– vatiCAN –
Vetted, Authenticated CAN Bus

Stefan Nürnberger(✉) and Christian Rossow

CISPA, Saarland University, Saarbrücken, Germany
nuernberger@cs.uni-saarland.de, crossow@mmci.uni-saarland.de

Abstract. In recent years, several attacks have impressively demonstrated that the software running on embedded controllers in cars can be successfully exploited – often even remotely. The fact that components that were hitherto purely mechanical, such as connections to the brakes, throttle, and steering wheel, have been computerized makes digital exploits life-threatening. Because of the interconnectedness of sensors, controllers and actuators, any compromised controller can impersonate any other controller by mimicking its control messages, thus effectively depriving the driver of his control.

The fact that carmakers develop vehicles in evolutionary steps rather than as revolution, has led us to propose a backward-compatible authentication mechanism for the widely used *CAN* vehicle communication bus. vatiCAN allows recipients of a message to verify its authenticity via HMACs, while not changing CAN messages for legacy, non-critical components. In addition, vatiCAN detects and prevents attempts to spoof identifiers of critical components. We implemented a vatiCAN prototype and show that it incurs a CAN message latency of less than 4 ms, while giving strong guarantees against non-authentic messages.

1 Introduction

In the highly competitive field of automobile manufacturing, only those have survived who have adopted the art of extreme cost savings by establishing a well-coordinated concert of manufacturers, suppliers and assemblers. It is this fragile chain, which now turns out to be too static when it comes to cross-sectional changes as would be needed by a radically new, secure architecture.

Even though security experts agree that an overhauled, security-focused architecture is much-needed [2,10,16,17], carmakers simply cannot easily change established designs. Arguably, two major obstacles are (1) the industry-wide "never touch a running system" attitude, which originates in legislative burdens and safety concerns, and (2) the overwhelming complexity of regulations in different jurisdictions of the world, which have fostered the outsourcing to highly specialized suppliers. This effect is even more amplified due to the tendency of acquisition rather than in-house innovation. As a result, desired functionalities are put out to tender and the hardware and software is instead developed by a long chain of suppliers. For example, Porsche claims to have the lowest manufacturing depth in the automotive industry with more than 80 % of production

B. Gierlichs and A.Y. Poschmann (Eds.): CHES 2016, LNCS 9813, pp. 106–124, 2016.
DOI: 10.1007/978-3-662-53140-2_6

cost spent for supplier's parts, while the remaining 20 % are spent on engine production, the assembly, quality control and sale of their vehicles [3].

This is in contrast with the needed extensive architectural changes to implement at least some level of security. The lack of automotive security engineering principles as opposed to the desktop computer world is not surprising. The most widely used automotive communication protocol *CAN*[1] was designed to run in isolation stowed away behind panels. Faulty hardware or damaged wires were the only likely threat to such an isolated system. A deliberate manipulation could only happen with physical access to the inside of the car. While these design principles were absolutely adequate for safety requirements back then, modern cars have meanwhile reached an almost incomprehensive complexity and moreover violate the ancient isolation assumptions due to their promiscuous connectivity such as Bluetooth audio, 3G Internet, WiFi, wireless sensors, RDS[2], and TMC[3].

It is not only potentially possible but it has been practically shown that vulnerabilities in these wireless connections exist [2]. An attacker can then write arbitrary messages on the CAN bus, which connects the car's computers, the so-called *Electronic Control Units* (ECUs). While the culprit is indeed a vulnerable ECU that can be compromised, the exploited fact is that the CAN topology is a bus. This broadcast topology allows any connected device, including a compromised ECU, to send arbitrary control messages. The receivers have no way of verifying the authenticity of the sender or the control data.

Contributions. In this paper, we propose vatiCAN, a framework for embedded controllers connected to the CAN bus, which allows both senders and receivers to authenticate exchanged data. First and most importantly, receivers can check the authenticity of a received message. Second, senders can monitor the bus for their own messages to detect fraudulent messages. In detail, vatiCAN provides

- **Sender and message authentication** in the CAN bus broadcast topology, which prevents fake messages from illegitimate senders from being processed.
- Security against **replay-attacks** by incorporating a global nonce.
- **Spoof detection** of own messages in software by bus monitoring.
- Full **backward compatibility** as message payloads, sender IDs and most importantly CAN transceiver chips are left unmodified, which allows legacy devices to work without modifications.
- **Spoof prevention** of own messages is possible in hardware by changing bus arbitration.

While the possible improvements on a legacy architecture are somewhat limited by the intended backward compatibility, this paper shows what can be achieved when backward compatibility is the utmost goal. It thereby lays the foundation for automakers to increase the status quo of security while adhering to the established structures of minimal change.

[1] Controller Area Network - Developed by BOSCH and Mercedes-Benz in 1983.
[2] Radio Data System - digital payload for FM radio broadcast, e.g. station name.
[3] Traffic Message Channel - Traffic information over FM radio for navigation systems.

2 Background

To address the need to connect different sensors, actuators and their controllers with each other so that they can make informed decisions, BOSCH developed a new communication bus in 1983 [7,13]. For example, the widespread traction control system (TCS[4]) could use CAN to connect the necessary sensors (wheel rotation) and actuators (brakes). The TCS monitors the wheel spin on each of the four wheels and intentionally brakes individual wheels to get traction back (see Fig. 1).

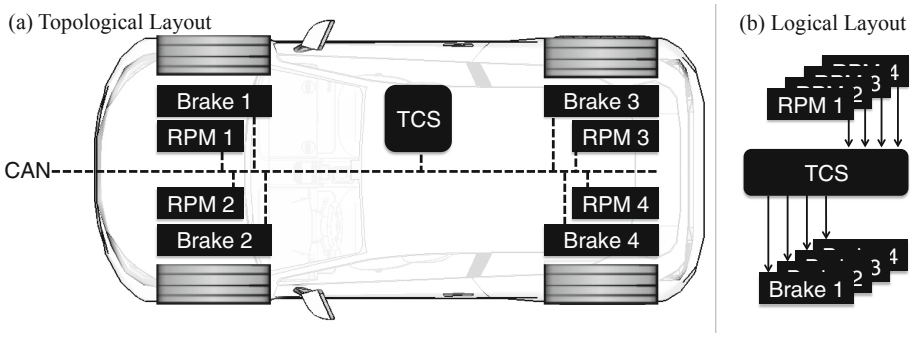

Fig. 1. Bus Topology on the example of the Traction Control System (TCS): Sensors (wheel RPM) are read as input while actuators (brakes) work as output.

CAN transmits so-called *CAN frames* consisting of a *priority*, the actual message *payload*, its *length* (1 to 8 bytes), and a *CRC checksum* followed by an *acknowledgment flag* (see Fig. 2).

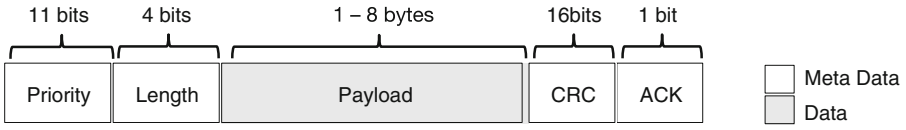

Fig. 2. A single CAN frame.

What makes CAN so widespread, are two important safety requirements it fulfils:

(1) the acknowledgement of reception and
(2) the arbitration of sending order.

[4] Also known as *ESP* - Electronic Stability Program.

	Priority/ Sender ID	Length	Data
Airbag	0x050	4	E0 F0 00 FF
Brake (Front Left)	0x1A0	8	10 F0 01 00 00 30 00 E1

Fig. 3. Example CAN messages from the airbag and brake captured on a 2005 Volkswagen Passat B6.

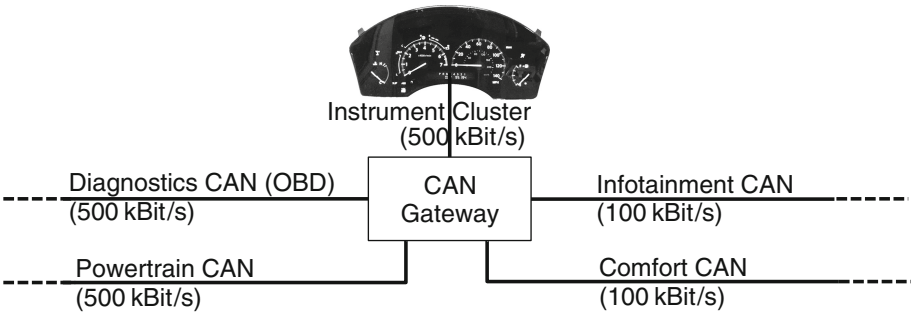

Fig. 4. Interconnected CAN buses through a CAN gateway in a VW Passat B6. (source: "Volkwagen erWin Online")

The acknowledgement of reception (1) is important because the sender of a critical message must be sure that it has been processed. Additionally, messages, which can only be sent one after the other, must be prioritized based on how important they are (2). This ensures the hard real-time requirements of the whole system.

The CAN bus achieves those two properties in hardware by using the electrical concept of so-called *dominant* and *recessive* bits. Bits are always transmitted synchronously at fixed time slots and dominant (logical 0) bits can overwrite recessive bits (logical 1). This simple principle allows the sender to check if at least one ECU has received the frame by reading the acknowledgement bit. Similarly, the prioritization of frames is solved: Since each frame starts with its 11-bit priority, the dominant bits overwrite the recessive priority bits of other frames transmitted at the same time. Only the highest priority frame is received by all the connected nodes, while a lower priority device automatically backs off when a recessive bit has been overwritten by somebody else. The priority is at the same the time CAN sender ID. Hence, if the airbag has sender ID `0x050` and front left brakes have ID `0x1A0`, then the airbag has a *higher* priority because `0x050` is numerically lower than `0x1A0` (see Fig. 3).

CAN buses have different standardized speeds ranging from 5 to 1000 kBit/s. Most common are 500 kBit/s and 100 kBit/s, while 500 kBit/s networks have higher demands in terms of cables and CAN bus tranciever chips. Depending on the make and model, there are several CAN buses in a modern car.

The most prominent reasons for having more than one CAN bus are (1) Clear separation for safety reasons, (2) fault-tolerance in case one bus fails, (3) cost reduction due to lower speed CAN buses where high-speed CAN buses are not needed. An exemplary CAN bus network and its interconnectedness is depicted in Fig. 4.

3 Design

3.1 Problem Statement

The steadily increasing number of components that are connected on the CAN bus introduce a high likelihood that any of such components may be compromised [4]. Unfortunately, such a compromise may have severe security (and therefore also safety) implications to the automotive network. Of all possible threats, message spoofing remains one of the largest unsolved issued on the current CAN bus designs. In the worst case, a compromised component may inject fake CAN messages, e.g., messages that make the parking assistant turn the steering wheel.

In the current CAN design, there is no protection against these threats. First of all, CAN has no scheme to verify the *authenticity* of the messages, i.e., neither the sender information, nor the actual message payload. In principle, an attacker that controls any component on the CAN bus, can thus

(a) spoof the identity of any other component (e.g., to escalate privileges), or
(b) send arbitrary payload (e.g., to perform malicious actions).

Our goal is to introduce authenticity schemes to CAN. First, we aim to add *sender authenticity* to guarantee that CAN components can protect their identity by denying messages that spoof their identity. Second, we plan to add *content authenticity* to guarantee that a message was intentionally sent and its content was neither manipulated nor replayed.

It may seem trivial to redesign CAN such that those features are added. However, a practical solution faces several challenges:

C1 CAN has been designed primarily to match real-time characteristics of the communication. Any security mechanism must not add unacceptable overheads that significantly increase latency or lead to message collisions.
C2 ECUs are typically microcontrollers with very constrained computational power and storage space. Thus, heavy-duty crypto operations cannot be performed as they would add an unacceptable overhead.
C3 CAN messages are limited to 8 bytes, which requires us to either squeeze secure crypto into 64 bits or to use a higher level transmission protocol that re-assembles longer messages that are spread across several CAN frames.
C4 Cryptographic keys must be individual per car to render extracting keys in one car useless. Moreover, key agreement between ECUs must be dynamic to allow for broken parts to be replaced.

C5 Adapting all ECUs and all messages would be disproportionate as many ECUs are non-critical and do not need to be protected. Further, changing messages would result in compatibility problems and enormous costs as mass-produced components could otherwise no longer be used. To keep cost down, as few ECUs as possible should be modified.

C6 Authenticated messages should be immune to replay attacks. However, introducing a global state is against the design principle of CAN, which is stateless in order to tolerate packet loss.

Instead, our goal is to retrofit vatiCAN to CAN by adding a backward-compatible security add-on for selected senders and messages. Our vatiCAN add-on should not influence components that do not support the new security mechanisms. Components that *do* support vatiCAN, on the other hand, will benefit from the authenticity checks. Such a model allows for a gradual, evolutionary change towards a more secure CAN standard and can at the same time protect vital components, such as power steering, brakes and airbag, from the beginning.

3.2 Threat Model

We assume an attacker who does not have physical access to the car but she can fully compromise one (or a few) wireless ECUs that usually use several ID_k $k \in \{0, \ldots, 2^{11}\}$ to send on the CAN bus. The attacker's goal is to impersonate another ECU with ID_x with $k \neq x$. After the compromise of ECU with ID_k, the attacker has full flexibility in sending arbitrary messages to the CAN bus, i.e., she can fake sender identities and chose any message payloads.

The attacker is assumed not to compromise the ECU for which she intends to fake the packets—otherwise the attacker would already be using the genuine sender and can likely extract any cryptographic key material from the compromised devices and thus fake the identity regardless of any cryptographic scheme. Instead, we protect the identities of critical devices that might be impersonated (and not compromised) by an attacker.

In addition, we consider an attacker that can passively monitor the CAN bus. She can observe and record all messages that have been broadcasted on the CAN bus. This way, the attacker can also learn about components' identities.

3.3 High Level Concept

In this section, we describe the individual features of vatiCAN that address the challenges C1 through C5.

Message Organisation (C1, C5). The cryptographic authentication mechanism that vatiCAN uses must be decoupled from the actual message to remain backward compatibility. We opted for a separate message with a different sender ID such that legacy devices still see the original message content from the sender

ID they expect (C5). As a side effect, the induced cryptographic performance and bandwidth overhead only applies to critical messages that have been manually selected at development time. For those selected messages, their additional authentication message is then sent from a different ID for which only vatiCAN-aware recipients listen. The separation of messages also has the advantage that the recipient experiences no delay when receiving the original, unmodified message (C1) and can compute the necessary cryptographic authentication in parallel to the reception of the authentication message.

Special care must be taken when selecting the sender ID for the additional authentication message from ID_j for each legacy sender ID_i. Since senders correspond to priorities on the CAN bus, a careless selection of a new j may result in other messages being delayed if the priority is too high or may lead to the priority inversion problem if the priority is too low. We therefore choose the new $j = i + 1$, which lowers the authentication's priority by the smallest granularity possible. Under the assumption that this new ID is not taken, this effectively assigns the same priority to the authentication message as all other message priorities are still lower or higher, respectively.

Message Authentication (C2, C3). vatiCAN supports content authenticity, which cryptographically ensures that the sender was in possession of the required and correct cryptographic key. As a cryptographic primitive, we chose a light-weight keyed-Hash Message Authentication Code (HMAC) since symmetric cryptography fulfils our requirements and is more suitable for embedded resource-constrained devices with real-time requirements. As underlying hash function for the HMAC construction, we chose the Keccak algorithm that has been standardized as SHA-3. According to the performance evaluation of hash functions on the popular Atmel embedded microcontroller [1], Keccak is the fastest hash function (C2). We chose the Keccak parameters to produce 64 bit output with 128 bits of rate and capacity ($r = c = 128$). The input size of 128 bits was chosen to accommodate the original payload of up to 64 bits, the sender ID and a nonce (see below).

For messages that have been selected for authentication at development time, an additional HMAC CAN frame is sent from the sender j. The recipients can then verify if the HMAC matches the content received earlier, and if so, accept the message. This two-step process can also be used to pre-condition the upcoming action (e.g. move brakes to the disk) as soon as the first un-authenticated, legacy content has been received and then defer the actual command execution (e.g., brake) until the authentication approval arrives. If the HMAC is invalid or has not been received in a given time frame, the recipient can discard the message and potentially issue a warning.

3.4 Replay Attacks (C6)

To prevent **replay attacks**, vatiCAN incorporates a nonce in the HMAC computation. Otherwise an attacker could record a vital message and its HMAC and

then replay both later. In contrast to other authentication schemes, we do not require the nonce to be non-predictable but we require the nonce not to produce two messages with the same HMAC. Since we opted for not modifying the original payload, we cannot distribute the nonce for each authenticated message. To avoid an additional broadcast of the nonce, the sender and all receivers must implicitly agree on the same nonce that must be different for each message. For this purpose, we introduce a counter c_j, which is specific for each sender j and is incremented with each message sent by ID_j. This ensures that the HMAC of recurring CAN payloads is different each time. We chose sender-specific counters rather than a global counter because some ECUs might not be online all the time and most ECUs implement a hardware ID filter that only forwards CAN frames from those IDs that are of interest to this particular ECU.

To account for the fact that messages might get lost or ECUs are temporarily in power-safe mode, we additionally introduce the global Nonce Generator (NG). The NG periodically broadcasts a random global nonce g, which shall be used by all counters c_j as their new value. This way, counters get synced up again across different ECUs. In other words, on each broadcast of g, all ECUs reset their $c_j = g$ and start counting from there with each message they send.

Consequently, the HMAC of the each message m is computed by the sender (recall that i is the ID of the legacy sender, and $j = i+1$ is the ID of the HMAC message) and by the recipient as follows:

$$h = \mathrm{HMAC}(i \mid m \mid c_j)$$

We incorporate the legacy sender i in the HMAC, such that no identical payloads from different senders produce the same HMAC if they happen to share the same key. Since messages on the CAN bus are received in the same order for all nodes, choosing the right g is deterministic for all nodes. The g that was valid before the legacy message from i was sent has to be used to verify the HMAC sent by j in order to avoid race conditions.

The overall timing and message exchange is exemplified in Fig. 5, in which the throttle sends a message that the engine validates. First, the NG broadcasts the new nonce (761). Next, the throttle broadcasts the legacy message. Afterwards, it computes the HMAC, using its legacy ID, the legacy message payload (30) and the nonce. Simultaneously, the engine also computes the HMAC over the message payload sent by the throttle. Finally, after the throttle broadcasts the HMAC, the engine will verify if the computed HMAC values match.

Even though the attacker can receive the current nonce g, she is not in possession of the necessary key to compute the HMAC. In other words, g is not secret.

A nonce introduces a "state" in a protocol that is designed to be stateless to accommodate for packet loss. Should a packet loss occur (e.g., damaged cables) all subsequent HMACs could not be verified anymore until the next global nonce g is broadcasted by the NG. Hence, the interval at which the NG broadcasts g at the same time sets the *deaf* time, which is the maximum time an ECU might see invalid HMACs after a hardware failure. A suggested frequency is every 50 ms, which corresponds to an additional bus utilization of ≈1 % at 500 kBit/s.

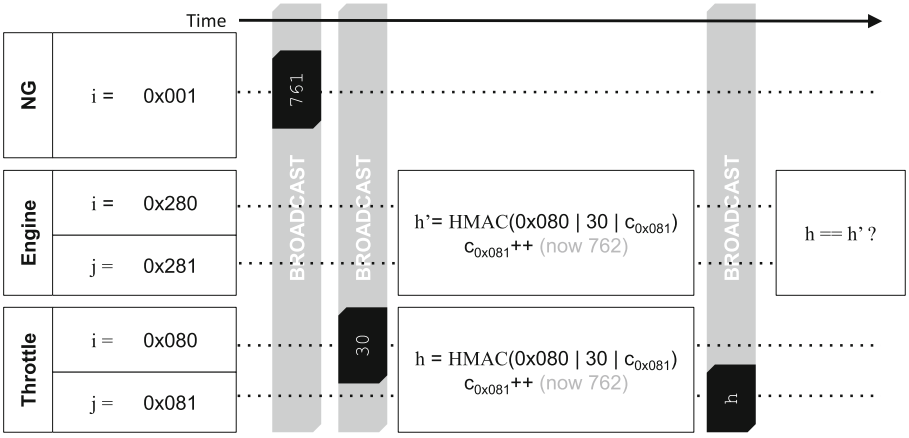

Fig. 5. Timing of interaction between NG, legacy and authenticated messages.

Spoof Detection and Prevention. A first primitive of vatiCAN is to mitigate the risk that compromised components can fake the identity of other components. To this end, we leverage the fact that CAN is a bus-oriented network, and components thus receive messages from all other components on the bus. In fact, if a component monitors the CAN communication, it can identify spoofed messages by monitoring messages with its own sender identification. If a component detects a message with its own sender ID, it must be a spoofed message. Since messages transmitted by a CAN transceiver are not considered received messages, this can be clearly distinguished. Once detected, this issue is made available in software in the form of an exception. However, other recipients might have already processed this message. This software solution is only suitable for detecting and, e.g., displaying a message to the driver. Alternatively, it is also possible to drop a detected spoofed message by intentionally destroying the CRC checksum. However, the CRC checksum part is usually processed in hardware and hence deliberate destruction is only possible when the CAN transceiver chip is modified. A difficulty of this approach is that the CAN messages have to be destroyed before the transmission of the spoofed frame is completed. Luckily, the sender information is at the start of each frame and is synchronously processed for bus arbitration anyway. This means an early detection stage is possible by invalidating the CRC bits using dominant bits (e.g. all zeros) while a CAN frame is still being processed. This approach is similar to the already implemented acknowledgement (see Sect. 2) in the CAN bus standard, which is also set at the correct timing of the *ACK* bit during transmission. We assume that (at least) the NG—which is the only component that needs to be added for vatiCAN—is protected with such a hardware-assisted spoofing prevention mechanism. Hence an attacker cannot impersonate the NG and inject arbitrary nonces.

Should a sender ID be shared between ECUs, the spoof detection described above cannot be applied. A shared sender ID could be used for door ECUs that simply send a message that the door is open. Each door has an ECU that uses the same sender ID because it only matters *that* a door is open but not which.

Key Distribution (C4). According to the HMAC construction, the used cryptographic key is either padded to the length of the hash function's input block size, or it is hashed if it is longer than the block size. To avoid an additional hashing operation, we recommend setting the length of the cryptographic key exactly to the length of the input block size of the hash function: 128 bits.

We also chose not to use one global key, but individual keys per ECU. Of course, it is also possible to group ECUs that share the same key. This saves precious flash memory at the expense of reduced security. Typically, ECUs form logical clusters, e.g., all four wheel rotation sensor ECUs, and all four brake ECUs form a logical cluster of *traction control*. vatiCAN leverages this and supports assigning sets of IDs the same cryptographic key, bonding them to a group.

The most critical aspect is the key provisioning: the process of getting keys into ECUs in the first place. The cryptographic key for each ECU or group needs to be provisioned to each ECU that is part of that logical cluster. Generally, two possibilities exist:

(a) **During Assembly.** The keys could be generated randomly during production and automatically be injected into the flash memory of the corresponding ECUs. However, this makes replacing ECUs after a fault or accident more involved, as either keys have to be extracted from other ECUs or new keys need to be generated and distributed to all clusters that the faulty ECU communicates with.
(b) **Key Agreement.** Alternatively, keys can also be agreed upon using Diffie-Hellman key exchange every time the car is turned on. However, this option has several disadvantages: ECUs that switch on on-demand have to re-run the key agreement. Moreover, man-in-middle attacks are possible without certificates stored in the ECUs, which is not practical. And lastly, multi-party key exchange is non-trivial for an embedded microcontroller.

This is why we chose option (a) to provision the keys during production of an ECU. In case an ECU needs to be replaced, all other ECUs need to be updated with the new key. Luckily, software updates through the on-board diagnostics (OBD) port are commonplace and supported by most ECUs. This allows for re-programming of keys without physically removing the ECUs from the car. To protect against malicious key updates by compromised ECUs, the key provisioning could be protected using asymmetric cryptography. For example, signed updates are a viable option, despite the fact that they are relatively slow.

4 Implementation

We implemented a proof-of-concept of vatiCAN on the popular Atmel AVR microcontrollers and used off-the-shelf automotive components, such as an

instrument cluster, that act as legacy devices. Our implementation is also available as download in the form of a library for the popular Arduino development environment (see Appendix A).

4.1 Hardware Platform

As hardware platform, we used Atmel AVR microcontrollers, each of which is connected to a Microchip MCP2515 CAN bus controller over SPI [11]. The CAN bus is built on a work bench top and connects an off-the-shelf automobile instrument cluster as symbolic legacy device. The bench top CAN test network resembles a *Hardware-in-the-Loop-Test* (HIL), which is common in the automobile industry [5]. We used this hardware to build the prototype (see Fig. 6):

Atmel AVR ATMega328p	8 bit microcontroller (32 kB of flash, 2 kB of SRAM, 16 MHz)
Microchip MCP2515	CAN bus controller chip (3 TX / 2 RX buffers)
Seat Ibiza instrument cluster	From the 2009 model 6J
Logitech Formula GP	Accelerator and brake pedals

Components of the topology of the HIL setup shown in Fig. 7 are:

(a) Electronic Throttle Control (ETC)
(b) Powertrain Control Module (PCM)
(c) Instrument cluster (speed, RPM, airbag warning, ABS warning).
(d) Auxiliary Simulator (AS) for airbag, brakes, and wheels.

The ETC reads the analog potentiometer mounted to the accelerator pedal (0 % to 100 % pressed) and broadcasts the value on the CAN bus so that it can be interpreted by the PCM. The PCM in turn simulates an internal combustion engine and broadcasts engine RPM and oil temperature on the CAN bus. The Seat Ibiza instrument cluster shows the engine RPM using an analog dial.

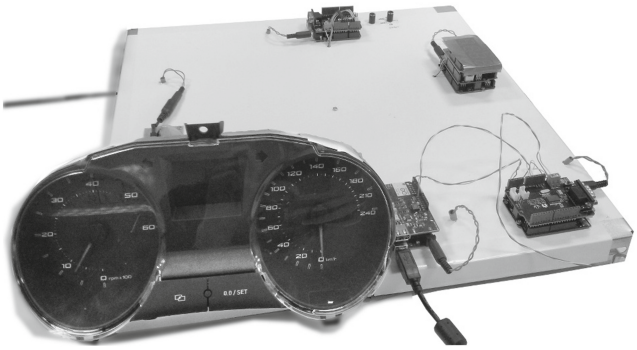

Fig. 6. Bench HIL setup with original instrument cluster ECU and re-engineered ETC and PCM ECUs.

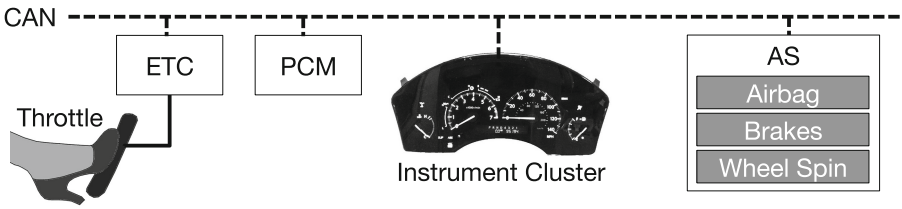

Fig. 7. Hardware-in-the-Loop Test (HIL): ETC, PCM and AS.

The corresponding speed that the speedometer dial shows is being sent by the AS and calculated from the engine RPM and currently selected (simulated) gear.

4.2 Secure Message Selection

The strong-suit of vatiCAN is its interoperability and backward-compatibility with legacy devices that do not understand authenticated messages. For this reason, we chose the original instrument cluster from a 2009 Seat Ibiza, which shows speed and engine RPM despite being secured by vatiCAN. Not all exchanged messages in the HIL were secured, because not all messages are vital. We chose to secure the (1) throttle position message, (2) engine RPM, (3), wheel rotation (vehicle speed), and (4) anti-lock brake controller

All the other messages needed to operate properly (see Appendix B) are not authenticated as they are not vital.

4.3 Software Architecture

The vatiCAN library abstracts CAN bus access to sending and receiving messages, while received messages incorporate the notion of being authenticated or not. The application using vatiCAN registers known sender IDs for authenticated messages and two callbacks. One callback for receiving messages (legacy and authenticated) and one for errors (authentication mismatch, timeouts).

For this purpose, the vatiCAN library keeps a list of authenticated sender IDs i and thus can perform a look-up based on the sender ID for every received CAN frame. Then, vatiCAN knows whether to expect an additional authentication CAN frame from j. All CAN frames are delivered immediately to the application using the provided call-backs. However, CAN bus frames originating from senders not in the list of authenticated senders are flagged *insecure* while frames originating from senders that should authenticate their messages are flagged as *not authenticated yet*. If authentication messages are expected, the HMAC calculation is started in the background. The application code using vatiCAN can then decide whether to prepare or pre-compute intermediate steps until the authentication message arrived and was verified. If the authentication message arrives, vatiCAN automatically compares the computed HMAC to the received authentication message and either invokes the message reception

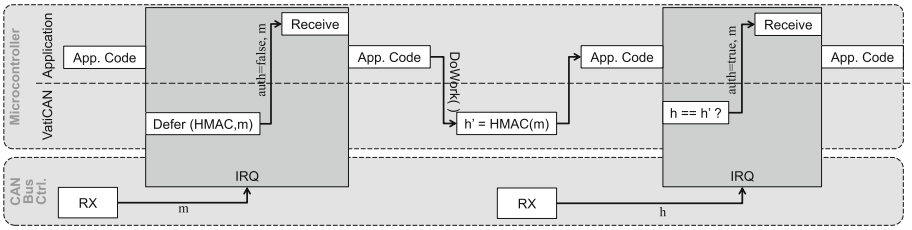

Fig. 8. CAN bus frame reception, message processing and the application.

call-back indicating an authenticated message, or invokes the error call-back if the HMAC comparison failed. The message verification is designed in such a way that the order of internal HMAC computation and authentication message reception does not matter. Whichever completes last, triggers the comparison and forwards either the message or an error to the application.

Since AVR ATmega microcontrollers do not support hardware multi-threading, the background HMAC computation is implemented as an interrupt service routine (ISR), which is triggered on CAN frame reception and defers computations to a later point indicated by the application code. The interaction diagram shown in Fig. 8 depicts the processing of a single, authenticated message.

The implemented code consists of a vatiCAN CAN bus interface library written in C++ and a hash function that currently uses the Keccak (SHA-3) implementation in assembler that was adapted from [1]. The C++ library consists of only 314 lines of code, while the hash function in assembler consists of 250 lines of code. The compiled code sizes can be found in the evaluation in Sect. 5.

5 Performance Evaluation

We evaluated the performance in terms of message reception delay, bus congestion due to added CAN frames and in terms of memory footprint. The evaluation was conducted on the hardware presented in Sect. 4. To obtain time measurements, we used the internal ATmega timer with a prescaler of 64, which divides the used 16 Mhz clock speed in 250 kHz accuracy (4 μs accuracy). All experiments have been conducted on the very common 500 kBit/s bus speed.

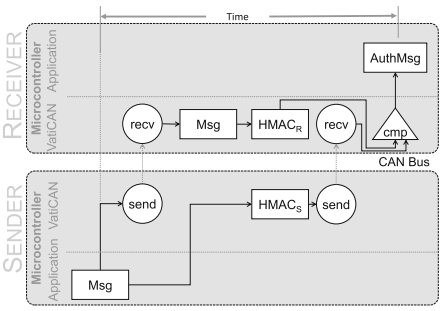

Fig. 9. Parallel computation of HMACs

We then measured the time it takes to calculate a single HMAC for a CAN frame, given *nonce* and sender ID as additional input. The HMAC calculation

takes about 47,600 clock cycles or 2.95 ms for the used clock speed of 16 Mhz. The total time the reception of a message is deferred due to the calculation of the HMAC and comparison with the received authentication CAN frame is 3.3 ms. That means, the look-up if the sender ID is in the list of secure senders plus the string comparison of calculated and received HMAC make up for 0.35 ms. Note that the application gets notified immediately after reception of the payload and can start precomputations. This means that both the sender and receiver can compute the HMAC of the payload *in parallel*.

Figure 9 illustrates the parallel computation. $HMAC_S$ is the sender's computation of the HMAC including the currently valid nonce, while $HMAC_R$ is the receiver's computation of the received message Msg. The HMAC computations take place simultaneously on the receiver's and sender's side, as the receiver starts computing the HMAC as soon as the plain text message Msg arrives. The receiver then compares $HMAC_R$ against $HMAC_S$ to check if they match. This parallel computation is a major benefit of HMAC compared to asymmetric cryptographic message signatures, for which the receiver has to wait for the signature before further validations.

Next, we measure the round-trip time for legacy vs. vatiCAN-secured CAN frames. In case of legacy frames, one microcontroller broadcasts an 8-byte CAN frame and another microcontroller receives the message and immediately broadcasts another message. The time measured is the interval between sending the first message and after receiving the response. For plain, unauthenticated 8-byte CAN frames, the ping-pong time interval is 1.08 ms and consequently 2 messages were exchanged in total. For vatiCAN, 2 messages have to be sent and 2 messages have to be received. Both microcontrollers must calculate 2 HMACs (one for sending, one for verification). The total time between sending the secure message until after reception of the secure response is 4.5 ms.

Please note that the used ATmega 8 bit microcontrollers represent the lower bound of an automotive performance evaluation. The common v850 32 bit microcontrollers offer ≈2.6× the performance.

5.1 Bus Congestion

For safety reasons, it is important to know the limits of the CAN bus network in terms of throughput in order to ensure that no messages get lost. Car manufacturers also use HIL tests to ensure a safety margin such that under all conceivable conditions the maximum bandwidth and thus the maximum intended latency is never exceeded. To measure the typical utilization of a heavily used bus, we chose the 500 kBit/s instrument cluster CAN (see Fig. 4) because it combines messages from the powertrain CAN bus, the infotainment CAN bus, and the comfort CAN bus. Appendix B lists all recorded messages and their frequency of occurrence in the VW Passat B6.

Due to the re-occurring nature of CAN messages, every 100 ms the same messages were seen on the bus. Per second, 560 messages are sent with a total payload of 4,230 bytes (33,840 bits). Due to the CAN frame overhead (start bit, length bits, CRC, stuff bits etc.) each frame needs an additional 47 bits to

be transmitted. Hence, per second a total of $33,840 + 560 \cdot 47 = 60,160$ bits are transmitted. We tested the maximum possible bandwidth under realistic conditions by flooding the bus with 8-byte CAN frames. Counting whole CAN frames (payload + header bits) we achieved a throughput of 448 kBit/s. Thus, the measured utilization of 60.2 kBit/s corresponds to 13.4 % utilization. With 3 out of 13 senders protected by vatiCAN, per second 110 messages of the 560 total messages are protected. This accounts for additional $110 \cdot (47 + 64)$ bits = 12,210 bits. Thus, the total bus utilization increases to 72.4 kBit/s (16.2 %).

5.2 Memory Footprint

The total vatiCAN library size is 2152 bytes of AVR instructions of which 678 bytes are attributed to the Keccak implementation and the remaining 1474 bytes are the surrounding vatiCAN message verification, HMAC and interrupt logic. In addition, vatiCAN has to store an additional 32-bit word for the sender's nonce (4 byte) per sender ID. Even in the unlikely case that an ECU expects 100 different vatiCAN sender IDs, this would result in mere 100*4=400 bytes.

6 Security Evaluation

The goal of an attacker is to inject a specific, potentially dangerous CAN frame and to forge its HMAC. Since the attacker needs to forge an HMAC for one specific message (or a few specific messages), it does not suffice to find an arbitrary collision in the underlying hash. Instead, the attacker's goal is to find a concrete collision or the actual cryptographic key.

We chose the Keccak (SHA-3) parameters ($r = c = 128$, $n = 64$) such that it projects its input to 64 bits (8 byte CAN frame) output. While an output size of merely 64 bits is significantly shorter than the typical length of SHA3's 224 bits, the increased advantage of the attacker is offset by the limited validity of a message due to the cyclic message nature of CAN bus and the invalidation through the Nonce Generator NG.

The cryptographic strength of the used HMAC construction depends on the length of the secret key and on the chosen output size. An attacker could record a payload message and its corresponding HMAC. Using the known sender j and the calculated nonce c_j, she can then brute-force all possible keys until she finds an input that matches the recorded HMAC. Because of the fixed 128 bit key, an attacker would need 2^{127} tries on average. Hence, the success probability $P_{key} = 2^{-127}$. The other option, to guess the output of the HMAC correctly is $P_{out} = 2^{-63}$ due to the 64 bit output length. Please note that the birthday paradox in finding an arbitrary hash collision does not apply here, since the attacker has to match a specific plaintext legacy payload. On the Atmega328p running at 16 MHz, the computation of 2^{32} HMACs would need $2^{32} \cdot 2.95\ ms = 12,670,154\ s = 146$ days, which is well outside the validity period set by the NG. Although a faster ECU could brute-force the HMAC quicker, this is likely not

fast enough. Even though a dual-core 32-bit ARM 1 GHz (e.g., the infotainment system) would be about 100x faster, it still takes 24 h to brute-force for a nonce update interval of 50 ms.

It should be considered that an attacker might successfully compromise an ECU on which a key is stored that is used for vatiCAN. If keys are grouped and used on multiple ECUs, the attacker can use this key to generate valid HMACs for any sender to which the group key belongs.

7 Related Work

The first paper that extensively demonstrated practical vulnerabilities of a modern automobile [10] was published in 2010 and has been cited many times in academia and the press since. The authors demonstrated that it is possible to inject code into ECUs, which are connected to various CAN buses. Further, they demonstrated that bridging the CAN gateway is possible, effectively connecting a less-critical to a highly-critical CAN bus. Further, they demonstrated that even remote attacks exist that do not require physical access to the car [2].

Despite the ECUs being the culprit in terms of vulnerabilities, the underlying CAN bus makes a life-threatening attack feasible since a compromised ECU may affect any other connected system. In recent years, several authentication methods for broadcast buses have been introduced. The closest related work to vatiCAN is *CANAuth* [16]. CANAuth proposes a similar HMAC-based authentication scheme, however, their goal was to incorporate the HMAC into the payload CAN frame itself. They achieve this by basing their solution on CAN+, a physical layer modification of the CAN protocol to achieve higher data rates [18]. Since the additional CAN+ bits are stuffed in-between legacy CAN bits, it is backward-compatible to CAN controllers, which do not support CAN+. However, CAN+ would require new hardware to be used for the nodes that should support CANAuth, and no such hardware exists yet. Our goal was to update software only and to re-use existing hardware and CAN controllers.

Also purely theoretical work on the topic of CAN bus sender authentication exists [17] that formalizes which cryptographic primitives are required to guarantee secure communication between different ECUs – even across different buses. The authors consider key distribution, PKIs and encrypted communication. In contrast to our solution, their sender authenticity is proven using signature, i.e. asymmetric cryptography. A similar solution to our spoofed message detection has been presented in [12]. In contrast to invalidating the spoofed message by destroying the CRC, the authors detect the spoofed message and immediately send an error frame on the bus.

More general, the TESLA protocols [14,15] are designed to authenticate a broadcast sender with symmetric cryptography. However, they use delayed disclosure of keys, i.e. the sender uses uses a symmetric key for the HMAC that nobody else knows. Consequently, at reception time, no receiver is able to authenticate the packet until the key will be made available in a later packet. This clearly violates our real-time challenge C1. While the improved version of

TESLA [14] supports immediate disclosure of the key, each packet incorporates a hash of the succeeding packet to build a chain. This is clearly unsuited for highly lively but predictable CAN bus traffic.

Finally, the AUTOSAR standard [6] also supports an HMAC-based message authentication scheme. In contrast to vatiCAN, Autosar is not backward compatible, as Autosar uses higher level communication (PDUs) to which an HMAC is appended. Moreover, Autosar does not support spoofing prevention.

8 Limitations and Future Work

Due to the rather restrictive payload of 8 bytes maximum, several protocols have emerged that build on top of CAN to implement higher layers, such as longer payloads and transmission control. Popular examples are KWP2000 [9] or ISO TP [8], which are commonly used for software updates and ECU diagnostics. Using vatiCAN, especially for software updates originating from outside the vehicle, makes sense. However, the current implementation which authenticates every single CAN frame would induce an impractical bandwidth overhead. A more elegant solution would be to authenticate the payload on the KWP2000 or ISO TP layer by attaching a digital signature.

While the achieved latency of only 3.3 ms on a simple microcontroller is seemingly fast, for high motorway speeds, a few milliseconds make a difference between life and death. Should vatiCAN be applied to active safety functions of a car (e.g., collision avoidance through active braking), the induced latency of 3.3 ms results in a traveling distance of 0.9 m at motorway speed of 100 $\frac{km}{h}$.

9 Conclusion

The adaptation of new technology in the automobile sector is a cautious and slow process. It is therefore important to change only a few parts, while the established and reliable majority of components can be re-used. Therefore, vatiCAN is designed to be backward-compatible to allow tried and trusted components to rely on the same CAN messages without need for modification. However, those parts for which a manufacturer decides to enhance security can be easily protected by means of a software upgrade, which uses vatiCAN instead of another CAN bus interface library. Our vatiCAN implementation is able to deliver real-time protection to ensure that a compromised ECU cannot be leveraged to fake messages, which are potentially life-threatening. The induced latency of 3.3 ms for authenticated messages is fast enough for most situations and shows the practicality and feasibility of the approach. However, for highly timing-critical functions, such as brakes, a millisecond delay might be unacceptable.

While the presented results should encourage automakers to implement what is currently possible given a dated CAN bus architecture, it also shows the need for a novel design to achieve stronger security claims and better performance.

Acknowledgments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

A Availability

Our vatiCAN implementation is available as free software download published under the LGPL v2. We provide a library for the popular Arduino development environment for Atmel's AVR microcontrollers. Its source code is publicly available at http://automotive-security.net/securecan

B VW Passat B6 CAN Messages

The following messages were captured on the instrument cluster CAN bus on a VW Passat B6 and are reproduced in the HIL to have realistic bus utilization.

Function	CAN ID	Every	Frequ.	Length	Bytes/s	Exemplary Payload	vatiCAN
Airbag	050	20 ms	50 Hz	4 bytes	200	E0:F0:00:FF	⊘
Steering	0C2	20 ms	50 Hz	8 bytes	400	F0:00:00:00:80:40:00:CF	⊘
Electronic Power Steering (EPS)	0D0	50 ms	20 Hz	6 bytes	120	D7:C0:61:08:5E:20	⊘
ABS1	1A0	10 ms	100 Hz	8 bytes	800	00:00:00:00:FE:FE:00:00	⊘
ABS2	4A0	10 ms	100 Hz	8 bytes	800	00:00:00:00:FE:FE:00:00	⊘
Brakes	1AC	20 ms	50 Hz	8 bytes	400	00:80:7F:7F:69:A1:00:C2	☑
ETC / Engine RPM	280	20 ms	50 Hz	8 bytes	400	49:00:20:20:00:FA:36:00	☑
Engine Status	35B	100 ms	10 Hz	8 bytes	80	0F:00:00:B8:28:19:02:96	⊘
Coolant	288	20 ms	50 Hz	8 bytes	400	43:78:00:04:00:56:00:00	⊘
Instrument Cluster	320	20 ms	50 Hz	8 bytes	400	02:00:00:ff:ff:cd:ff:96	⊘
Vehicle Speed	5A0	100 ms	10 Hz	8 bytes	80	00:00:00:5B:B6	☑
Instrument Cluster	621	100 ms	10 Hz	7 bytes	70	04:00:01:00:02:00:00	⊘
Instrument Cluster	727	100 ms	10 Hz	8 bytes	80	02:00:00:ff:ff:c6:ff:9e	⊘

Throughput (net)	**33840**	bits/s
Throughput (overhead)	**26320**	bits/s
Throughput (gross)	**60160**	bits/s
Average Utilization	**13.4%**	

References

1. Balasch, J., et al.: Compact implementation and performance evaluation of hash functions in ATtiny devices. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 158–172. Springer, Heidelberg (2013)
2. Checkoway, S., McCoy, D., Kantor, B., Anderson, D.,Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: USENIX Security Symposium (2011)
3. Dr. Ing. h.c. F. Porsche Aktiengesellschaft: Annual report 2004/2005. http://www.porsche.com/filestore.aspx/default.pdf?pool=uk&type=download&id=annualreport-200405&lang=none&filetype=default
4. Ebert, C., Jones, C.: Embedded software: facts, figures, and future. Computer **4**, 42–52 (2009)
5. Hanselmann, H.: Hardware-in-the loop simulation as a standard approach for development, customization, and production test of ECUs. Technical report, SAE Technical Paper (1993)

6. AUTOSAR Specifications 4.2 (2016). http://autosar.org
7. ISO. ISO 11898-1:2003 Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling. International Organization for Standardization (ISO), Geneva (1993)
8. ISO. ISO/DIS 15765-2 Road Vehicles – Diagnostic Communication Over Controller Area Network (DoCAN) – Part 2: Transport Protocol and Network Layer Services. International Organization for Standardization (ISO), Geneva (2011)
9. ISO. ISO 14230-2:2013 Road Vehicles – Diagnostic Communication Over K-Line (DoK-Line) – Part 2: Data Link Layer. International Organization for Standardization (ISO), Geneva (2013)
10. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al.: Experimental security analysis of a modern automobile. In: IEEE Symposium on Security and Privacy, pp. 447–462 (2010)
11. Leens, F.: An introduction to I2C and SPI protocols. IEEE Instrum. Meas. Mag. **12**(1), 8–13 (2009)
12. Matsumoto, T., Hata, M., Tanabe, M., Yoshioka, K., Oishi, K.: A method of preventing unauthorized data transmission in controller area network. In: Vehicular Technology Conference (VTC), pp. 1–5. IEEE (2012)
13. Navet, N., Simonot-Lion, F.: Automotive embedded systems handbook, CRC Press (2008)
14. Perrig, A., Canetti, R., Song, D., Tygar, J.D.: Efficient and secure source authentication for multicast. Netw. Distrib. Syst. Secur. Symp. (NDSS) **1**, 35–46 (2001)
15. Perrig, A., Canetti, R., Tygar, J.D., Song, D.: Efficient authentication and signing of multicast streams over lossy channels. In: IEEE Symposium on Security and Privacy, pp. 56–73. IEEE (2000)
16. Van Herrewege, A., Singelee, D., Verbauwhede, I.: CANAuth – a simple, backward compatible broadcast authentication protocol for CAN bus. In: 2011 ECRYPT Workshop on Lightweight Cryptography (2011)
17. Wolf, M., Weimerskirch, A., Paar, C.: Security in automotive bus systems. In: Proceedings of the Workshop on Embedded Security in Cars (ESCAR) (2004)
18. Ziermann, T., Wildermann, S., 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, DATE 2009, pp. 1088–1093. IEEE (2009)