[12] Hoppe, T., Kiltz, S., and Dittmann, J. Security threats to automotive CAN networks–practical examples and selected short-term countermeasures. In *International Conference on Computer Safety, Reliability, and Security* (2008), Springer, pp. 235–248.

[13] ISO. 11898-2, Road Vehicles Controller Area Network (CAN) Part 2: High-speed medium access unit. *International Organization for Standardization* (2003).

[14] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 447–462.

[15] Makowitz, R., and Temple, C. Flexray - A communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems* (2006), IEEE, pp. 207–212.

[16] Müller, D., Sommer, D., and Stegemann, S. Local Interconnect Network (LIN), 2009.

[17] Mundhenk, P., Steinhorst, S., Lukasiewycz, M., Fahmy, S. A., and Chakraborty, S. Lightweight authentication for secure automotive networks. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition* (2015), EDA Consortium, pp. 285–288.

[18] Schweppe, H., Roudier, Y., Weyl, B., Apvrille, L., and Scheuermann, D. Car2x Communication: Securing the Last Meter - A cost-effective approach for ensuring trust in Car2x applications using in-vehicle symmetric cryptography. In *2011 IEEE Vehicular Technology Conference (VTC Fall)* (2011), IEEE, pp. 1–5.

[19] Seifert, S., and Obermaisser, R. Secure Automotive Gateway: Secure communication for future cars. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)* (2014), IEEE, pp. 213–220.

[20] Staggs, J. How to Hack your Mini Cooper: Reverse Engineering CAN Messages on Passenger Automobiles.

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

[22] Wang, Q., and Sawhney, S. VeCure: A practical security framework to protect the CAN bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)* (2014), IEEE, pp. 13–18.

[23] Zaldivar, J., Calafate, C. T., Cano, J. C., and Manzoni, P. Providing accident detection in vehicular networks through OBD-II devices and Android-based smartphones. In *2011 IEEE 36th Conference on Local Computer Networks (LCN)* (2011), IEEE, pp. 813–819.

[24] Ziermann, T., Wildermann, S., and Teich, J. CAN+: a new backward-compatible Controller Area Network (CAN) protocol with up to 16× higher data rates. In *2009 Design, Automation & Test in Europe Conference & Exhibition* (2009), IEEE, pp. 1088–1093.