S2-CAN: Sufficiently Secure Controller Area Network

Mert D. Pesé, Jay W. Schauer, Junhui Li, and Kang G. Shin
The University of Michigan
Ann Arbor, MI 48109-2121, USA
{mpese,jschauer,opheelia,kgshin}@umich.edu

ABSTRACT

As automotive security concerns are rising, the Controller Area Network (CAN) — the *de facto* standard of in-vehicle communication protocol — has come under scrutiny due to its lack of encryption and authentication. Several vulnerabilities, such as eavesdropping, spoofing, and replay attacks, have shown that the current implementation needs to be extended. Both academic and commercial solutions for a Secure CAN (S-CAN) have been proposed, but OEMs have not yet integrated them into their products. The main reasons for this lack of adoption are their heavy use of limited computational resources in the vehicle, increased latency that can lead to missed deadlines for safety-critical messages, as well as insufficient space available in a CAN frame to include a Message Authentication Code (MAC).

By making a trade-off between security and performance, we develop `S2-CAN`, which overcomes the aforementioned problems of S-CAN. We leverage protocol-specific properties of CAN instead of using cryptographic primitives and design a "sufficiently secure" alternative CAN with minimal overhead on resources and latency. We evaluate the security of `S2-CAN` in four real-world vehicles by an automated vehicular attack tool. We finally show that CAN security can be guaranteed by the correct choice of a design parameter while achieving acceptable performance.

ACM Reference Format:
Mert D. Pesé, Jay W. Schauer, Junhui Li, and Kang G. Shin. 2021. S2-CAN: Sufficiently Secure Controller Area Network. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485832.3485883

1 INTRODUCTION

Since the advent of the first comprehensive automotive security analysis in 2010 [11, 27], this field has attracted significant attention. While the first generation of vehicle security (c. 2010-2015) focused on exploiting physical interfaces, such as the OBD-II port [31], or reverse-engineering Electronic Control Unit (ECU) firmware [29], the second generation (c. 2015-now) has been focusing on scaling attacks to multiple vehicles by analyzing remote attack surfaces [30]. The most prominent and comprehensive attack of this generation that led automotive cyber security to become a mainstream research and engineering subject was the Jeep Hack [32] that allowed the attacker to remotely compromise and steer the affected vehicles. With further scaling in each generation, the risk of automotive vulnerabilities towards driver/passenger safety and privacy, as well as financial and operational damage potential increases [24]. All attacks in each generation have (CAN) injection/spoofing as the necessary (final) component of causing havoc in common. This enables the compromise of the vehicle which can, in the worst case, have a serious impact on driver safety, for instance, by electronically disabling the brakes or accelerating the vehicle.

Unfortunately, CAN injection is the easiest part of the aforementioned attacks. This can be explained by vulnerabilities in the CAN design which dates back to 1987. Despite allowing a fast, robust, and reliable communication, CAN was not designed with security in mind, and vehicles can no longer be regarded as closed systems due to an increased number of external interfaces with unpredictable input. CAN is a broadcast bus without encryption and authentication. Messages are sent in plain text and everyone who has access to the CAN bus can inject arbitrary messages or spoof existing ones. Encryption and authentication in a vehicle should usually go hand in hand. In order for spoofed messages to cause a visible impact on the compromised vehicle, the attacker needs to **(a)** know the syntax and semantics of the crafted CAN payload, and **(b)** be allowed to inject the targeted CAN message. In case of **(a)**, this is only possible by reverse-engineering unencrypted CAN data traces since OEMs keep the aforementioned semantics secret instead of disclosing them publicly (*security by obscurity*). Recently, automated CAN reverse-engineering is shown to be achievable in a few minutes [36], enforcing existing attack vectors and necessitating an encrypted CAN. Finally, for case **(b)**, authentication will prevent unauthorized entities to perform the CAN injection.

To defend against vehicular attacks, we need a holistic, multilayer security approach. The authors of [45] propose 4 layers of countermeasures which build on one another: access control, secure on-board communication, data-usage policies and anomaly detection/prevention (see Sec. 3). Here we assume OEMs follow basic security practices such as access control and focus on the challenges of secure on-board communication. As we discuss in Sec. 4, many researchers have attempted to apply the security properties of confidentiality, authenticity, integrity, freshness, and availability on the CAN bus. Almost all of them provide authentication and replay protection — but no encryption — by deploying well-studied cryptographic algorithms. A comparison of existing approaches is provided in Table 1.

Although mechanisms such as encryption and authentication are widely used and accepted in traditional computer communication networks, their adoption in the automotive domain comes with three major problems related to performance that currently limit their deployment in commercial vehicles:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '21, December 6–10, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485883

Table 1: Comparison with related approaches

	Protection	Algorithm	HW/SW	Bus Load	Latency	MAC Length	Security Level
CaCAN [28]	Authenticity + Freshness	SHA256-HMAC	HW+SW	+100%	+2.2-3.2μs	1 Byte	2^7
IA-CAN [21]	Authenticity	Randomized CAN ID + CMAC	SW	+0%	8bit: +72ms 32bit: +150μs	1-4 Bytes	2^7-2^{31}
vatiCAN [33]	Authenticity + Freshness	SHA3-HMAC	SW	+16.2%	+3.3ms	8 Bytes	2^{63}
TESLA [34]	Authenticity + Freshness	PRF+HMAC	SW	+0%	+500ms	10 Bytes	2^{79}
LeiA [37]	Authenticity + Freshness	MAC	SW	+100%	N/A	8 Bytes	2^{63}
CANAuth [41]	Authenticity + Freshness	HMAC	HW+SW	+0%	N/A	10 Bytes	2^{79}
S2-CAN	Confidentiality + Authenticity + Freshness	Circular Shift + Internal ID Match	SW	+0%	+75μs	N/A	$\sim 2^{49}$

(1) Cost: For cost reasons, ECUs in an in-vehicle network (IVN) are resource-constrained. Since most safety-critical functionalities require simple computations and do not need high-performance hardware, these legacy ECUs are very simple and highly optimized for repetitive control operations. For instance, current Engine Control Modules can have 80MHz clock frequency, 1.5MB Flash memory and 72kB of RAM (Bosch [4]). Using cryptographic algorithms for encryption and/or authentication would require more performant hardware which drive up the cost for OEMs. Besides unit costs, adding security protocols to certain legacy ECUs (especially in the powertrain domain) that have been in use in cars for multiple years or even decades due to lack of necessary software improvements would increase the development cost [40].
(2) Latency: In order to guarantee functional safety in a vehicle, there are stringent hard real-time requirements for certain safety-critical control data. The maximum permitted end-to-end (E2E) latency for cyclic control data transmitted on the CAN bus can range from a few milliseconds to a second [16]. Since secure encryption and authentication algorithms add a non-negligible delay (see Sec. 7), as well as block CAN messages to be sent until fully encrypted (due to block size), message deadlines can be missed which can endanger driver safety (imagine the car braking too late!).
(3) Bus Load: CAN messages contain only 8 bytes of payload. Message Authentication Codes (MACs) to protect data integrity have to be appended to the data, but due to lack of space, several existing solutions [28, 33, 37, 42] send the MAC in a separate CAN message. This increases the bus load which is an indicator for the utilization of the network. A high bus load can lead to certain CAN messages missing their (hard) deadlines, harming safety. To avoid this, the average bus load must be kept under 80% at all times [3].

For the above reasons, encryption and authentication on the CAN bus have not yet been adopted in commercial vehicles. Traditional cryptography-based solutions — we will summarize these under the generic term ***Secure CAN*** (S-CAN) — offer a medium to high level of security (see the number of combinations to brute-force MAC, labeled as *Security Level*, in Table 1) at the expense of performance (i.e., CPU, latency, bus load). As the authors of [33] have shown, brute-forcing a MAC would take too long for in-vehicle ECUs, especially if keys are dynamically refreshed. As a result, we would like to break away from traditional cryptography-based solutions to address the aforementioned three problems while providing reasonable, albeit relaxed security guarantees. We propose S2-CAN (**S**ufficiently **S**ecure **CAN**) to enable a **tradeoff** between *performance* and *security* to offer a feasible and secure real-world solution for the automotive industry.

S2-CAN tries to protect the confidentiality, authenticity and freshness of CAN data during operation of the vehicle without using cryptography. In particular, S2-CAN consists of two phases in its core: a handshake and operation phase. In the former, it establishes unique sessions of specific length and distributes necessary *session parameters* to all participating ECUs. This phase resembles the key management phase in traditional S-CAN approaches where session keys are shared among the ECUs to both encrypt and authenticate CAN messages in their respective operation phase. Since S2-CAN avoids using cryptography in its operation phase, it uses the *session parameters* from the handshake to **(a)** first include a randomly generated *internal ID* and counter for authenticity and freshness into the CAN payload before **(b)** each byte of the payload is shifted cyclically by a random integer (*encoding parameter*) in fixed time intervals. These two steps can be compared to **(a)** appending a MAC to provide authenticity and **(b)** encrypting the plain-text CAN message to provide confidentiality in S-CAN. Compared to breaking traditional CAN authentication solutions that only require brute-forcing the MAC, the cyclic shift encoding further masks the plain-text by making it more difficult to decode and thus provides confidentiality protection as well. Due to the encoding, CAN reverse-engineering — which is the first essential step of a CAN injection attack — has to be performed in real time for the current *encoding parameter* and cannot be computed *a priori* to be used for the lifetime of the vehicle. Despite intentional weaker security of S2-CAN, a frequent update of sessions with new *encoding parameters* will render reverse-engineering very tedious, if not impossible. Hence, session cycle is *the* crucial parameter to provide security in S2-CAN. Furthermore, even after guessing the encoding correctly, an attacker would still need to calculate the

internal ID and counter to bypass authentication. All in all, brute-forcing S2-CAN would require $\sim 2^{49}$ combinations for an ECU (see Sec. 8 and 9) while it does not increase the bus load in the operation phase, outperforms the E2E latency of the best comparable S-CAN approach by 44x, as well as incurs less than 0.1% CPU overhead as evaluated with our experimental setup (see Sec. 7). Finally, we conduct a security evaluation in Sec. 8 to demonstrate that even an intelligent attacker who leverages protocol-specific knowledge to circumvent brute-forcing can be thwarted to show that S2-CAN can be sufficiently secure.

2 CAN PRIMER

Vehicular sensor data is collected from in-vehicle ECUs. The latter are typically interconnected via an in-vehicle network (IVN), with the CAN bus being the most widely-deployed bus topology. Fig. 1 depicts the structure of the most common CAN 2.0A data frame.

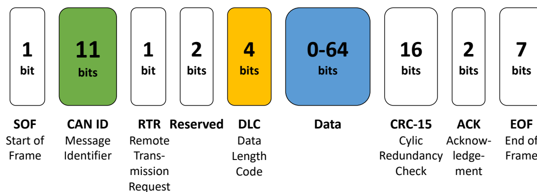

Figure 1: CAN2.0A data frame structure (adapted from [36])

The colored three fields are essential for the understanding of CAN:

- **CAN ID:** CAN is a multi-master, message-based broadcast bus that is message-oriented. CAN frames do not contain any information regarding their source or destination ECUs, but instead each frame carries a unique message ID that represents its meaning and priority. Lower CAN IDs represent higher priority or criticality.
- **DLC and Data:** Data is the payload field of a CAN message containing the actual message data of length of 0–8 bytes depending on the value of the DLC field.

The payload field consists of one or more "signals," each representing information like vehicle speed. Messages transmitted with the same CAN ID usually contain related signals. Raw CAN data is not encoded in a human-readable format and does not reflect the actual sensor values. In order to obtain the actual sensor values, raw CAN data must first be decoded [13]. Letting r_s, m_s, t_s, and d_s be the raw value, scale, offset, and decoded value of sensor s, respectively, the actual value can be determined as follows:

$$d_s = m_s \cdot r_s + t_s. \quad (1)$$

All recorded CAN data can only be interpreted using the translation tables for that particular vehicle. The most common format used for this purpose is DBC [18] which contains information about available signals in each CAN ID, their scale and offset, as well the senders and receivers of CAN messages. In order to execute a successful spoofing attack (i.e., with a visible outcome towards vehicle operation), the CAN payload has to be carefully crafted by the attacker. As a result, an adversary needs to determine the scale and offset for the CAN signal they want to spoof. Furthermore, some DBCs store if a CAN message is periodic (including its cycle time) or sporadic. Note that in the remainder of this paper, we will only modify the CAN payload/data field and **NOT** the CAN ID to preserve backward-compatibility and not interfere with schedulability.

There are multiple CAN buses (e.g., powertrain, infotainment) in the vehicle that are separated via a gateway ECU. It is possible to physically tap into any CAN bus domain (after removing plastic compartments) by using an Arduino with a CAN bus shield [47]. Another — more realistic — way of accessing the CAN bus is the on-board diagnostics (OBD-II) interface under the steering wheel which is mandatory for all gasoline cars in the US since 1996. OBD-II tools are manifold and cheap [6, 19]. Theoretically, it is possible to read and write the CAN traffic on all in-vehicle buses through the OBD-II interface. In practice, however, not all buses are mirrored out to it. This can be explained by access control [35] that OEMs implement. Nevertheless, previous literature [29] has shown that CAN injection through the OBD-II port is possible in numerous cars.

3 THREAT MODEL

As briefly mentioned in Sec. 1, the common and final part of every automotive attack — which is the main threat to protect against — is to gain access to the CAN bus for a *CAN injection* attack which can lead to various forms of vehicle misbehavior, including (safety-critical) sudden acceleration. In general, there are two ways an attacker can achieve CAN bus access: **(a)** by connecting a *physical* CAN device/ECU to the IVN, e.g., an OBD-II dongle or by tapping into the CAN bus, or **(b)** compromising an existing ECU *remotely*. The former is relatively easy to accomplish as long as the attacker has physical access to the target vehicle, while the latter is more complicated and multi-layered (and thus less likely) as the attacker has to usually leverage vulnerabilities in wireless interfaces of an ECU to gain access to the device. We refer to the attacker in case of **(a)** as an *external* attacker, whereas an *internal* attacker is capable of **(b)**. Furthermore, the aforementioned separation of domains by a central gateway complicates a compromised ECU — which is usually on a less safety-critical bus (e.g., infotainment) — to affect more safety-critical domains such as powertrain which has no remote attack surfaces. Finally, even if a proper S-CAN approach is implemented, an *internal* compromise of an ECU (as in case **(b)**) will lead to exposure of secret keys which the attacker can use to forge the desired message's Message Authentication Code (MAC) and/or encrypt the CAN payload.

Although remote attacks on vehicles have skyrocketed over the last decade [8], a breakdown of attack vectors shows that most of these *remote* attacks are targeting key fobs, OEM servers and mobile companion apps. Remote attacks to compromise an ECU usually exploit the In-Vehicle Infotainment (IVI) and require significant effort (usually multiple months) as shown in the Jeep Cherokee hack [32] to achieve CAN bus access and cannot be thwarted even by a properly secured CAN bus (S-CAN). In contrast, OBD-II attacks are the fourth most common attack vector and account up to over 10% of all attacks. Nevertheless, recent research [43] has shown that remote attacks can also be launched by an *external* adversary by exploiting vulnerabilities in wireless OBD-II dongles. Many commercial OBD-II dongles feature Wi-Fi or Bluetooth capabilities which open a new over-the-air attack surface. The researchers'

findings show that CAN injection can also be performed by remote, external attackers. As a result, *external* attackers in scenario **(a)** form the most crucial threat. In what follows, we will focus on protection from this type of adversaries and describe their attack capabilities.

Once CAN bus access has been achieved, the attacker will continue a *CAN injection* attack. The authors of [12] introduce three possible CAN injection attacks as discussed next. *Fabrication attacks* allow the adversary to fabricate and inject messages with a forged CAN header and payload at a higher frequency to override cyclic CAN messages sent by legitimate ECUs that can render safety-critical receiver ECUs inoperable [27]. *Suspension attacks* on the compromised ECU prevent its broadcast of legitimate, potentially safety-critical CAN messages to the intended recipient(s). Finally, *Masquerade attacks* combine both of the above attacks by *suspending* the CAN broadcast of one ECU and deploying another ECU to *fabricate* malicious CAN messages. Only *fabrication attacks* can be mounted by our adversary from scenario **(a)**, since the others require an *internally* compromised ECU. We would like to emphasize that fabrication attacks can not only be mounted by attackers having physical access to the car, but also by remote attackers [43] which makes *external* attacks from scenario **(a)** an highy likely and scalable threat.

As a result, we assume the (external) adversary to only be able to perform *fabrication attacks* in our threat model. Even then, the attacker can cause havoc for both vehicle and driver, as shown in the Toyota Prius hack [29]. To prevent fabrication attacks, a solution for secure CAN must have the following two security properties:

Authenticity. As outlined before, any CAN node can join the IVN. There is no provision of verifying the authenticity of an added malicious device to the CAN bus by default. So, device authentication is important, i.e., only pre-authorized ECUs will be allowed to communicate. Furthermore, an attacker should not be able to spoof legitimate CAN messages during a fabrication attack. This can be prevented by adding a MAC to each message to ensure *integrity*. The latter also includes protection against replay attacks by adding a counter to each message. The major drawback of protecting against fabrication or replay attacks is the required additional space for MACs and freshness values. This is challenging because CAN only has an 8-byte payload field, with most of the space already occupied by control data (see Sec. 5.2).

Confidentiality. CAN message data is not encrypted, and therefore, messages between ECUs can be eavesdropped and analyzed by anyone accessing the IVN. To prevent this type of attack, mechanisms to guarantee *confidentiality* are required. As mentioned before, plaintext data can be recorded and used for reverse-engineering the proprietary CAN message format (i.e., signal location, scale and offset) which can be ultimately used to craft well-formed CAN messages in a fabrication attack to cause visible damage. Encryption with symmetric session keys between participating ECUs is a solution, although it will incur additional latency overhead.

In this paper, we want to protect against fabrication attacks by leveraging a combination of confidentiality and authenticity protection. Since we focus on the tension between security and performance as previously discussed, `S2-CAN` uses a non-traditional approach instead of cryptographic encryption and authentication in order to optimize performance.

4 RELATED WORK

4.1 Authenticity and Integrity

Most existing work on Secure CAN (see Table 1) focuses on the authentication of sender ECUs, protecting the integrity of the payload, as well as against replay attacks.

vatiCAN [33] offers backward-compatible sender and message authentication, as well as protection against replay attacks for safety-critical CAN messages via HMACs computed from preinstalled keys. The HMAC is sent in a separate message with a different CAN ID. vatiCAN adds 3.3ms latency per CAN message, a 16.2% increase in bus utilization and 400 bytes of memory overhead.

IA-CAN [21] provides sender authentication via randomization of CAN IDs on a per frame basis and payload data authentication using two different session keys. The receiver only accepts a message if the MAC is correct and the CAN frame has the expected CAN ID that changes with each frame using a function. The receiver's filter is updated accordingly when the next frame is accepted.

CaCAN [28] uses a hardware-modified central monitoring node to perform the entire authentication on the CAN bus. As with the general case of centralized authorities, if the monitor node is compromised or removed, the entire network is compromised. Furthermore, no encryption is used and the bus load is doubled.

TESLA [34] protocol is a lightweight authentication protocol, relying on delayed key disclosure to guarantee message authenticity. It provides authenticated broadcast capabilities, albeit with additional latency during authentication.

CANAuth [41] uses out-of-band transmission of integrity and freshness values to avoid bus load overhead. Its major drawback is the lack of backward compatibility with regular CAN controllers.

LeiA [37] is a counter-based authentication protocol that uses extended (29-bit) CAN IDs to include freshness values and a generic MAC algorithm for authentication. The MAC is 8 bytes long and transmitted in a separate CAN message, doubling the bus load. No latency numbers are reported.

4.2 Confidentiality

The space-limited payload field of 8 bytes in CAN messages is a major problem for encryption algorithms such as AES-128 that depend on a 16-byte block size. As a result, multiple messages have to be sent, increasing the bus load. Latency is another issue due to the limited computational power on ECUs if implemented in software to guarantee backward compatibility. [9] surveyed different encryption methods for the CAN bus in terms of bus load, latency and security. Existing approaches use AES-128 [17], AES-256 [38], XOR [20, 23], Tiny Encryption Algorithm (TEA) [25] and Triple DES (3DES) [22].

4.3 Key Management

Secret keys are necessary to generate and verify MACs, and to encrypt and decrypt data. Instead of using a single long-term key for the entire lifespan of a car — which is 12 years on average [10] — session keys can be generated periodically that are only valid for a certain period to limit their exposure.

In Secure CAN (S-CAN) solutions, there are two general approaches to in-vehicle key management. The first approach is to

deploy an OEM backend and request new keys periodically via Over-the-Air (OTA) using the *authenticated key exchange protocol 2* (AKEP2) [44]. Keys can be stored in the central gateway (acting as the in-vehicle key master) in a *Trusted Platform Module* (TPM) or *Hardware Security Module* (HSM). The second approach tries to do the key management completely on-board without the need for an OEM-provided backend which can reduce complexity, bandwidth and cost [39]. The key distribution *inside* the vehicle can be done in two ways. First, the key master generates and distributes new session keys based on the *Secure Hardware Extensions* (SHE) Key Update Protocol. Second, the key master triggers the ECUs to derive session keys from a nonce and long-term keys installed at manufacturing time. The first approach is superior if security is the most important and waiting on startup time is acceptable. The second approach can be used when speed is the most important and no wait time for key distribution is acceptable.

5 SYSTEM DESIGN

We now present the system design of S2-CAN, which consists of three phases: **Key Management**, **Handshake**, and **Operation**. Although no cryptography will be used in the operation phase (Sec. 5.3), establishing a session S_i during the handshake (Sec. 5.2) needs the distribution of keys which will be briefly discussed in Sec. 5.1. In our prototype, we use $N = 2$ slave ECUs and one master ECU which is the central gateway. The master ECU will be responsible for establishing new sessions during the handshake phase. There is no real value of expanding the testbed to more than 2 slave ECUs since the benchmark in Sec. 7 shows that S2-CAN does not add any communication overhead and is thus independent of traffic/bus load during the operation phase, i.e., when operation-related CAN messages are exchanged between ECUs. S2-CAN is applied to each CAN sub-bus independently. As a result, the OEM can choose which CAN buses to protect. We will use the syntax $m = (CAN_ID, Payload)$ for a CAN message m exchanged on the bus. Furthermore, we require a logical ordering of the slave ECUs for error handling and timeout purposes during the handshake (Sec. 5.2), i.e., that ECU_A transmits before ECU_B. The ordering can be assigned randomly (as in our case) or according to criticality/relevance of the ECU, with the more safety-critical slave ECU being assigned as ECU_A. This knowledge of ordering can be stored as an additional one-byte unsigned integer in each ECU's non-volatile memory.

5.1 Phase 0: Key Management

S2-CAN refrains from using Message Authentication Codes (MACs) and encryption based on cryptographic keys during the vehicle's operation mode (Sec. 5.3). During the handshake phase (Sec. 5.2), we will distribute S2-CAN-specific *session parameters* from the master ECU (gateway ECU_{GW}) to the two slaves ECU_A and ECU_B on a safety-critical CAN domain named CAN_1. These session parameters establish a new S2-CAN session S_i that is valid for a *Session Cycle* T. To distribute these parameters securely in each session, we CANNOT avoid cryptography in the handshake phase and need to ensure that the CAN payload is both authenticated and encrypted to defend against spoofing and eavesdropping attacks on the handshake. This requires the existence of pre-shared secret keys that are provided by the key management system in a vehicle. Since a detailed discussion of key management is not in the scope of this paper, we use pre-installed symmetric keys on each ECU and refer to the aforementioned best practices of in-vehicle key management (see Sec. 4.3). Note that it is transparent to the design of S2-CAN of *how* these symmetric keys are obtained, i.e., if a backend periodically provides them via OTA or they are derived from a long-term key installed at manufacturing time. Nevertheless, the use of short-lived session keys is recommended to limit exposure of the long-term key which would allow eavesdropping attacks on the handshake and thus fully compromise S2-CAN.

5.2 Phase 1: Handshake

Overview: Upon initialization, ECU_{GW}, ECU_A and ECU_B on CAN_1 will perform a 3-way handshake in order to exchange the information about the aforementioned session parameters and agree on "talking" in S2-CAN syntax. The session parameters consist of a global **(a)** *encoding parameter* f, **(b)** a slave ECU-specific *integrity parameter* int_ID_j, **(c)** a slave ECU-specific *integrity parameter* $pos_{int,j}$, and **(d)** a slave ECU-specific *counter value* cnt_j, with j denoting the respective slave ECU. Parameter **(a)** will be distributed in Stage 1, whereas the other three parameters **(b)-(d)** will be exchanged between ECUs in Stage 2. The handshake comprises three stages and repeats for each new session S_i in periodic fixed-intervals T which represents the *session cycle*. In what follows, we will describe the handshake process for an arbitrary session S_i. The communication diagram for Phase 1 is depicted in Fig. 2 and separated into the three stages. The CAN IDs used for messages during the handshake are merely examples, but should have a low ID or high priority.

Stage 1 (Initialization): The master ECU (ECU_{GW}) indicates that it wants to start a new session S_i. It randomly generates an 8-byte *encoding parameter* $f_0 = (r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7), r_l \in [0, 7]$. r_l corresponds to the bit rotation number for the lth byte in the 8-byte CAN payload. Each r_l can be expressed with 3 bits for a total of 3 bytes to include in the payload p of the gateway initialization message $m_{GW,init}$ = (0x010, p). As discussed before, due to the sensitivity of handshake messages, each CAN message during the handshake has to be both authenticated and encrypted to prevent spoofing and eavesdropping, but also replay attacks. To achieve the latter, we first add a 2-byte counter cnt_0 (not to be confused with the ECU-specific session parameter cnt_X) to defend against replay attacks. In order to prevent spoofing attacks on this message, we calculate the SHA256-HMAC of the previous 5 bytes (i.e., f_i and cnt_i) to obtain a 32-byte output with the symmetric key k from Phase 0. Since the payload of $m_{GW,init,i}$ only has another 3 bytes of free space to fit the MAC which would be too small to defend against brute-force attacks, we have to truncate the HMAC (taking the MSBs per definition). The truncation can be done safely since the increased advantage of the attacker would be offset by the limited availability of a CAN message due to the cyclic message nature of CAN and the invalidation through the counter value cnt_i. Nevertheless, we believe that 3 bytes for a truncated HMAC is too small. As a result, we split $m_{GW,init,i}$ into two consecutive CAN messages $m_{GW,init,i,0}$ and $m_{GW,init,i,1}$ with respective payloads p_1 and p_2 to **(a)** utilize another 8 bytes for the truncated HMAC, resulting to a total of 11 bytes, and **(b)** allow encryption with a

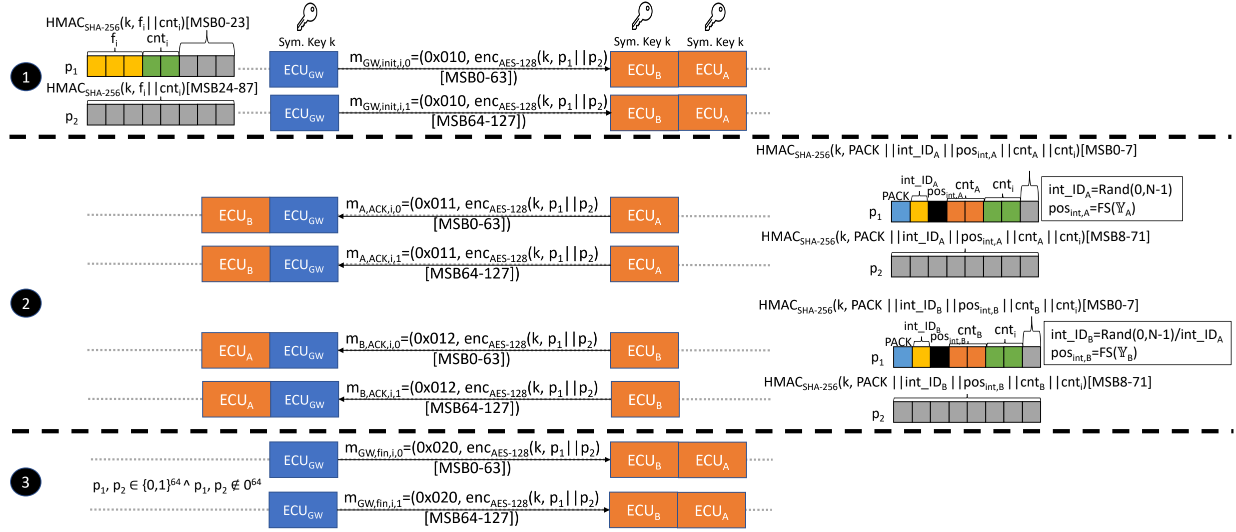

Figure 2: Handshake communication diagram

secure block cipher such as AES-128 which has a block size of 16 bytes.

In summary, two CAN messages with the following syntax are broadcast sequentially on CAN_1:

$$m_{GW,init,i,0} = (0x010, enc_{AES128}(k, p1||p2)[MSB0-63])$$
$$m_{GW,init,i,1} = (0x010, enc_{AES128}(k, p1||p2)[MSB64-127])$$

Stage 2 (Acknowledgment): Upon receiving both initialization messages from ECU_{GW}, ECU_A and ECU_B first decrypt the ciphertexts $p_1^\star$ and $p_2^\star$ using the symmetric key k and extract the *encoding parameter* f_i into local memory. Each slave ECU will then broadcast an acknowledgment (ACK) message $m_{j,ACK,i}$ (which will be split into two messages again due to AES-128 encryption), where $j \in [0,\ldots,N-1]$, consisting of a 1-byte positive acknowledgment code (PACK) and the three slave ECU-specific parameters **(b)-(d)** in the CAN payload. Parameter **(b)** is a randomly generated unique internal ID $int_ID_j \in [0, N-1]$ representing ECU_j on CAN_1 during the current session S_i. This parameter can be encoded with 1 byte since a CAN domain (or even vehicle in general) never has more than 256 ECUs.

Next, parameter **(c)** specifies the random position $pos_{int,j}$ of where the internal ID (parameter **(a)**) will be located within the CAN payload. Since space within the payload is limited and specific positions are occupied by CAN signal data that cannot be overwritten, the internal ID has to be included in available free space. The set of available free spaces for a CAN ID in a given vehicle is defined as $\mathcal{Y}_j$. Sec. 6 discusses the distribution of free spaces among CAN IDs by analyzing the DBCs of 4 different vehicles. For instance, $\mathcal{Y}_j = 12, 13, 14, 25, 26, 54, 55, 63$ states that the CAN ID belonging to ECU_j possesses only 8 bits of free space over 4 non-consecutive "regions". This set of bits is then used by the *Free Space* (FS) function to randomly determine the first bit $pos_{int,j}$ where int_ID_j will be placed:

$$pos_{int,j} = FS(\mathcal{Y}_j) \quad (2)$$

In our example, if $pos_{int,j} = 54$, the MSB of the one-byte internal ID will be stored at bit position 54 and the LSB at bit position 26.

The last parameter **(d)** is the initial value of an ECU-specific counter cnt_j for replay protection and is also randomly generated. This parameter consists of 2 bytes and is also included in available free space together with int_ID_j by Eq. 2.

Besides including these functional handshake parameters, the ACK messages will also include a 2-byte handshake counter cnt_i and truncated HMAC for integrity and freshness protection, just like in Stage 1. We obtain 2 consecutive CAN messages broadcast by ECU_j that are both authenticated and encrypted with the following syntax:

$$m_{A,ACK,i,0} = (ID_j, enc_{AES128}(k, p1||p2)[MSB0-63])$$
$$m_{A,ACK,i,1} = (ID_j, enc_{AES128}(k, p1||p2)[MSB64-127])$$

Due to the aforementioned pre-determined order for all slave ECUs, ECU_A will first transmit with CAN ID 0x011 and ECU_B needs to wait until it has received both $m_{A,ACK,i,0}$ and $m_{A,ACK,i,1}$ from ECU_A before it can broadcast $m_{B,ACK,i,0}$ and $m_{B,ACK,i,1}$. For the latter two messages, the CAN ID can simply be incremented by one as depicted in Fig. 2, as each ECU will use a distinct CAN ID. Once ECU_B receives the aforementioned ACK message, it first extracts the received *integrity parameters* into its memory and then repeats the ACK process for itself. To avoid collisions in internal ID assignment, it needs to exclude int_ID_A during the random ID generation.

Stage 3 (Finalization): ECU_{GW} finalizes the handshake after receiving ACKs from all slave ECUs. It sends $m_{GW,fin,i}$ with a random non-zero payload to signal that it has received well-formed ACK messages from all slave ECUs and monitored a successful handshake. The finalization message is again split into two CAN messages and broadcast with CAN ID 0x020.

Security and Reliability Analysis: Due to authentication, an adversary cannot spoof the contents of a handshake message. An attacker cannot replay handshake messages due to the freshness counter, and eavesdropping attacks can be mitigated by encryption.

If any ACK message takes too long due to bus or ECU errors, the handshake times out and ECU_{GW} restarts the handshake with

Stage 1. If the handshake is still unsuccessful even after repeating it r times, all ECUs on CAN_1 can revert to regular CAN communication until the next start of the vehicle. Although this countermeasure has been designed for non-adversarial reliability issues, an adversary still cannot exploit it. An attacker could launch a Denial-of-Service (DoS) attack through the OBD-II device by injecting high-priority CAN IDs (e.g., 0x0) with the goal to circumvent successful handshakes and downgrade to regular CAN communication. Since vehicles have a holistic security concept in place (as discussed in Sec. 1), the gateway (which is directly connected to the OBD-II port) can defend against this availability attack by discarding injected CAN messages under a certain CAN ID threshold, i.e., the lowest handshake CAN ID.

5.3 Phase 2: Operation

After the handshake for a session S_i has been completed, slave ECUs can start the *Operation Mode* exchanging regular data on CAN_1. To save space in the CAN payload field, we perform the following operation on the 1-byte int_ID_j and 2-byte cnt_j that ECU_j stored during the handshake to calculate the 2-byte parameter q_j:

$$q_j = \text{LEFTZEROPAD}(int_ID_j, 8) \oplus cnt_j. \quad (3)$$

First, the payload of a CAN message is being logically ORed with q_j which includes the integrity parameters into the free space of a CAN message. Second, a *Circular Shift* (CS) operation is performed on the new payload using the stored encoding parameter f_i which does a byte-wise bit rotation to the lth byte according to the value of the lth element of f_i. Finally, the message is broadcast on CAN_1. For the next CAN message sent by ECU_j, its local counter will be incremented.

On the receiver side, the respective slave ECU(s) need(s) to execute the above process reversely, i.e., rotate each byte of the encrypted payload in the opposite direction according to r_l, extract the position information from $pos_{int,j}$, determine the internal ID and finally the counter/freshness value by XORing it with int_ID_j of the sender.

Based on these extracted values, the receiver can then perform an integrity and freshness check: (1) The extracted counter cnt_j is compared with the expected counter for the respective sender. If the two values match, the local counter for sender ECU_j on the receiver is incremented, and (2) the internal ID of the sender int_ID_j is compared with the stored internal ID for the respective sender on the receiver ECU. Only if these two checks do not fail, the receiver can assume that the message came from a legitimate sender ECU_j and start processing the data in the payload. Otherwise, it may either suspect a replay attack or a message with fabricated information from a malicious ECU and drop the CAN message.

The operation mode with the respective encoding and integrity parameters ends once a new handshake has been completed. A new session S_{i+1} begins. The operation mode does not get interrupted by the start of a new handshake to guarantee functionality and safety.

Finally, we discuss what happens in the case of packet drops that can happen naturally on the CAN bus. Since each CAN message has a counter to prevent replay attacks and the receiver expects the next message with an incremented counter value, a packet drop can lead to inconsistencies with the local state counter on the receiver side. In order to account for packet drops, the receiver ECU will still accept CAN messages with counter values higher than the previous message within a specific threshold. The latter depends on the packet loss rate on the CAN bus which is usually very robust. The authors of [46] suggested to setting this threshold to 1.

6 FINDING FREE SPACE

To gain a better understanding of how many signals are used in a CAN ID and thus how much of free space (FS) is available to include our integrity parameters int_ID_j and cnt_j, we analyzed the DBC files of four passenger vehicles from a North American OEM under NDA (see Sec. 8.1). Since we include a 2-byte parameter q_j into the CAN payload, only a maximum of 6 bytes may be used for data. Among all CAN IDs in each DBC, we identified certain low-priority *non-operation-related CAN IDs* that do not occur during regular operation of the vehicle. Hence, we manually removed these irrelevant CAN IDs for our purposes and analyzed the remaining *operation-related CAN IDs* for available unused space.

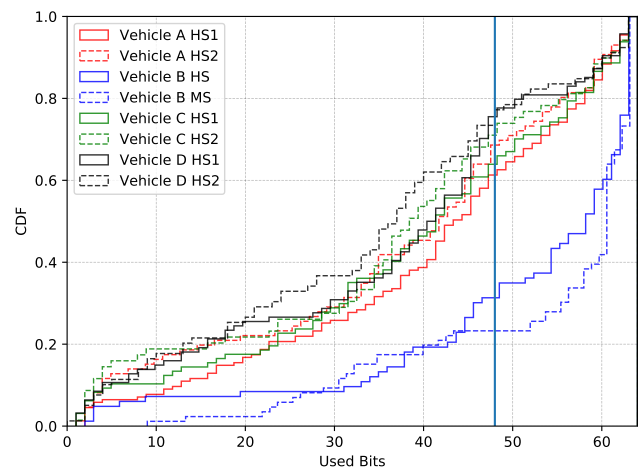

Figure 3: CDF of used bits

A Cumulative Distribution Function (CDF) for each vehicle is plotted in Fig. 3. The vertical marker indicates that all vehicles — with the exception of Vehicle B — contain between 60% and 80% CAN IDs that have at least 16 bits of free space. As a result, we can apply S2-CAN for the majority of CAN IDs, but would like to analyze how to further improve this metric to maximize the number of usable CAN IDs. Note that we are referring to the free space in the CAN payload/data field and not the CAN ID field (see Fig. 1).

OEMs could re-balance the disparity of available space in a CAN message with a more careful design of the CAN communication matrix while still considering functional requirements. In what follows, we present a possible re-balancing approach. CAN messages are differentiated by four types: fixed-periodic, event-periodic, event-on-change and network management. First, we grouped CAN IDs based on the sender ECU. As mentioned before, a sender can transmit multiple CAN IDs with different cycle times if the CAN ID is fixed-periodic or event-periodic. The latter message type is similar to fixed-periodic except a CAN message is not necessarily transmitted at every cycle time. Both message types cannot be grouped together.

Table 2: Free space in DBCs

Veh.	Bus	#IDs	#Rebalancable IDs	#IDs with FS	Usable CAN IDs (%)
Veh.A	HS1	102	31	63	92.2
	HS2	53	2	35	69.8
Veh.B	HS	81	5	26	38.3
	MS	62	3	16	30.6
Veh.C	HS1	57	7	38	78.9
	HS2	42	1	26	64.3
Veh.D	HS1	58	7	43	86.2
	HS2	51	4	38	82.4

As an example, Fig. 4 depicts the number of used bits of fixed-periodic CAN messages with exactly the same cycle time that a sender ECU transmits on HS1-CAN (high-speed CAN 1) of Vehicle A. Points above the red threshold line of 48 bits depict CAN IDs that do not have sufficient free space for S2-CAN. Since all vertical dots are grouped by sender ECU and cycle time, they can be re-balanced by packing signals of their mean value per CAN ID (depicted with marker x). For Veh. A HS1, there are a total of 101 fixed-periodic CAN IDs. A mean value below 48 bits indicates that the CAN IDs in the group can be re-balanced. 27 CAN IDs can be re-balanced this way, besides those already under this threshold. We repeated this experiment for all other vehicles and buses for both fixed-periodic and event-periodic messages and summarized the number of re-balancable and existing CAN IDs with free space in Table 2. The sum of these two yields the number of usable CAN IDs for S2-CAN. With the exception of Veh. B, around 79–92% of all CAN IDs can be used with S2-CAN for the more safety-critical HS1-CAN. The remaining non-periodic CAN IDs can be re-balanced further by OEMs based on functionality — something that we cannot interpret.

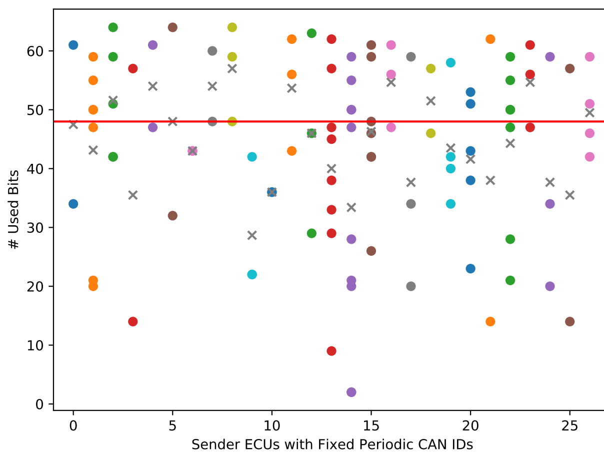

Figure 4: Re-balancing Vehicle A HS1

Finally, no relationship between message priority and free space can be derived. This analysis is depicted in Fig. 6 (in Appendix B).

7 EVALUATION

7.1 Experimental Setup

We have built a prototype with three CAN nodes, each of which consists of an Arduino Mega 2560 board and a SeeedStudio CAN shield [47]. This prototype was set up to operate at a 500 kBit/s baud rate as in a typical high-speed safety-critical CAN bus. Note that the entire evaluation is based on a simple scenario with the sender ECU transmitting only one CAN message. In reality, multiple CAN messages will be broadcast on the CAN bus in a relatively short time and CAN scheduling will pick the highest-priority CAN message to be broadcast first. This will inherently lead to blocking time t_b for lower-priority messages which depends on the number of higher-priority messages that have to be transmitted first. Nevertheless, using a simpler setup does not affect our evaluation metrics except the operation latency which is discussed in Sec. 7.3.

Since we want to compare the performance of S2-CAN with prior work, we implemented existing CAN bus encryption methods from Sec. 4.2 with vatiCAN [33] for authentication. We chose vatiCAN among various existing SW-only CAN authentication approaches due to its decent performance for both latency and bus load, as well as existing and well-documented Arduino implementation.

7.2 Handshake Latency

We measured the time it took to complete a handshake while varying the number of slave ECUs in a CAN domain. As outlined in Sec. 5.2, the handshake process is repeated every T. The old session still continues with the existing parameters until the handshake is completed. As a result, no critical message exchange during the operation mode of the previous session is interrupted. The handshake of the new session will be executed in parallel with the operation of the previous session. The only critical time when the handshake latency can affect operations of the car is during the initial start-up of the car since a session S_0 of S2-CAN cannot start until the initial handshake has been completed. We simulated a varying number of slave ECUs by having our two prototype ECUs take turns to send ACK of the handshake, in a ping-pong manner. We surveyed the DBCs of four vehicles (see Sec. 8.1) to find that each CAN bus has 9–23 different ECUs. So, we consider a maximum of 25 slave ECUs in our simulation. For two slave ECUs, the average total handshake time stands at 303ms, for five at 529ms, for ten at 907ms and for the maximum number of 25 slave ECUs, we achieve around 2 seconds of handshake latency t_{hs}, i.e., the car starts talking S2-CAN after 2s when it is powered on. Our calculations also show that each additional slave ECU on the bus will add an average of 75.5ms towards the latency. Furthermore, the handshake process will be started at $P \cdot T - t_{hs} - Q \cdot t_b$ *before* the current session expires to provide a smooth transition to the next session. P denotes the session number and Q the average number of higher-priority CAN messages that can be expected to cause the blocking of handshake messages.

7.3 Operation Latency

CAN messages have stringent deadlines, i.e., when they must arrive at the receiver. Although the authors of [16] suggest deadlines of cyclic safety-critical CAN messages standing at 2.5–10ms, this is outdated. Modern HS-CAN buses have minimum cycle times (and

thus deadlines) of 10ms, as our manual inspection of the four DBCs also confirmed. Latency measurements are averaged from a sample of 1000 messages sent over 100 seconds, or one message every 100ms. We were interested in calculating the E2E latency t_{E2E} for

(1) Regular CAN with vatiCAN authentication ("NONE"),
(2) 3DES, TEA, XOR, AES-128 and AES-256 encrypted CAN with vatiCAN authentication,
(3) and finally S2-CAN.

In the first case, E2E latency consists of processing delays of the sender and receiver, the time to calculate the MAC on the sender and check the MAC on the receiver, as well as the CAN bus network latency. In the second case, encryption/decryption latencies are added on the respective sides. S2-CAN uses the latter calculation methodology as well, while the MAC and encryption/decryption latencies are replaced by the delay to calculate/check the internal ID and counter, and encode/decode through Circular Shift (CS).

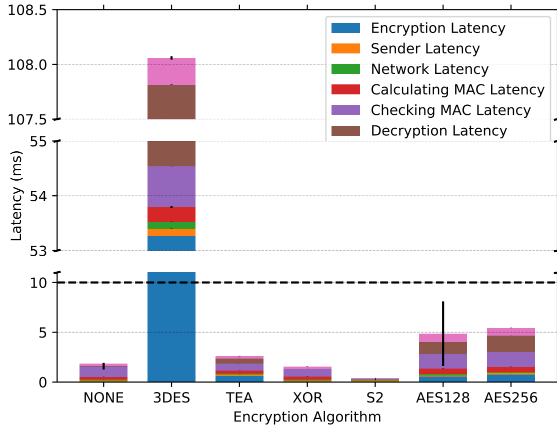

Figure 5: E2E latency for different "encryption" algorithms

Fig. 5 depicts the breakdown of the E2E latency for all three aforementioned cases. Furthermore, the dotted horizontal line indicates the aforementioned deadline of 10ms. It can be easily seen that the encryption/decryption of 3DES takes much longer on Arduinos than other encryption algorithms that can still satisfy the 10ms deadline. Tiny Encryption Algorithm (TEA) and XOR seem to satisfy it although they are not considered secure [7, 26] and are thus not recommended to be used in production. Furthermore, in all experiments, we did not include any additional traffic, so that the reported E2E latencies assume no blocking time due to higher-priority CAN messages and can be considered a lower bound. Hence, even AES-128 and AES-256 are likely to miss the 10ms deadline if they lose the CAN arbitration to a message with lower ID. S2-CAN with $t_{E2E} = 414\mu s$ satisfies both deadlines and only adds an overhead of $75\mu s$ to the E2E latency of a regular CAN message (i.e., no encryption or authentication).

Latency numbers for MAC operations by vatiCAN are lower in Fig. 5 than the reported 3.3ms from Table 1. We used a sponge capacity of $c = 8$ instead of the original, more secure $c = 128$ to provide a lower bound for vatiCAN's latency overhead.

Table 3: Benchmark of other metrics

Encr.	Auth.	BL (%)	CPUo (%) S / R	RAM(kB) S / R	Flash(kB) S / R
None	None	0.25	0/0	1.24/1.29	10.1/11.96
	VatiCAN	0.5	86.7/82.3	1.57/1.66	17.25/17.07
AES128	None	0.5	0.8/2.0	1.25/1.30	10.30/12.02
	VatiCAN	1	87.0/82.8	1.60/1.67	17.35/17.13
AES256	None	0.5	1.0/2.5	1.27/1.31	10.31/12.04
	VatiCAN	1	87.0/82.9	1.61/1.69	17.37/17.15
3DES	None	0.25	52.8/53.5	1.26/1.31	12.27/14.22
	VatiCAN	0.5	93.8/90.8	1.60/1.69	19.38/19.33
TEA	None	0.25	0.5/0.5	1.27/1.32	10.55/12.50
	VatiCAN	0.5	86.8/82.4	1.60/1.69	17.78/17.61
XOR	None	0.25	0.01/0.01	1.25/1.30	10.16/12.05
	VatiCAN	0.5	86.7/82.3	1.57/1.67	17.31/17.17
S2	S2 Auth	0.25	0.04/0.03	1.25/1.30	10.24/12.10

7.4 Other Metrics

Besides the E2E latency, we measured bus load, CPU overhead, and memory usage of each encryption method with and without vatiCAN authentication. The results are summarized in Table 3. The metrics are calculated for messages exchanged during *Operation Mode*, unless noted otherwise.

Bus Load. The bus load (BL) b is calculated as follows [2]:

$$b = \frac{s_{frame}}{f_{baud}} \sum_{m \in M} \frac{1}{p_m}, \tag{4}$$

where we used $f_{baud} = 500\ kBit/s$ as baud rate on the CAN bus, and p_m is the period/cycle time of message m, and assuming each CAN frame uses 125 bits, $s_{frame} = 125$. With regular CAN (no encryption and authentication), we send one message every p_m. AES has a block size of 16 bytes and the maximum size of the payload is 8 bytes. Thus, we send two consecutive messages, each with a period of p_m. With vatiCAN authentication, an additional MAC is sent after each message, effectively doubling the bus load. Table 3 shows that only S2-CAN does not add any overhead to the bus load of regular CAN during operation mode, but provides protections against both confidentiality and integrity. Note that the bus load does increase during each handshake due to additional $2(N + 2)$ exchanged messages. Nevertheless, the handshake adds an overhead of merely 2.5% to the bus load.

CPU Usage. CPU overhead (CPUo) c_y of ECU_y is calculated by measuring how many idle cycles pass per message. We establish regular CAN to be the baseline, then calculate overhead c_y for $y \in \{\mathbf{S}ender, \mathbf{R}eceiver\}$ as follows:

$$c_y = 1 - \frac{cycles_{idle}}{cycles_{baseline}}. \tag{5}$$

We see in Table 3 that vatiCAN authentication accounts for the largest CPU overhead. (with the exception of 3DES). The CPU utilization on each ECU almost doubles. With S2-CAN, we have a negligible CPU overhead that demonstrates the lightness of our approach on computational resources.

Memory Consumption. Finally, Flash and RAM usage are reported when our code compiles to the Arduinos. No dynamic memory is used. All approaches except S2-CAN add up to 30% more RAM and 70–90% of Flash usage compared to the memory consumption for regular CAN. The memory consumption (both RAM and Flash) for S2-CAN is minimal.

8 SECURITY ANALYSIS

To measure the security level of S2-CAN, we need to determine the time an attacker requires to correctly spoof a specific CAN message. To be more concrete, we assume the adversary will try to accelerate the vehicle by CAN injection through the OBD-II port. Furthermore, we assume that the gateway blocks CAN messages with IDs under a certain threshold to secure the handshake (see Sec. 5.2) and no intrusion detection system is installed in the target vehicle. Given the current state of commercial passenger vehicle security, this is a very likely scenario. In order to affect the acceleration behavior by CAN message injection, the adversary needs to know the message format (i.e., CAN ID, signal position, scale and offset) of the signal they want to spoof. For regular CAN, this is possible by existing automated CAN bus reverse-engineering tools such as LibreCAN [36]. In the following security analysis, we will deploy Phases 0 and 1 of LibreCAN with some modifications to adapt to S2-CAN and try to measure the time an attacker would need to determine the correct payload to inject into the CAN bus. The modified attack tool is called LibreCAN+, consisting of three stages that are discussed below.

8.1 Experimental Setup

All experiments were conducted using Python 3 on a computer running 64-bit Ubuntu 18.04.4 LTS with 128 GB of registered ECC DDR4 RAM and two Intel Xeon E5-2683 V4 CPUs (2.1 GHz with 16 cores/32 threads each). We evaluate the security of S2-CAN by using one-hour real-world traces collected from four recent (2016-2019) vehicles: Veh. A is a luxury mid-size sedan, Veh. B a compact crossover SUV, Veh. C a full-size crossover SUV and Veh. D a full-size pickup truck. Veh. A, C and D have at least two HS-CAN buses, both of which are routed out to the OBD-II connector, whereas Veh. B has at least one HS-CAN and one MS-CAN, with only the former being accessible via OBD-II. All raw CAN data was collected with the OpenXC VI [6].

8.2 Stage 0: Generating S2-CAN Traces

The recorded traces from our four evaluation vehicles are in regular CAN-syntax. To enable S2-CAN-compliant communication, we have to process the one-hour traces according to simulated handshake parameters and convert them into S2-CAN-syntax. First, we analyze the DBC file of the vehicle to determine the ECU nodes that are present in the network, free space of each CAN ID payload, and group CAN IDs based on the node that emits them since the handshake assigns the parameters on a per-node basis. Then, we randomly assign each node a unique internal ID $\in [0, N_{ECU} - 1]$. The counter of each node is also initialized to a random number in range $[0, 2^{16} - 1]$. Third, we assign incrementing counter values for each CAN message. After specifying values for the internal ID and counter of each CAN message, we XOR the two values to obtain q_j, assign it to a free space in each CAN message (if possible) and finally OR it with the original payload. In order to be compliant with S2-CAN, the payload needs to have at least 2 bytes of free space, but these do not have to be contiguous. We removed CAN IDs from the trace that do not have the necessary free space. Finally, we perform the byte-wise circular shift (CS) on each remaining message according to the randomly generated encoding parameter f.

8.3 Stage 1: Cracking the Encoding

First, the adversary can assume that the targeted CAN signal is two bytes or less in size since this applies to most powertrain-related signals. In all four vehicles the target signal is 13 bits long. Next, the attacker can brute-force the CAN trace with each possible encoding for each of the 7 pairs of contiguous bytes in the CAN message. Our encoding scheme has 8 possibilities for each byte, so without accounting for duplicates, there are $8 \cdot 8 \cdot 7 = 448$ combinations an attacker must try. However, because encodings for unconsidered bytes are set to zero, we can reduce this to 400 combinations by eliminating duplicates: One combination of all zeros, $7 \cdot 8 = 56$ combinations where all but one byte are zero, and $7 \cdot 7 \cdot 7 = 343$ combinations where all but two contiguous bytes are zero. For each potential encoding, the attacker decodes the trace and runs it through Phases 0 and 1 of the original LibreCAN, resulting in a list of three-tuples (candidate CAN ID, encoding, normalized cross-correlation score). The pairs with the highest X correlation scores (X is a design parameter in Sec. 8.5) can then be used in Stage 2. Note that we used multi-threading in this stage to calculate up to 50 combinations simultaneously.

8.4 Stage 2: Authenticating Correctly

For the adversary to successfully spoof a message, they must be able to increment the message counter to the correct value. This requires the knowledge of the position of the counter bits within the message, the value of the counter, and the internal ID. After determining the top X CAN IDs by correlation score from Stage 1, the adversary can extract a subtrace consisting of only the messages for that candidate CAN ID. With the subtrace in hand, the adversary calculates the frequency of bit flips for each bit in the subtrace's messages, and matches these flip frequencies to what frequency the bits of a counter should be. This is done using Algorithm 1. Note that only the lowest $\lfloor \log_2(\text{trace length}) \rfloor$ bits of the counter can be determined, since these are the only bits that are guaranteed to flip at least once.

Algorithm 1 Determine Counter Position

```
procedure MATCH-FREQUENCY(flip_freqs, trace_len)
    counter_length ← min(16, ⌊log2 trace_len⌋)
    counter_positions ← []
    for i ← counter_length to 1 do
        match ← argmin({|f − 2^−(i−1)| : f ∈ flip_freqs})
        APPEND(counter_positions, match)
    return counter_positions
```

After determining the position of the counter bits, the internal ID can be extracted. To do this, the adversary compares consecutive messages in the subtrace, and sees if one of the counter bits flips in

the second message. If this occurs, the adversary knows the next lowest bit of the counter must have been a 1 in the first message. Then, to extract the internal ID, the adversary XORs the counter bit with 1. This is repeated until all bits of the internal ID are known. This procedure is summarized in Algorithm 2.

Algorithm 2 Determine Internal ID

```
procedure CALCULATE-INT-ID(counter_pos, subtrace)
    c_length ← LENGTH(counter_pos)
    id_length ← min(8, c_length − 1)
    int_id ← []
    offset ← c_length − id_length
    c_pos ← counter_pos[offset : c_length]
    prev_m ← GET(subtrace, 0)
    for i ← 0 to id_length − 1 do
        for m ∈ subtrace do
            if m[c_pos[i]] ≠ prev_m[c_pos[i]] then
                int_id[i] ← prev_m[c_pos[i + 1]] ⊕ 1
                BREAK
    return BITS-TO-INTEGER(int_id)
```

Now, after obtaining the position of the counter and the internal ID, the attacker can spoof a message. First, they use the encoding determined in Stage 1 to decode the latest message from the desired CAN ID. Next, the attacker replaces the value of the signal they are spoofing with their own fabricated value in that message. Before re-encoding the message with f, the attacker extracts the counter value from the latest real-time message on the CAN bus, increments it by 1, and inserts it into their new message. This spoofed message will then be injected through the adversary's rogue node into the CAN bus and accepted by the respective receiver ECUs.

8.5 Difficulty of Successful Cracking

The recorded traces of all evaluation vehicles were around 60 minutes long. We integrated the above procedure into LibreCAN — creating a new version of LibreCAN, named LibreCAN+ — and evaluated its success on those four traces using the ground truth DBC files of each vehicle. The outcome is shown in the last column of Table 4. The cracking success is dependent on finding the correct CAN ID and encoding in Stage 1 (abbreviated at ST1 in the table) by picking the top candidate in the sorted correlation list, as well as determining the correct internal ID (ID) and counter (cnt). For Vehicles A, B and C, cracking S2-CAN with LibreCAN+ works. Vehicle D already failed in Stage 1 to determine the correct CAN ID for spoofing the desired signal.

Furthermore, we wanted to analyze how a shorter recording would affect this metric. We re-ran all three stages with 5%, 10%, 25%, 50% and 75% of full trace length. To avoid bias towards more city or highway driving, we calculated the precision for all non-overlapping segments of this trace. As can be seen in Table 4, traces of 5% and 10% length fail in most cases. We color-coded the table to indicate the number of split traces cracked correctly. If all split traces can be cracked, we highlighted them in green color. Otherwise, if under 2/3 of split traces are unsuccessful, we highlighted these in red, with the remaining portion colored in orange.

Table 4 only considers those candidates in Stage 1 with the highest correlation score ($X = 1$) that match the correct encoding and CAN ID as successful. In many cases, we observed that the second-best candidate was ideal. As a result, we also wanted to see if considering the top $X = \{2, 3, 5, 10\}$ candidates from Stage 1 would lead to success in cracking S2-CAN . If any of the candidates in the top X were correct, we would mark ST1 for the respective vehicle and split trace as correct. Similar tables for the aforementioned values of X are presented in Appendix A. Based on these, we summarize the cracking performance for varying X in Table 5. The values are reported as average numbers over all four vehicles. Note that the color coding is different from Table 4. Green cells indicate that the adjacent X value to its right is identical and thus does not provide a performance improvement. We suggest using at least a trace of 25% length (15 minutes) and consider the Top 3 candidates for optimal brute-forcing success.

Table 4: Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	11/20	6/10	4/4	3/3	2/2	1/1
	ID	10/20	6/10	4/4	3/3	2/2	1/1
	cnt	11/20	6/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	3/4	2/3	1/2	1/1
	ID	11/20	3/10	3/4	1/3	1/2	1/1
	cnt	12/20	4/10	3/4	2/3	1/2	1/1
Veh. C	ST1	8/20	5/10	3/4	3/3	2/2	1/1
	ID	8/20	5/10	3/4	3/3	2/2	1/1
	cnt	8/20	5/10	3/4	3/3	2/2	1/1
Veh. D	ST1	6/20	3/10	0/4	0/3	0/2	0/1
	ID	6/20	3/10	0/4	0/3	0/2	0/1
	cnt	6/20	3/10	0/4	0/3	0/2	0/1

Table 5: Brute-Forcing Success for Top X Candidates

TL (%)		Top 1	Top 2	Top 3	Top 5	Top 10
5	ST1	46%	58%	58%	61%	65%
	ID	44%	54%	54%	58%	61%
	cnt	46%	58%	58%	61%	65%
10	ST1	45%	68%	68%	73%	78%
	ID	43%	58%	58%	63%	68%
	cnt	45%	68%	68%	73%	78%
25	ST1	63%	81%	88%	88%	88%
	ID	63%	81%	88%	88%	88%
	cnt	63%	81%	88%	88%	88%
50	ST1	67%	92%	92%	92%	92%
	ID	58%	83%	83%	83%	83%
	cnt	67%	92%	92%	92%	92%
75	ST1	63%	88%	88%	88%	88%
	ID	63%	88%	88%	88%	88%
	cnt	63%	88%	88%	88%	88%
100	ST1	75%	100%	100%	100%	100%
	ID	75%	100%	100%	100%	100%
	cnt	75%	100%	100%	100%	100%

8.6 Determining Session Cycle T

So far, we observed that brute-forcing S2-CAN successfully is possible. The total time t_a required by an attacker to crack S2-CAN is the sum of the passive recording time t_r, time t_{st1} to crack the encoding in Stage 1, time t_{st2} to determine the integrity parameters in Stage

2 and time t_i to inject a well-formed CAN message on the CAN bus:

$$t_a = t_r + t_{st1} + t_{st2} + t_i \approx t_r + t_{st1}. \quad (6)$$

Our timing analysis shows that the time to determine the two integrity parameters *int_ID* and *cnt* on the full trace (60 minutes) takes less than one second. The time to inject the correct CAN message can also occur instantly with minimal network delay from the workstation to the adversary's CAN node (e.g., an Arduino). Hence, t_{st2} and t_i are negligible and the main contributing factors are t_r and t_{st1}.

Table 6: Timing analysis for full traces (minutes:seconds)

	CAN (LibreCAN)	S2-CAN (LibreCAN+)
Veh. A	0:27	10:33
Veh. B	0:36	18:32
Veh. C	0:26	10:42
Veh. D	0:26	10:52

As shown in Table 6, the total time stands at around t_a = 70 min for full traces (i.e., t_r = 60 min). Since our threat model stipulates that the attacker can also physically tap into one specific CAN bus (and thus only has access to one bus), we run LibreCAN+ with messages from Bus 1 only. Unfortunately, due to architecture specifics of Vehicle B, all messages are logged on Bus 1, which makes the trace longer and thus affects cracking time. The attacker can only perform a CAN injection attack on a bus equipped with S2-CAN if the session cycle T is larger than t_a since with each new handshake, new parameters will be generated and the attacker has to re-do the entire attack. As a result, we can deem S2-CAN secure if the following condition is met:

$$t_a \approx t_r + t_{st1} > T. \quad (7)$$

In Sec. 8.5 an attacker was shown to succeed cracking S2-CAN with less passive recording time t_r. Since less messages have to be processed, t_{st1} will also be proportionally smaller. With the minimum recording time $t_{r,min}$ to have a successful outcome, we can now set the maximum session cycle T_{max}. We already determined that a trace length of t_r = 15 minutes is sufficient to succeed. The Top X consideration does not affect the timing since Stage 2's contribution is negligible. If the attacker doesn't achieve the desired outcome (i.e., vehicle malfunction), they can repeat the process with the second and third candidates immediately. For Vehicles A, C and D, t_{st1} stands at less than 3 minutes and for Vehicle B at less than 5 minutes. Hence, the maximum session cycle T_{max} will stand at 18-20 minutes.

9 DISCUSSION AND CONCLUSION

Based on the results from the previous section, we can guarantee that S2-CAN is secure if the cycle time T does not exceed 18-20 minutes. The experiments were conducted on a machine with relatively good specs (see Sec. 8.1). Nevertheless, a determined attacker can use an even more powerful setup to brute-force S2-CAN faster. The feasibility of such an attack depends on the attacker's incentive, i.e., tradeoff between monetary cost and dedication towards the outcome.

To be flexible, an attacker could rent computational resources online. Both Amazon and Google provide cloud computing resources called AWS EC2 [14] and Google Cloud [5]. The main bottleneck of brute-forcing is the time required in Stage 1. Due to multi-threading the combinations, these can be linearly scaled with multiple instances. We obtained the cost of running a comparable instance to our experimental setup on AWS. Their pricing calculator [1] suggested an on-demand hourly cost of US$1.088 for an EC2 instance with 32 vCPUs and 64 GB RAM. In our experiments from Sec. 8, the peak RAM usage stood at 16 GB, but with the configured number of cores, EC2 did not provide any smaller instance. To brute-force S2-CAN with a passive recording time t_r = 15 minutes in less than 20 seconds, 10 EC2 instances have to be rented. This sums up to a monthly cost of $7,972.40 for the attacker. Given that the attacker only spends $t_a \approx 15$ minutes per attempt (if $T > t_a$), they could conduct 2880 attempts per month at an average cost of $2.77 and still fail, if T is set smaller than the minimum recording time $t_{r,min}$. Although the actual cracking (i.e., t_{st1}) can be sped up, $t_{r,min}$ acts as a lower bound to the total attack time t_a and thus the attacker will have no chance of cracking S2-CAN.

Finally, we would like to briefly compare S2-CAN's security with S-CAN approaches. For instance, vatiCAN [33] discusses how long it would take to forge the SHA3-HMAC which depends on the length of the MAC tag. On average, it requires $2^{MAC_Length-1}$ combinations to brute-force the MAC which is depicted in the last column of Table 1. The authors mentioned that it would still take a day to brute-force all combinations on a powerful in-vehicle ECU, but due to their nonce update interval of 50ms (comparable to our session cycle T), it would be impossible for the attacker to calculate a correct HMAC. Although the same calculation cannot be directly applied to S2-CAN due to lack of MAC and changing position for each CAN message, an online attacker (i.e., on an in-vehicle ECU) would require $\binom{64}{16} \approx 2^{49}$ combinations to spoof the valid 2-byte integrity parameters which allows a fair comparison with the other numbers in Table 1. Given modern GPUs' capabilities [15] (also considering advances since this paper's publication), an attacker with similar cost assumptions from above could brute-force S2-CAN in multiple hours due to its 49-bit entropy. Such an attacker would still fail if $T_{max} \approx 15$ minutes.

In this paper, we have developed S2-CAN by making a trade-off between performance and security, and verified its performance on Arduinos mimicking real ECUs on a CAN bus. with regards to multiple metrics. It performs better for all metrics than each surveyed S-CAN approach, especially reducing E2E latency. Then, we have tried to brute-force S2-CAN by using a modified version of the existing CAN reverse-engineering tool LibreCAN. Although the total attack time can be minimized to roughly 15 minutes, by setting the session cycle properly, our approach is deemed secure. Due to both favorable performance and practically acceptable security guarantees, we envision S2-CAN to finally be a compelling and practical security solution for OEMs to be deployed in their vehicles in the near future.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by an Intel Labs grant and Ford Motor Company.

REFERENCES

[1] [n.d.]. AWS Pricing. https://calculator.aws/.
[2] [n.d.]. CAN Bus Load Calculation. https://kb.vector.com/entry/1519/.
[3] [n.d.]. CAN bus Load Calculator. http://www.canbusacademy.com/resources/can-bus-load-calculator/.
[4] [n.d.]. Electronic engine control unit for commercial vehicles. https://www.bosch-mobility-solutions.com/en/products-and-services/commercial-vehicles/powertrain-systems/natural-gas/electronic-engine-control-unit/.
[5] [n.d.]. Google Cloud. https://cloud.google.com/.
[6] [n.d.]. The OpenXC Platform. http://openxcplatform.com/.
[7] 2020. One-time pad. https://en.wikipedia.org/wiki/One-time_pad
[8] Emad Aliwa, Omer Rana, Charith Perera, and Peter Burnap. 2020. Cyberattacks and Countermeasures For In-Vehicle Networks. *arXiv preprint arXiv:2004.10781* (2020).
[9] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and Ian Jennions. 2020. Evaluation of CAN Bus Security Challenges. *Sensors* 20, 8 (2020), 2364.
[10] Ken Budd. 2018. How Long Do Cars Last? A Guide to Your Car's Longevity. https://www.aarp.org/auto/trends-lifestyle/info-2018/how-long-do-cars-last.html
[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive experimental analyses of automotive attack surfaces.. In *USENIX Security Symposium*, Vol. 4. San Francisco, 447–462.
[12] Kyong-Tak Cho and Kang G Shin. 2016. Fingerprinting electronic control units for vehicle intrusion detection. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 911–927.
[13] CSS Electronics. [n.d.]. CAN Bus Explained - A Simple Intro (2019). https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en.
[14] Donald J. Daly. 1987. Economics 2: EC2. https://aws.amazon.com/ec2/.
[15] Tomoiagă Radu Daniel and Stratulat Mircea. 2011. AES algorithm adapted on GPU using CUDA for small data and large data volume encryption. *International journal of applied mathematics and informatics* 5, 2 (2011), 71–81.
[16] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. 2007. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems* 35, 3 (2007), 239–272.
[17] Tri P Doan and Subramaniam Ganesan. 2017. *CAN Crypto FPGA Chip to Secure Data Transmitted Through CAN FD Bus Using AES-128 and SHA-1 Algorithms with A Symmetric Key*. Technical Report. SAE Technical Paper.
[18] CSS Electronics. [n.d.]. CAN DBC File - Convert Data in Real Time (Wireshark, J1939). https://www.csselectronics.com/screen/page/dbc-database-can-bus-conversion-wireshark-j1939-example/language/en.
[19] Elm Electronics, Inc. [n.d.]. OBD. https://www.elmelectronics.com/products/ics/obd/.
[20] Wael A Farag. 2017. CANTrack: Enhancing automotive CAN bus security using intuitive encryption algorithms. In *2017 7th International Conference on Modeling, Simulation, and Applied Optimization (ICMSAO)*. IEEE, 1–5.
[21] Kyusuk Han, André Weimerskirch, and Kang G Shin. 2015. A practical solution to achieve real-time performance in the automotive network by randomizing frame identifier. *Proc. Eur. Embedded Secur. Cars (ESCAR)* (2015), 13–29.
[22] Adam Hanacek and Martin Sysel. 2016. Design and Implementation of an Integrated System with Secure Encrypted Data Transmission. In *Computer Science On-line Conference*. Springer, 217–224.
[23] Assaf Harel and Amir Hezberg. 2019. *Optimizing CAN Bus Security with In-Place Cryptography*. Technical Report. SAE Technical Paper.
[24] Olaf Henniger, Ludovic Apvrille, Andreas Fuchs, Yves Roudier, Alastair Ruddle, and Benjamin Weyl. 2009. Security requirements for automotive on-board networks. In *2009 9th International Conference on Intelligent Transport Systems Telecommunications,(ITST)*. IEEE, 641–646.
[25] M Jukl and J Čupera. 2016. Using of tiny encryption algorithm in CAN-Bus communication. *Research in Agricultural Engineering* 62, 2 (2016), 50–55.
[26] John Kelsey, Bruce Schneier, and David Wagner. 1997. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. In *International Conference on Information and Communications Security*. Springer, 233–246.
[27] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 447–462.
[28] Ryo Kurachi, Yutaka Matsubara, Hiroaki Takada, Naoki Adachi, Yukihiro Miyashita, and Satoshi Horihata. 2014. CaCAN-centralized authentication system in CAN (controller area network). In *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*.
[29] Charlie Miller and Chris Valasek. 2013. Adventures in automotive networks and control units. *Def Con* 21 (2013), 260–264.
[30] Charlie Miller and Chris Valasek. 2014. A survey of remote automotive attack surfaces. *black hat USA* 2014 (2014), 94.
[31] Charlie Miller and Chris Valasek. 2015. *Car hacking: for poories*. Technical Report. Tech. rep., IOActive Report.
[32] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* 2015 (2015), 91.
[33] Stefan Nürnberger and Christian Rossow. 2016. –vatican–vetted, authenticated can bus. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 106–124.
[34] A. Perrig, R. Canetti, J. Tygar, and D. Song. 2000. Approaches for secure and efficient in-vehicle key management. In *Proceedings of the IEEE Symposium on Security and Privacy (SP 2000)*. 56Ű73.
[35] Mert D Pesé, Karsten Schmidt, and Harald Zweck. 2017. *Hardware/software co-design of an automotive embedded firewall*. Technical Report. SAE Technical Paper.
[36] Mert D Pesé, Troy Stacer, C Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G Shin. 2019. LibreCAN: Automated CAN Message Translator. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2283–2300.
[37] A.-I. Radu and F.D. Garcia. 2016. LeiA: a lightweight authentication protocol for CAN. *Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016* 878 (2016).
[38] Ali Shuja Siddiqui, Yutian Gui, Jim Plusquellic, and Fareena Saqib. 2017. Secure communication over CANBus. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 1264–1267.
[39] Takeshi Sugashima, Dennis Kengo Oka, and Camille Vuillaume. 2016. Approaches for secure and efficient in-vehicle key management. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems* 9, 2016-01-0070 (2016), 100–106.
[40] A. S. Thangarajan, M. Ammar, B. Crispo, and D. Hughes. 2019. Towards Bridging the Gap between Modern and Legacy Automotive ECUs: A Software-Based Security Framework for Legacy ECUs. In *2019 IEEE 2nd Connected and Automated Vehicles Symposium (CAVS)*. 1–5. https://doi.org/10.1109/CAVS.2019.8887788
[41] A. Van Herrewege, D. Singelee, and I. Verbauwhede. 2011. CANAuth – a simple, back-ward compatible broadcast authentication protocol for CAN bus. *ECRYPT-Workshop on Lightweight Cryptography* (2011).
[42] Qiyan Wang and Sanjay Sawhney. 2014. VeCure: A practical security framework to protect the CAN bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)*. IEEE, 13–18.
[43] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. 2020. Plug-N-pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new over-the-air attack surface in automotive IoT. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 949–965.
[44] Zhihong Wu, Jianning Zhao, Yuan Zhu, Ke Lu, and Fenglue Shi. 2019. Research on In-Vehicle Key Management System under Upcoming Vehicle Network Architecture. *Electronics* 8, 9 (2019), 1026.
[45] Michael Ziehensack. 2015. Safe and Secure Communication with Automotive Ethernet.
[46] Qingwu Zou, Wai Keung Chan, Kok Cheng Gui, Qi Chen, Klaus Scheibert, Laurent Heidt, and Eric Seow. 2017. The Study of Secure CAN Communication for Automotive Applications, In SAE Technical Paper. https://doi.org/10.4271/2017-01-1658
[47] Baozhu Zuo. [n.d.]. CAN-BUS Shield V2.0. https://wiki.seeedstudio.com/CAN-BUS_Shield_V2.0/.

A CRACKING SUCCESS

Table 7: Top 2 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	3/4	2/3	1/2	1/1
	ID	11/20	3/10	3/4	1/3	1/2	1/1
	cnt	12/20	4/10	3/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	f	12/20	7/10	3/4	3/3	2/2	1/1
	ID	11/20	5/10	3/4	3/3	2/2	1/1
	cnt	12/20	7/10	3/4	3/3	2/2	1/1

Table 8: Top 3 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	12/20	4/10	4/4	2/3	1/2	1/1
	ID	11/20	3/10	4/4	1/3	1/2	1/1
	cnt	12/20	4/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	12/20	7/10	3/4	3/3	2/2	1/1
	ID	11/20	5/10	3/4	3/3	2/2	1/1
	cnt	12/20	7/10	3/4	3/3	2/2	1/1

Table 9: Top 5 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	14/20	9/10	4/4	3/3	2/2	1/1
	ID	13/20	8/10	4/4	3/3	2/2	1/1
	cnt	14/20	9/10	4/4	3/3	2/2	1/1
Veh. B	ST1	13/20	5/10	4/4	2/3	1/2	1/1
	ID	12/20	4/10	4/4	1/3	1/2	1/1
	cnt	13/20	5/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	14/20	8/10	3/4	3/3	2/2	1/1
	ID	13/20	6/10	3/4	3/3	2/2	1/1
	cnt	14/20	8/10	3/4	3/3	2/2	1/1

Table 10: Top 10 Cracking Success based on Trace Length (in %)

Trace Length		5	10	25	50	75	100
Veh. A	ST1	15/20	10/10	4/4	3/3	2/2	1/1
	ID	14/20	9/10	4/4	3/3	2/2	1/1
	cnt	15/20	10/10	4/4	3/3	2/2	1/1
Veh. B	ST1	13/20	5/10	4/4	2/3	1/2	1/1
	ID	12/20	4/10	4/4	1/3	1/2	1/1
	cnt	13/20	5/10	4/4	2/3	1/2	1/1
Veh. C	ST1	8/20	7/10	3/4	3/3	2/2	1/1
	ID	8/20	7/10	3/4	3/3	2/2	1/1
	cnt	8/20	7/10	3/4	3/3	2/2	1/1
Veh. D	ST1	16/20	9/10	3/4	3/3	2/2	1/1
	ID	15/20	7/10	3/4	3/3	2/2	1/1
	cnt	16/20	9/10	3/4	3/3	2/2	1/1

B REBALANCING CAN MESSAGES

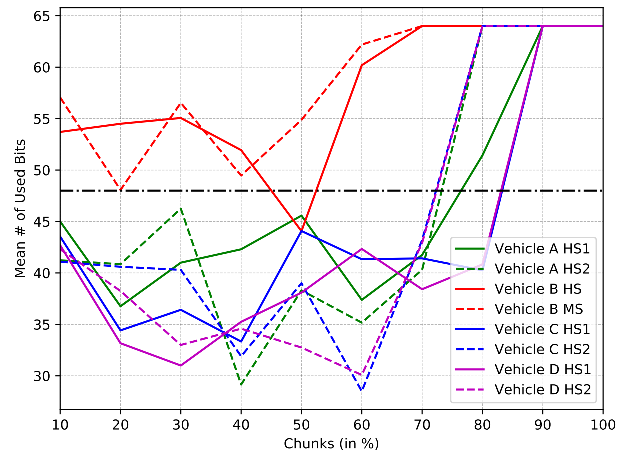

Figure 6: Relationship between Free Space and Message Priority